A synchronous functional language with integer clocks

Adrien Guatto

THÈSE DE DOCTORAT
de l'Université de recherche
Paris Sciences et Lettres -
PSL Research University

Préparée à
l'École normale supérieure

A Synchronous Functional Language with Integer Clocks

par Adrien Guatto

École doctorale n°386
Spécialité : Informatique
Soutenue le 07/01/2016

Composition du jury :

M. Gérard Berry
Professeur, Collège de France
Président

Mme. Mary Sheeran
Professeur, Chalmers University
Rapporteur

M. Robert de Simone
Directeur de recherche, INRIA Sophia-Antipolis
Rapporteur

M. Stephen Edwards
Professeur, Columbia University
Examinateur

M. Dan Ghica
Professeur, Birmingham University
Examinateur

M. Marc Duranton
Senior Research Scientist, CEA
Invité

M. Albert Cohen
Directeur de recherche, INRIA Paris
Directeur de thèse

M. Marc Pouzet
Professeur, École normale supérieure et UPMC
Codirecteur de thèse

M. Louis Mandel
Research Scientist, IBM Research
Encadrant de thèse

ENS ÉCOLE NORMALE SUPÉRIEURE | PSL RESEARCH UNIVERSITY PARIS

Remerciements

I would like to thank profusely the members of my committee for their participation. I am especially grateful for the patience shown by Robert de Simone and Mary Sheeran during the review process. I would also like to thank Dan Ghica and Mary Sheeran for having invited me to Birmingham and Göteborg, respectively.

Mes années de thèse au sein de l'équipe PARKAS ont été les plus enrichissantes qu'il m'ait été donné de vivre jusqu'ici. Je profite de ces quelques paragraphes pour remercier les personnes qui ont contribué à cette richesse.

J'aimerais avant tout remercier mes deux directeurs de thèse, Albert Cohen et Marc Pouzet, ainsi que mon encadrant, Louis Mandel. Ils ont chacun su mettre leurs qualités propres au service de la construction d'un environnement scientifique et humain exceptionnel. J'aimerais en particulier rendre grâce à la générosité d'Albert, à la droiture et à la créativité de Marc, et à la bonté de Louis. Chacun d'eux est à sa manière un modèle pour moi.

Ma thèse s'inscrit dans la droite ligne de celle de Florence Plateau. Son élégance m'a séduit et sa clarté a beaucoup facilité mon travail. C'est une oeuvre difficile à égaler.

J'ai été accueilli au début de thèse par les grands anciens, Léonard Gérard et Cédric Pasteur, avec qui j'ai pu travailler sur le compilateur Heptagon. Ce fut une expérience formatrice à bien des égards. Léonard en particulier m'a transmis une bonne partie de son point de vue sur notre sujet d'étude commun. Merci à vous deux.

J'ai partagé le bureau B11 avec Francesco Zappa Nardelli et Nhat Minh Lê durant la seconde moitié de ma thèse. Merci à Francesco pour les conseils, pénétrants, et les bières, désaltérantes. Merci à Nhat pour notre amitié, qui a grandement éclairé ma thèse.

Guillaume Baudart est un peu mon petit frère de thèse, même s'il me semble que c'est normalement à l'aîné de supporter les sautes d'humeur du cadet plutôt que l'inverse. J'espère pouvoir continuer à gravir la montagne académique à ses cotés, avec ou sans Geneviève.

Timothy Bourke transforme magiquement tout endroit où il se trouve en un lieu sympathique et intéressant. Cela fonctionne même pour la cantine de l'ENS !

Je remercie également tous les autres membres de PARKAS avec qui j'ai pu interagir durant ces quatre années. Merci donc à Cédric Auger, Thibaut Balabonski, Riyadh Baghdadi, Ulysse Beaugnon, Guillaume Chelfi, Boubacar Diouf, Mehdi Dogguy, Brice Gelineau, Tobias Grosser, Jun Inoue, Louis Jachiet, Michael Kruse, Cyprien Lecourt, Feng Li, Antoine Madet, Cupertino Miranda, Robin Morisset, Antoniu Pop, Pablo Rauzy, Chandan Reddy, Konrad Trifunovic, Ramakrishna Upadrasta, Jean Vuillemin, Zhen Zhang, Jie Zhao et aux éventuels oubliés.

Le personnel administratif du Département Informatique de l'ENS est tout à la fois char-

mant et d'une efficacité qui dépasse bien souvent ce que le professionnalisme seul impose. Merci à Lise-Marie Bivard, Isabelle Delais, Joëlle Isnard et Valérie Mongiat. Merci également aux assistantes de l'équipe PARKAS à l'INRIA, Anna Bednarik et Assia Saadi.

Le *workshop* annuel SYNCHRON rassemble les chercheurs de la communauté "langages synchrones". Y participer a toujours été une expérience très agréable. Merci à Benoît Caillaud, Gwenaël Delaval, Alain Girault, Nicolas Halbwachs, Erwan Jahier, Florence Maraninchi, Michael Mendler, Xavier Nicollin, Pascal Raymond et Lionel Rieg.

J'ai eu plaisir à discuter avec Jean-Christophe Filliâtre et Paul-André Melliès à chacunes de leur venues dans le passage saumon. L'érudition et la gentillesse de Jean-Christophe en font un interlocuteur privilégié pour toute personne s'intéressant aux langages de programmation. Paul-André incarne un point de vue radical sur la science informatique qui me convainc un peu plus chaque jour. Merci à eux.

Pendant mes trois premières années de thèse, j'ai eu la chance d'enseigner à l'Université Pierre et Marie Curie ainsi qu'à Polytech'UPMC. Merci à Olivier Marchetti pour les cours de langage C et d'architecture des ordinateurs. Merci à Emmanuelle Encrenaz pour le cours de langages synchrones. Merci à Francis Bras pour le cours-projet d'électronique, où j'ai pu découvrir que les circuits ne transmettent pas que des zéros et des uns.

J'ai fait durant mes études universitaires des rencontres qui se sont avérées déterminantes, et à qui je dois en large partie cette thèse. Je remercie, dans l'ordre chronologique, Christian Queinnec, Karine Heydemann, Mathieu Jaume et Thérèse Hardin.

J'ai eu le plaisir de fréquenter le Groupe de Travail Logique de l'ENS depuis ses débuts. J'y ai trouvé un cadre ouvert et accueillant pour les gens intéressés par l'articulation entre logique et calcul. Merci donc à Marc Bagnol, Aloïs Brunel, Guillaume Brunerie, Charles Grellois, Baptiste Mélès, Guillaume Munch-Maccagnoni, Pierre-Marie Pédrot, Maël Pégny, Silvain Rideau, Gabriel Scherer, Anne-Sophie de Suzzoni et les autres.

Les personnes qui suivent comptent beaucoup pour moi. Merci, en vrac : aux salvateurs Kévin, Sylvain, Alexandre et Camille; à Alix et Olga, si proches et si importantes; à Choupichoups, la meilleure équipe d'EUW; à Maël, le type le plus chouette de l'univers; à Gaïa également; à Nhat, le kenshiro de la choupitude; à Gucile, pour la Troisième République; à Bruno, le plus écossais de tous les français; à Anthony, troll humaniste; à Aurore, que j'admire; aux chimistes et ex-chimistes Gabriel, Jacques-Henri, Thomas et Peter McDag; à PIM, matérialiste éhonté; à Guillaume-des-couettes, pour sa poker-face; à Marc, pour les traces qu'il laisse; à Charles, pour sa rematérialisation; à Annso, pour sa capacité à être "complètement Roi Lion"; à Laure pour son amitié et son soutien, et pour l'invitation à Lyon; à Olivier, qui est souvent un modèle; à Karine, ainsi qu'au couple royal Homer et Heidi; à Mam'zelle, roturière et fière de l'être.

Je remercie également ma tante, mon oncle, ma cousine, mon cousin et sa femme. Je suis heureux de vous avoir et espère vous voir souvent.

Enfin, je remercie ma mère, pour tout.

Résumé

Cette thèse traite de la conception et implémentation d'un langage de programmation pour les systèmes de traitement de flux en temps réel et à haute performance, comme l'encodage vidéo. Le modèle des réseaux de Kahn est bien adapté à ce domaine et y est couramment utilisé. Dans ce modèle, un programme consiste en un ensemble de processus parallèles communicant à travers des files mono-producteur, mono-consommateur. La force du modèle réside en son déterminisme.

Les langages synchrones fonctionnels comme Lustre sont dédiés aux systèmes embarqués critiques. Un programme Lustre définit un réseau de Kahn qui est *synchrone*, c'est à dire, qui peut être exécuté avec des files bornées et sans blocage. Cette propriété est principalement garantie par un système de types dédié, le *calcul d'horloge*, qui établit une échelle de temps globale à un programme. Cette échelle de temps globale est utilisée pour définir les *horloges*, séquences booléennes indiquant pour chaque file, et à chaque pas de temps, si un processus produit ou consomme une donnée. Après le calcul d'horloge vient l' *analyse de causalité*, qui garantit l'absence de blocage. Les programmes corrects du point de vue des horloges et causaux sont compilés vers des machines à état fini, réalisés soit en logiciel soit en matériel.

Nous proposons et étudions les *horloges entières* dans le but d'utiliser les langages synchrones fonctionnels pour programmer des applications de traitement de flux. Les horloges entières sont une généralisation des horloges booléennes qui comprennent des entiers arbitrairement grands. Elles décrivent la production ou consommation de plusieurs valeurs depuis une même file au cours d'un instant. Nous les utilisons pour définir la notion d'*échelle de temps local*, qui peut masquer des pas de temps cachés par un sous-programme au contexte englobant. Les échelles de temps local sont introduites par une opération dite de *rééchelonnement*.

Ces principes sont intégrés à un calcul d'horloge pour un langage fonctionnel d'ordre supérieur. Ce système de types capture en une unique analyse toutes les propriétés requises pour la génération d'une machine à état fini. En particulier, il intègre et étend l'analyse de causalité trouvée dans les langages synchrones fonctionnels existants. Nous étudions ses propriétés, prouvant entre autres résultats que les programmes bien typés ne bloquent pas. Nous adaptons ensuite le schéma de génération de code dirigé par les horloges de Lustre pour traduire les programmes bien typés vers des circuits numériques synchrones. La génération de code est modulaire : il n'est pas nécessaire de recompiler les appelants d'une fonction lorsque le corps de celle-ci change, aussi longtemps que son type reste le même. L'information de typage contrôle certains compromis entre temps et espace dans les circuits générés.

Mots-clés : langages de programmation fonctionnels; langages de programmation synchrones; systèmes de types; compilation; circuits numériques synchrones.

Abstract

This thesis addresses the design and implementation of a programming language for high-performance real-time stream processing, such as video decoding. The model of Kahn process networks is a natural fit for this area and has been used extensively. In this model, a program consists in a set of parallel processes communicating through single reader, single writer queues with blocking reads. The strength of the model lies in its determinism.

Synchronous functional languages like Lustre are dedicated to critical embedded systems. A Lustre program defines a Kahn process network which is *synchronous*, that is, which can be executed with finite queues and without deadlocks. This is mainly enforced by a dedicated type system, the *clock calculus*, which establishes a global time scale throughout a program. The global time scale is used to define *clocks*: per-queue boolean sequences indicating, for each time step, whether a process produces or consumes a token in the queue. After the clock calculus comes the *causality analysis*, which guarantees the absence of deadlocks. Well-clocked and causal programs are compiled to finite state machines, realized either as hardware or software code.

We propose and study *integer clocks* in order to extend the reach of synchronous functional languages to high-performance real-time streaming applications. Integer clocks are a generalization of boolean clocks that feature arbitrarily big natural numbers. They describe the production or consumption of several values from the same queue in the course of a time step. We use integer clocks to define the notion of *local time scale*, which may hide time steps performed by a sub-program from the surrounding context. Local time scales are introduced by an operation called *rescaling*.

These principles are integrated into a clock calculus for a higher-order functional language. This type system captures in a single analysis all the properties required for the generation of a finite state machine. In particular, it subsumes and extends the causality analysis found in other languages. We study its properties, proving among other things that well-typed programs do not deadlock. We then adapt the clock-directed code generation scheme of Lustre to translate typed programs to digital synchronous circuits. Code generation is modular: one does not need to recompile the callers of a function when its body changes, as long as its type is not modified. The typing information controls certain trade-offs between time and space in the generated circuits.

Keywords: functional programming languages; synchronous programming languages; type systems; compilation; digital synchronous circuits.

Contents

Notations

$\mathbf{1} = \{*\}$	Singleton set
$x \mapsto f(x)$	Set-theoretic function
$X \times Y$	Cartesian product of sets
$X \uplus Y = \{\iota_1 x \mid x \in X\} \cup \{\iota_2 y \mid y \in Y\}$	Disjoint union of sets
$f(X') = \{f(x') \mid x' \in X'\}$	Image of a set $X' \subseteq X$ under a function $f : X \to Y$
$Im(f) = f(X)$	Image of function $f : X \to Y$
$X \hookrightarrow Y$	Inclusion map of X into Y; supposes that $X \subseteq Y$
$\mathcal{P}(X)$	Powerset of set X
X^*	Finite sequences of elements of X
X^+	Non-empty finite sequences of elements of X
X^ω	Infinite sequences of elements of X
$X^\infty = X^* \cup X^\omega$	Finite or infinite sequences of elements of X

Chapter 1

Introduction

1.1 Real-Time Stream Processing

Most human beings experience computing by interacting with personal computers, smartphones, or tablets. But while such user-facing systems are numerous, they are dwarfed by the vast cohorts of computers that carry out autonomous supervision, control, and decision tasks. Computers have indeed found their way into a variety of settings, including for example plant control, autonomous vehicles, automatic trading, or network processing. In contrast with traditional batch-oriented computing, the execution of programs involved in such systems proceeds as a succession of reactions to an external environment. The precise nature of this environment varies; common cases include the physical world, accessed through sensors, or a network, accessed through an interface card. Reactions result in the production of new information which may then be transmitted through the network or used to drive actuators.

This thesis is about programming the specific subclass of interactive computing systems that perform what we call *real-time stream processing*. They are characterized by the fact that they perform an unbounded number of reactions, each taking a bounded amount of time and memory. In practice, such systems execute forever while consuming a finite amount of resources. Examples include critical systems, such as the fly-by-wire software found in planes, but also more computationally-intensive tasks like real-time video encoding or decoding, scientific data acquisition, and low-latency packet routing. These systems consist in either software, hardware, or a mixture of both.

Real-time stream processing is often implemented in languages such as C, assembly, or hardware description languages such as VHDL or Verilog. This is sometimes considered part and parcel of this application domain, with resource constraints mandating a programming style where low-level details have to be handled with utmost care and tedious precision. In this thesis we adopt a different point of view by contributing to the design and implementation of specialized programming languages for real-time stream processing. We hope that higher-level languages will lead to stronger safety and efficiency guarantees, as well as improved programmer productivity. Moreover, we think that such an investigation is interesting in itself because it helps uncover interesting pieces of programming language theory.

1.2 Kahn Process Networks and Synchrony

One may think of a stream processing application as an assemblage of processes executing concurrently. The processes may have internal state. Each of them communicates with its neighbors or the environment via message exchanges through dedicated channels.

While this description is reminiscent of distributed systems, it is actually much simpler. Message inversion, duplication, and loss do not occur. Security and fault-tolerance are also non-issues. A few simple primitives suffice to describe the communication patterns of each process. Message reception is blocking: a process deciding to read an incoming message is suspended until the message is effectively received. Furthermore, a process may only read on a unique input channel at a time. Contrast with the select() function or the timeouts provided by UNIX. An important consequence of these restrictions is determinism: assuming that the sequence of output messages of each process depends only on its sequence of input messages, the same is true of the application as a whole.

Kahn process networks The description of stream processing given above is both vague and rather operational in nature. In a seminal paper, Kahn [1974] advocates a more abstract point of view based on elementary domain theory. He interprets processes as mathematical functions and channels as mathematical streams, which are the infinite sequences of values transmitted through them. The functions corresponding to processes enjoy special properties that reflect the aforementioned discipline in the use of communication primitives. Such properties ensure that the general theory of domains applies; the benefit is that this theory immediately provides useful constructions and results. For instance, one may use domain theory to describe the concurrent and sequential composition of two networks, or the creation of feedback loops. It also gives basic results such as determinism for free and includes rigorous proof techniques for recursively defined streams.

The account given by Kahn of stream processing was so influential that nowadays an application belonging to this framework is often referred to as a *Kahn Process Network* (KPN). This is often used in a broad sense that do not involve denotational ideas: a concurrent application is a Kahn process network when its communications obey the unique blocking read discipline, making it deterministic.

Synchrony The original paper by Kahn proposed the use of domain theory for *modeling* stream-processing applications. Reading between the lines, the proposed methodology was to give a denotational description of an imperative program and then use the clean formalism of domain theory to prove properties of this description.

There are several problems with this initial idea. First, it is not very practical as it forces the programmer to deal with two very different languages, an imperative language for writing programs and a mathematical metalanguage for proving properties. Second, real-world stream processing must often deal with resource constraints. This often lead to optimizations that obscure the logical structure of the application, such as the implementation of communications over shared memory and the sequentialization of concurrency. Moreover resource constraints

are difficult to address in the idealized denotational setting employed by Kahn, which has little to say about space and time usage.

To circumvent these limitations while preserving the original insight, Caspi, Pilaud, Halbwachs, and Plaice [1987] put forward an original idea. The idea is to reverse the roles: rather than write an imperative program and then interpret it in a domain of streams and functions, we will program directly with streams and then generate imperative code. The goal is to get the best of both worlds: simple high-level programs one can reason rigorously about, compiled to resource-conscious low-level code.

The concrete embodiment of this idea is Lustre, a domain-specific functional programming language where streams are tangible values rather than ethereal mathematical objects. Any Lustre program describes a Kahn process network, and the role of the Lustre compiler is to produce an implementation obeying resource constraints. More precisely, since Lustre targets real-time critical systems, the generated code must be sequential, and work within bounded memory and terminate in bounded time. Programs for which the compiler cannot produce such an implementation are rejected.

The inner working of a Lustre compiler is based on a principle called *synchrony*. Synchrony is a sufficient condition for a Kahn network to be implementable within bounded space. Two streams are synchronous when their elements are computed simultaneously. This involves a notion of time step shared between subprograms; one goal of the Lustre compiler is to actually establish such a time base, and use it to check synchrony. The notion is actually so integral to Lustre that the language is generally designated as a *synchronous language*.

Operationally, the fact that all streams in a Kahn network are synchronous means that any value that is produced can be consumed instantaneously. This has several important consequences for its compilation to sequential imperative code. First, streams can be implemented as local variables since no buffering between time steps is required. Second, during the course of a time step, the production of an element must occur before its consumption. A Lustre program where it is indeed the case is said to be *causal*, a property which is also enforced by the compiler. This ensures that the underlying Kahn network is deadlock-free.

Compilation We can now give a broad overview of the most idiosyncratic stages of a Lustre compiler following the principles outlined above. This is a very schematic view that we will detail and refine during the course of the thesis.

First, the compiler checks for synchrony. This is done using a dedicated static analysis, the *clock calculus*. This consists in associating to each stream a *clock*, which describes the time step at which its elements are computed. Technically, a clock is a boolean stream which represents the presence and absence of data in another stream. Two streams that have the same clock are necessarily synchronous. The clock calculus is often implemented as a type system [Caspi and Pouzet, 1996].

Second, the compiler checks that the program is causal. This is what its *causality analysis* [Cuoq and Pouzet, 2001] does, by checking that during a time step, no cyclic dependency between computation occurs. Lustre proposes a special *delay* operator which makes it possible to transmit a value from one time step to the next. As its current output never depends on its

current input, this operator is handled in a special way by the causality analysis. This is why the definition of recursive streams is possible as long as self-references occur through delays.

Finally, the compiler may generate code implementing the Kahn network. A traditional focus of Lustre has been bounded-memory C code, but this is not the only possibility. An interesting alternative is the generation of synchronous digital circuits, which are not very different from causal synchronous Kahn networks anyway. Targeting sequential software code actually requires more work than targeting digital circuits, since each program piece has to be *scheduled* so that productions occurs before consumptions. In a circuit this scheduling is done in a dynamic and implicit way by electrical currents. Schedulability is a sufficient condition for causality, and is in practice also checked by the causality analysis.

1.3 Synchrony and Performance

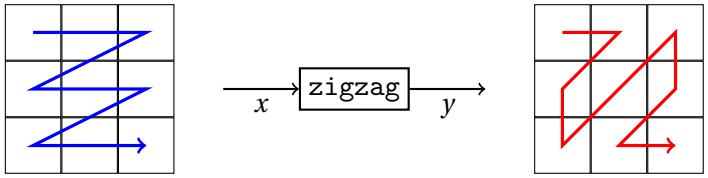
Lustre and its compilation technique based on synchrony has turned out to be well-adapted to the implementation of critical control systems. Its industrial variant, SCADE [Esterel Technologies, 2015], is widely used in aircrafts, nuclear plants, trains, or industrial systems. The success of Lustre can be traced back to several factors. An important practical reason is its proximity to difference equations, the traditional formalism used by control scientists. Another explanation which is closer to our concerns is the fact that control programs involve inexpensive but tightly synchronized computations for which synchrony is a natural condition.

In other subfields of stream processing, the situation is quite different. Consider for instance a real-time video processing program. Such a program naturally decomposes as a pipeline of filters. Each filter performs thousands of arithmetic operations on image chunks, and communicates with its neighbor by rare but large messages. Implementing such programs in Lustre is difficult, mainly because of the synchrony condition. Concretely, synchrony forces the programmer to introduce buffering by hand. This is a tedious and error-prone process without tool support. In practice it leads to low-level code in which the static analyses of the compiler are more of a hindrance than help.

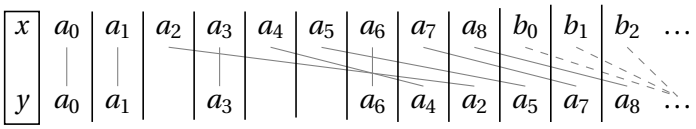
An important step towards addressing this limitation of synchrony was its generalization to *n-synchrony* by Cohen et al. [2006]. Communication between synchronous streams can be implemented with a buffer of size zero, that is with no buffer at all. Communication between n-synchronous streams can be implemented with a buffer of size n. The clock calculus of Lustre can be extended to check n-synchrony rather than ordinary 0-synchrony. Lucy-n [Mandel et al., 2010] is an experimental n-synchronous variant of Lustre in which the compiler infers clocks and buffer sizes. Interestingly, there are several valid clocks for the same program; distinct clocks describe distinct implementations of the Kahn network as a state machine, and may express space/time trade-offs.

The adoption of n-synchronous Kahn networks makes possible the automatic inference of buffers and buffer sizes from high-level code manipulating streams, avoiding manual coding altogether. But it does not solve another problem that arise when trying to implement efficient stream processing in Lustre-like languages. High-performance stream processing frequently involve sporadic but large transfers of data on which uniform treatments are applied. For effi-

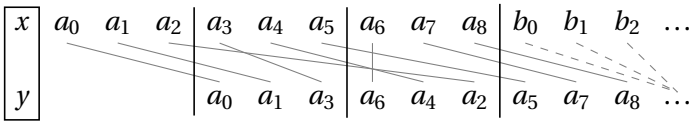
The JPEG image encoding process is organized as a series of transformations. The three last steps are the *discrete cosine transform* (DCT), followed by the *zigzag scan*, before finally going through *entropy coding*. We focus on real-time implementations of the zigzag scan.



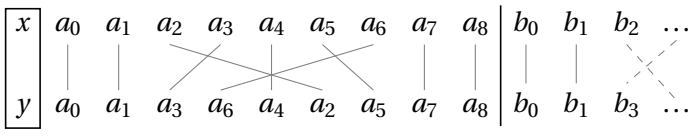
The goal of the zigzag scan is to reorder the numbers produced by the DCT in order to expose redundancies exploitable by the entropy coder. In a real-time JPEG encoder, the input of the zigzag scan would typically consist in a stream of elements of matrices serialized in the left-to-right, top-to-bottom order. Its output is a stream featuring the same elements, reordered by "zigzaging" along the diagonal of each matrix. The figure above describes this process for 3x3 matrices.



The chronogram above describes a real-time implementation of the zigzag scan that processes at most one element per time step. The first line represents the input stream x, produced by the DCT, the second the output stream y, fed to the entropy coder. Thick vertical gray bars represent time steps and thin black lines represent dependencies. The dependencies that cross time steps must be implemented using memory.



Another possibility is to process the input stream matrix line by matrix line, as in the chronogram above. More values are produced and consumed per time step, and the amount of memory needed changes.



Finally, an implementation may process a full matrix at each time step, as above. As shown in the chronogram above, no memory is needed. However, such an implementation will be more demanding in terms of code size or circuit area.

Figure 1.1: The JPEG Zigzag Scan

ciency reasons, one generally wants the generated code to take advantage of this loose coupling between computations, which is a form of *locality*. In software, programmers use *blocking* to add or modify loops and arrays in judicious places; this is particularly common in signal processing code. In hardware, circuit designers use *retiming, slowdown, and unfolding* to trade between throughput, latency, area, and frequency (e.g., Leiserson and Saxe [1991]; Parhi and Messerschmitt [1991]). Figure 1.1 illustrates implementation choices for a simple function.

A direct solution to this second problem is to introduce arrays in the source language and use them to program batched computations and communications. Unfortunately, while the extension of Lustre with arrays and array operators is well-studied [Morel, 2002; Gérard et al., 2012], the integration between array-oriented code and scalar code is not as tight as one could wish for. It is difficult to translate from one style, or even particular array size, to another. For example, in order to transform a stream of scalars into a stream of arrays, the programmer must introduce a hand-rolled serial-to-parallel converter. The amount of memory needed by such a converter depends on the rest of the program, and must be guessed by the programmer. This is brittle and error-prone. The same is true in other synchronous languages that feature arrays, such as Esterel v7 [Esterel Technologies, 2005]. This is problematic when exploring performance trade-offs in the implementation of a given function.

1.4 Contributions

In this thesis we propose several extensions to Lustre-like languages in order to make them more suitable for programming real-time stream processing systems. In particular, we address the aforementioned issues with the exploitation of locality and decoupling in high-performance applications. We believe that our contributions improve both the expressiveness of the language, its compilation process, and its metatheory.

Integer clocks In Lustre, at most one element of a given stream is computed during a time step. This is reflected by the fact that clocks are *boolean* streams. We relax this restriction by allowing arbitrarily large numbers to appear in clocks; we call this extension *integer clocks*. This models the simultaneous computation of several elements of a stream. We extend the clock calculus and code generation of Lucy-n to deal with integer clocks. The code generation scheme now produces code with arrays from a purely scalar source. The size of arrays is completely determined by clocks, and serial/parallel conversions are implicit. A benefit is the smooth interoperability between scalar and array-oriented programs.

Local time scales and rescaling A natural complement to integer clocks is the notion of *local time scale*. A subprogram that executes within a local time scale may go faster or slower than the outside world, and thus possibly hide some of its internal steps. The operation that creates a local time scale around a block of code is called *rescaling*. Rescaling is a generalization of the traditional *activation conditions* of Lustre and SCADE (see, e.g., Halbwachs [2005, Section 4]). We explain its action on clocks and give a typing rule characterizing correct uses. Each rescaling gives rise to a loop (in software) or an unfolded circuit (in hardware). A programmer

may trade space for time in the implementation of a function by combining rescaling and integer clocks, while being guaranteed that the final code.

Linear higher-order functions Lustre is a first-order functional language. The addition of higher-order functions has been studied in the Lucid Synchrone language of Caspi and Pouzet [1996], but in existing work its implementation relies on dynamic memory allocation or is incompatible with separate compilation. Inspired by recent work in category theory and hardware-oriented compilation [Joyal et al., 1996; Ghica, 2007], we propose submitting higher-order functions to a linear typing discipline. This restriction enables modular compilation to finite-state code.

Simplified metatheory We combine integer clocks, rescaling, and linear higher-order functions in a single type system. Thanks to rescaling we are able to express causality checking at the level of clocks. Thus, our type system fuses the separate typing, clock calculus, and causality analysis of existing languages into a single system. We believe that this type system is actually simpler than the previous ones. To demonstrate this fact we give full rigorous proofs of metatheoretical results, such as deadlock-freedom and refinement for well-typed programs, as well as a formal type-directed compilation scheme to circuits. This development relies on standard concepts of programming language theory.

1.5 Outline of the Thesis

The thesis consists in a description of AcidS, a tentative Lustre-like language featuring a novel treatment of clocks, activation conditions, and higher-order functions. We consider its core to be Chapters 2 and 4, which explain in much greater detail the points touched upon in this introduction. The thesis should be read in a sequential manner, as every chapter depends on all the previous ones.

Chapter 2 The first technical chapter of this thesis details the mathematical description of Kahn networks. We give a brief introduction to elementary denotational semantics before spending time on the domain of streams. The rest of the chapter describes the theory of integer clocks and how they characterize operational properties in the denotational setting.

Chapter 3 In the next chapter we describe μAS, a core higher-order synchronous language akin to Lucy-n. The language is endowed with a clock type system featuring ultimately periodic integer clocks, rescaling, and linear higher-order function. Programs have both untyped and typed denotational semantics built using the notions introduced in the previous chapter. We prove that for well-typed programs, the typed semantics is a deadlock-free synchronization of the untyped one, in a certain technical sense. These properties correspond to the soundness of the type system.

Chapter 4 We introduce a small language of machines for modeling synchronous digital circuits. This language is typed, first-order, and has a simple non-deterministic operational semantics. We build a higher-order layer on top of the first-order base using a source-to-source translation. The resulting higher-order machine language serves as a target for the type-directed code generation process of μAS. We prove that compilation is sound via a logical relation defined using a model of the target language.

Chapter 5 As a programming language, μAS is very restrictive, lacking traditional features from Lustre. The chapter presents four language extensions. The first one relaxes the linearity restriction on higher-order functions by allowing a function to be called a statically-fixed number of times. The second one shows how *nodes*, i.e. closed functions, can be reused at will as in Lustre. We finish with two extensions that increase the expressiveness of the type system, first with polymorphism and then with data dependency. The latter makes it possible to write programs whose behavior is not always periodic. These extensions, combined together, form AcidS.

Chapter 6 This thesis ends with a summary of related and future work. We compare our proposal to existing synchronous, dataflow, and functional languages, as well as analytic models for streaming systems such as Synchronous Dataflow graphs of Lee and Messerschmitt [1987]. Several issues remain to be addressed before a realistic implementation of AcidS is possible. We discuss type inference and software code generation in detail, benefiting from our experiments with the implementation of a prototype compiler. The chapter finishes with a compendium of short- as well as long-term theoretical questions.

Chapter 2

Streams

As explained in the first part of the thesis, synchronous functional languages are mainly concerned with the construction and manipulation of infinite sequences of data, called *streams*. They belong to the family of so-called *dataflow* synchronous languages because of the prominent role of streams; but in contrast with a language such as Signal [Le Guernic et al., 1991], programs denote functions rather than relations. A program written in a synchronous functional language describes a deterministic parallel network of operators connected through queues.

This vision dates back to the seminal article of Kahn [1974]. He studied networks of imperative processes communicating only through single-producer, single-consumer channels via blocking reads. Such programs are deterministic because blocking reads force processes to be *latency-insensitive*: once a process starts reading an input channel, it has no choice but to wait until a value arrives. Thus, communication delays or scheduling choices cannot affect the values computed by processes in any way.

Kahn argued that such programs were best described as set-theoretic functions rather than state machines, since in general state machines are more difficult to describe and manipulate mathematically. The inputs and outputs of these functions are streams that represent the *histories* of values transmitted through each input or output queue of a process. Yet, in order to make this *processes-as-functions* idea rigorous, one needs to overcome two roadblocks.

- One needs to model processes that produce only a finite amount of items on their output queues. In the extreme case, a process may not even produce anything on a given queue, and other processes trying to read from this same queue will get stuck forever. This may lead to partial or total deadlocks in the network.

- One should provide mathematical analogues to the ways in which processes may be composed. On the one hand, it is clear that connecting the outputs of one process to the inputs of another corresponds to ordinary function composition, or that one can use the cartesian product of streams and its associated projections to model processes having multiple inputs or multiple outputs. On the other hand, process networks also feature arbitrary feedback loops whose set-theoretic meaning is less evident. This seems to indicate that the functions in consideration should admit a fixpoint operation.

These two difficulties were solved by Kahn through the use of domain theory, introduced by Scott [1969] a few years earlier and intensively studied since. Domain theory provides a simple order-theoretic framework for describing computations between large or even infinite objects, such as streams. Its fundamental insight is that computable functions, when acting on infinite objects, are fully characterized by their action on *finite approximations* to these objects. This suggests the use of *partially ordered sets* where the order describes this notion of approximation, and where least upper bounds of sets of compatible approximants exist. Computable functions then naturally correspond to *continuous* functions, a class of functions behaving well with respect to approximation.

The notion of approximation provides a solution to the first problem above: the domain of streams consists not only in infinite streams but also in their finite approximants, which can be thought of as finite prefixes. Moreover, a continuous function always has a least fixpoint, which thus provides a natural way to model feedback loops, solving the second problem. Thus, in theory, one can define process networks as continuous functions, and use the tools and proof techniques provided by domain theory for reasoning, avoiding stateful code.

The language proposed in this thesis describes Kahn process networks with additional properties, such as deadlock-freedom and bounded buffers. Its mathematical semantics relies on the ideas introduced above. The goal of this chapter is to introduce them formally and describe some of their properties, paving the way for the mathematical study of the language in the rest of this thesis. Following Kahn, we will make elementary use of domain theory. The first section of this chapter provides a very brief introduction with pointers to more complete treatments.

Synchronous functional languages generally define a notion of *clock* to model the scheduling of Kahn networks; we will not depart from this rule. The clocks studied in this thesis, the so-called *integer clocks*, generalize previous definitions and will enable a finer-grained study of dependencies between computations. We develop the theory of integer clocks by reusing and extending notations and results from Plateau [2010].

2.1 Domains

Most of the formal developments in this thesis rely either implicitly or explicitly on elementary *Domain Theory*. Domain Theory is the mathematical theory of special kinds of ordered sets, called *domains*, and structure-preserving functions between them. It provides a framework for building denotational semantics of programming languages, where one interprets programs as mathematical objects and studies the objects obtained in the hope that they will shed light on properties that are difficult to grasp from a syntactic point of view.

This section gives a hurried introduction to the elementary aspects of Domain Theory used in this thesis. We merely state the theorems we need and direct the curious reader to the bibliography given in the end of this section for proofs and additional explanations. Our notations are mostly taken from the monograph of Abramsky and Jung [1994].

Partial Orders A (partial) *ordering relation* on a set X is a binary relation $\sqsubseteq$ that is reflexive, transitive and antisymmetric. Then, given $x, y, z \in X$, we say that z is an *upper bound* of x and y when $x \sqsubseteq z$ and $y \sqsubseteq z$. We say that a non-empty subset F of X is *directed* when any pair of elements of F has an upper bound in F. When it exists, we write $\bigsqcup F$ for the least upper bound, or *lub*, on an arbitrary subset F of X. Similarly, given $x, y \in X$ we write $x \sqcup y$ for $\bigsqcup\{x, y\}$ when it exists.

Given two ordered sets $(X, \sqsubseteq_X)$ and $(Y, \sqsubseteq_Y)$, a function $f : X \to Y$ is said to be *monotonic* when it is order-preserving i.e., for any $x, y \in X$, $x \sqsubseteq_X y$ implies $f(x) \sqsubseteq_Y f(y)$.

Domains and continuous functions We call *predomain* a *directed-complete* partial order. Such a partial order D consists in a pair $(|D|, \sqsubseteq_D)$, that is a set $|D|$ endowed with a partial order $\sqsubseteq_D$ such that given any directed subset $F \subseteq |D|$, the least upper bound $\bigsqcup F$ of F exists in $|D|$. Such a predomain D is a *domain* when there is an element $\bot_D \in |D|$ minimal for $\sqsubseteq_D$. We drop the indices from $\sqsubseteq_D$ and $\bot_D$ when D can be deduced from the context, and sometimes implicitly identify D with $|D|$.

We say that a subset X of $|D|$ is a *sub-domain* of D when it contains $\bot_D$ and the least upper bounds $\bigsqcup F$ of all directed sets $F \subseteq X \subseteq |D|$, considered with regard to $\sqsubseteq_D$.

A function $f : A \to B$ between (pre)domains A and B is continuous when it preserves the least upper bounds of directed sets. Such functions are necessarily monotonic. The identity function is continuous and function composition preserves continuity. We write $A \to_c B$ for the set of continuous functions between A and B. Two such functions $f, g \in A \to_c B$ can be compared, with $f \sqsubseteq g$ when $\forall x \in A, f(x) \sqsubseteq g(x)$. This ordering relation is sometimes called the *extensional* order.

Fixpoints Suppose given a domain D and a continuous function $f : D \to_c D$. Let us write f^i for f composed i times with itself, that is $f^0 = id$ and $f^{i+1} = f \circ f^i$.

Theorem 1 (Kleene's fixpoint theorem)**.** *The least fixed point* **fix** f *of* f *exists and is such that*

$$\mathbf{fix}\, f = \bigsqcup_{i \geq 0} (f^i \bot)$$

This result makes it possible to take fixpoints of arbitrary continuous functions as long as one deals with extra $\bot$ elements. This reflects the fact that in a Turing-complete programming languages one may define arbitrary recursive functions, at the price of allowing non-termination.

Special functions We now introduce some vocabulary regarding important properties of continuous functions between (pre)domains.

- *Strictness*: a function between domains is said to be *strict* when it preserves least elements.

- *Deflation*: a *deflation* is a continuous function $f : D \to_c D$ such that $f \sqsubseteq id_D$ and $f \circ f = f$. Deflations are sometimes called *projections* in the literature. We choose to use the term "deflation" to avoid ambiguities with regard to the next definition.

- *Embeddings and projections*: an embedding-projection pair (e, p) between two predomains A and B is a pair of continuous maps $e : A \to_c B$ and $p : B \to_c A$ such that we have $p \circ e = id$ and $e \circ p \sqsubseteq id$. We then write $(e, p) : A \lhd B$, and more generally say that A is a *retract* of B, written $A \lhd B$ when such an embedding-projection pair exists. The composite $e \circ p$ is always a deflation since $e \circ p \circ e \circ p = e \circ id \circ p = e \circ p$. We sometimes abbreviate the term "embedding-projection pair" as *e-p pair*.

- *Isomorphisms*: an isomorphism (i, i^{-1}) between two domains A and B is a pair of continuous maps $i : A \to_c B$ and $i^{-1} : B \to_c A$ such that $i^{-1} \circ i = id_A$ and $i \circ i^{-1} = id_B$. We then write $i : A \cong B$, or just $A \cong B$ when we want to indicate that such an isomorphism exists.

Domain constructors Domains can be constructed from smaller ones in various ways. We describe several domain constructors in turn, and give some programming intuition about the definitions.

- *Lifting*: given a domain D, we may form its lift $D_\perp$ by adding some $\perp \notin D$ and extending the partial order $\sqsubseteq_D$ to make $\perp$ least.

  We can always make a predomain out of a set X by endowing it with the *discrete* order which is $x \sqsubseteq x$ for all $x \in X$. The discrete predomain X can be turned into the *flat* domain $X_\perp$ where $x \sqsubseteq y$ when $x = \perp$ or $x = y$.

  One may think of a set such as $\mathbb{B}_\perp$ as the set of programs whose evaluation either returns a boolean or does not terminate.

- *'Unlifting'*: one can make a predomain $D_\downarrow$ out of a domain D by removing its least element $\perp$. Note that $D_\downarrow$ is not necessarily a domain since it does not have a least element in general.

- *Product*: one can form the product $A \times B$ of two predomains A and B by ordering the cartesian product of the underlying sets componentwise. When A and B are domains, so is $A \times B$ and its least element $\perp_{A\times B}$ is $(\perp_A, \perp_B)$.

  Intuitively, products correspond to *lazy* pairs, where the evaluation of one component may loop without affecting the evaluation of the other. For example, $(\perp, 1) \in \mathbb{B}_\perp \times \mathbb{B}_\perp$ models a program returning a pair whose right component evaluates to *true* while the evaluation of its left component loops forever.

- *Smash product*: the smash product $D_1 \otimes D_2$ of two domains D_1 and D_2 is a cartesian product where the two least elements of D_1 and D_2 have been identified:

$$D_1 \otimes D_2 = \left(D_{1\downarrow} \times D_{2\downarrow}\right)_\perp$$

This product typically models programs returning *strict* pairs. Given an element $x \in \mathbb{B}_\perp \otimes \mathbb{B}_\perp$, either the computation of the whole pair corresponding to x loops and $x = \perp$, or it converges to two booleans b_1 and b_2 and thus $x = (b_1, b_2)$. Observe that there is an embedding-projection pair $(e_\otimes, p_\otimes) : A \otimes B \triangleleft A \times B$.

- *Sum*: one can form the sum of two predomains D_1, D_2 by taking the disjoint union of their elements such that $x \sqsubseteq_{D_1+D_2} y$ just when $x = \iota_i x'$, $y = \iota_i y'$ and $x' \sqsubseteq_{D_i} y'$, with ι_i the injection into the i-th component of a disjoint union. Note that the resulting predomain is never a domain.

- *Coalesced sum*: similarly to the smash product, one may define the coalesced sum $D_1 \oplus D_2$ of two domains D_1 and D_2 as

$$D_1 \oplus D_2 = (D_{1\downarrow} + D_{2\downarrow})_\perp$$

The injections into the disjoint union naturally give rise to two embedding-projection pairs $(e_i, p_i) : D_i \triangleleft D_1 \oplus D_2$ by

$$e_i(x) = \begin{cases} \perp & \text{when } x = \perp \\ \lfloor \iota_i x \rfloor & \text{otherwise} \end{cases}$$

$$p_i(x) = \begin{cases} y & \text{when } x = \lfloor \iota_i y \rfloor \\ \perp & \text{otherwise} \end{cases}$$

Coalesced sums correspond to *strict* sums in the sense that $D_1 \oplus D_2$ does not contain elements of the form $(\iota_i \perp_i)$: one cannot get the tag ι_i without the corresponding value.

- *Function space*: The set of all continuous functions between two domains A and B, ordered extensionally, forms a domain $A \Rightarrow_c B$, with $\perp_{A \Rightarrow_c B}(x) = \perp_B$ as least element.

In addition to their action on domains themselves, domain constructors also give rise to new functions between domains: in category-theoretical terms, they are *functors*. Consider the cartesian product of domains for example: given domains A, A', B, B' and two functions $f : A \Rightarrow_c B$ and $g : A' \Rightarrow_c B'$, we write $f \times g : A \times A' \Rightarrow_c B \times B'$ for the function mapping the pair (x, x') to $(f(x), g(x'))$.

Recursive domain equations When describing programming languages, one often wishes to construct domains defined in terms of themselves. For example, programs written in an untyped language featuring higher-order functions and booleans might be modeled as elements of a domain U such that

$$U \cong \mathbb{B}_\perp \oplus (U \Rightarrow_c U)_\perp \tag{2.1}$$

In the right hand side U occurs in *contravariant* position, that is on the left of an odd number of arrows; this makes the existence of solutions to such equations non-trivial. Another use case

is languages with user-defined recursive data types, such as ML or Haskell. Such languages typically do not restrict the use of arrows in type definitions, thus allowing the type being defined to occur in contravariant positions in its definition.

Because of cardinality issues, the existence of such isomorphisms is not obvious. One of the key achievements–and indeed original motivation–of Domain Theory is to prove that such solutions actually exist for a very large class of equations. In particular, all equations built from flat domains and domain constructors presented above are of this class. We will not work out in detail how the construction proceeds but give a rough sketch.

Solving a recursive domain equation such as 2.1 consists in finding a domain U and an isomorphism $U \cong F(U,U)$, with F the *functor* modeling the right hand side of the equation. Here, a functor is a map sending pairs of domains to domains and pairs of continuous functions to continuous functions. More precisely, given domains A and B, we have a domain $F(A,B)$ and given functions $f : A_2 \to_c A_1$ and $g : B_1 \to_c B_2$, the functor defines a continuous function $F(f,g)$ from $F(A_1,B_1)$ to $F(A_2,B_2)$.

The two arguments of the functor respectively correspond to contravariant and covariant occurrences of U in the right hand side of the equation. For example, in the case of 2.1 one has

$$F(U^-,U^+) = \mathbb{B}_\perp \oplus (U^- \Rightarrow_c U^+)_\perp$$

One can then form the sequence of domains U_n such that $U_0 = \varnothing_\perp$ and $U_{n+1} = F(U_n,U_n)$. It is increasing in the sense that $U_n \lhd U_{n+1}$ for all n, with embedding-projection pairs given by $(e_0,p_0) = (\perp,\perp)$ and $(e_{n+1},p_{n+1}) = (F(p_n,e_n),F(e_n,p_n))$. Now, we can build U as the domain

$$\begin{array}{rcl} |U| & = & \{\prod_{n\in\mathbb{N}} x_n \mid x_n \in U_n \text{ and } x_n = p_n x_{n+1} \text{ for all } n\} \\ x \sqsubseteq_U y & = & x_n \sqsubseteq_{U_n} y_n \text{ for all } n \\ \perp_U & = & n \mapsto \perp_{U_n} \end{array}$$

whose elements are indexed products ordered componentwise. The domain U is the *limit* of the sequence U_n since each U_n embeds into U and that U itself embeds uniquely into any other domain U' such that $U' \cong F(U',U')$. This construction actually requires the functor F to satisfy technical conditions [Streicher, 2006].

Reasoning on domains Domain Theory provides reasoning principles usable for metatheoretical investigations or program proof. Of particular importance is a simple induction principle, often called *Scott induction*. This property can be used to show that the least fixpoint of a function $f : D \to_c D$ satisfies a certain property P, with P identified with a subset of D well-behaved with respect to continuity.

Theorem 2 (Validity of Scott Induction)**.** *Call* admissible predicate *a subset of a domain D that contains $\perp_D$ and is closed under directed lubs. We write $P(x)$ for $x \in P$. Then, given such a P and a continuous function $f : D \to_c D$ the following rule is valid*

$$\frac{\forall x.P(x) \Rightarrow P(fx)}{P(\mathbf{fix}\, f)}$$

Bibliography Domain Theory was introduced by Scott [1969] in the late sixties. The classic book of Reynolds [1996] on programming languages features among other things an introduction to domains and denotational semantics. Winskel [1993] has another good introduction. Streicher [2006] is more advanced and gives a technical yet clear explanation of the solution of recursive domain equations. Abramsky and Jung [1994] give a thorough treatment of the core mathematical aspects of the theory. The book by Amadio and Curien [1998] contains a host of references. Pitts [1996] derives general reasoning principles for recursively defined domains. Finally, all the ideas of this section are naturally expressed in the language of category theory. Awodey [2006] provides an introduction fit for computer scientists.

2.2 Streams

Informally, in a Kahn process network a stream is an infinite sequence that models the successive values transmitted through a queue during the whole execution. The only thing one can do with a stream is to attempt to *destruct* it into its first element, or *head*, and remaining elements, or *tail*. This operation may fail, because the stream might be empty. Thus, a stream is either empty, or can be decomposed into a value and another stream, which might itself be empty, and so on. This suggests the following definition.

Definition 1 (The Domain of Streams)**.** *Given a domain D, the domain Stream(D) of streams of D elements is the solution of the recursive domain equation*

$$\mathit{Stream}(D) \cong D \otimes \mathit{Stream}(D)_\perp$$

The solution of this equation comes equipped with an isomorphism i_s that defines stream construction (via i_s^{-1}) and destruction (via i_s). To make notation lighter, we write abbreviate stream construction by $x.s = i_s(p_\otimes(x,s))$ where $p_\otimes$ is the projection part of the embedding-projection pair $A \otimes B \lhd A \times B$. As expected since adding an element to a queue is strict, we have $\perp.s = \perp$ for any $s \in \mathit{Stream}(D)$. Also, for any continuous function $f : A \Rightarrow_c B$, we define its unique lift to streams $\mathit{Stream}(f) : \mathit{Stream}(A) \Rightarrow_c \mathit{Stream}(B)$ in the standard way; this corresponding to the `map` combinator of functional languages.

In this thesis, we will always use the above definition with D a flat domain, that is a domain of the form $X_\perp$ with X a discretely ordered set. In this case, $\mathit{Stream}(D)$ is isomorphic to $X^* \cup X^\omega$, the set of finite or infinite sequences of elements in X, ordered by prefix. For example, Figure 2.1 depicts the smallest elements of the domain of boolean streams and the ordering between them. Remember however that finite words do not model terminating computations but rather infinite computations that "get stuck" at a certain point. In particular, consider the following function from streams of natural numbers to the flat domain .

$$\mathit{inf}\, x = \begin{cases} 1 & \text{if } x \text{ is infinite} \\ \perp & \text{otherwise} \end{cases}$$

This function is not continuous: given an infinite stream x, it returns $\perp$ for all the finite prefixes of x yet returns 1 for their limit. In particular, it could not be programmed as a recursive function using only stream construction or destruction.

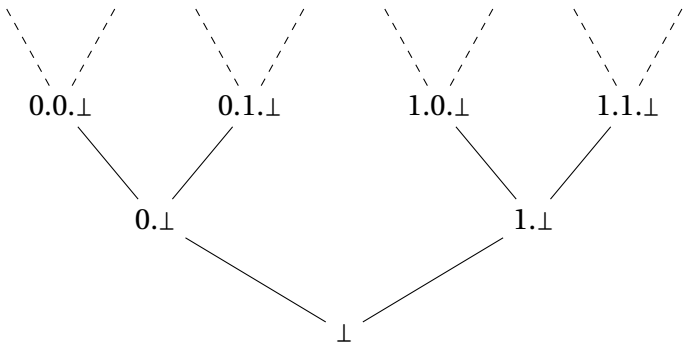


Figure 2.1: The domain $\mathit{Stream}(\mathbb{B}_\perp)$ of boolean streams

The fact that the function *inf* is not continuous simply reflects that it cannot be implemented as a program: it is impossible to decide whether a stream is infinite or not, since this is equivalent to the halting problem. Yet, in our later investigations of AcidS and its type system (Chapter 3), we will want to show that well-typed programs compute infinite streams. In other words, well-typed programs do not get stuck. Thus, we need to reason on this notion mathematically. Let us define a predicate characterizing streams holding at least n elements. We say that such streams *converge* up to n. Then, a stream that converges up to n for any n is clearly infinite.

Definition 2 (Convergence and Totality)**.** *The* convergence *predicate $s \Downarrow_n$ is inductively defined by the following rules*

$$\frac{}{s \Downarrow_0} \qquad\qquad \frac{s \Downarrow_n}{x.s \Downarrow_{n+1}}$$

The total convergence *predicate $s \Downarrow_\infty$ is defined by*

$$s \Downarrow_\infty \stackrel{\text{def}}{=} \forall n \in \mathbb{N}, s \Downarrow_n$$

We say that a stream s such that one has $s \Downarrow_\infty$ totally converges*, or simply is* total*. A non-total stream is said to be* partial

Convergence up to n is of course related to the ordering on the domain of streams. In particular, it is compatible with it in the following sense.

Property 1. *Given streams xs and ys such that $xs \sqsubseteq ys$ and $xs \Downarrow_n$, one has $ys \Downarrow_n$.*

Total convergence, on the other hand, is related to maximality.

Property 2. *Given a flat domain D and stream $xs \in \mathit{Stream}(D)$, xs is maximal just if $xs \Downarrow_\infty$.*

Remark 1. For any domain D, maximal elements of $\mathit{Stream}(D)$ are total. The latter situation does not arise in this thesis.

For any i, one may see convergence up to i as an approximation to totality. In particular, this has the convenient corollary that one can use ordinary induction on n to prove that an element is total. Another notion where this decomposition makes sense is equality. Let us define the following relation.

Definition 3 (Prefix Equality)**.** *The judgment $s =_n s'$ expresses that two streams xs and ys are equal on their first n elements, and is inductively defined by the following rules*

$$\frac{}{xs =_0 ys} \qquad \frac{xs =_n ys}{x.xs =_{n+1} x.ys}$$

In particular, when $xs =_n ys$ then both xs and ys converge up to n.

As expected, equality of prefixes is an approximation of equality, in the following sense.

Property 3 (Equality of Prefixes)**.** *Two streams are equal if all their prefixes are. Formally,*

$$\forall n \in \mathbb{N},\ xs =_n ys \Rightarrow xs = ys$$

We will sometimes use the convergence or prefix judgments up to n in an informal way, where the "integer" n is a member of $\mathbb{N}_\perp$ rather than a plain natural number. This supposes that we have a property ensuring $n \neq \perp$; we will make such conditions explicit when needed.

2.3 Segments

The domain of streams defined above is one of the simplest examples of an interesting non-flat domain; in particular, it contains infinite chains. This is to be expected since streams model processes which may become non-productive after some time. The ordering relation on streams reflects this: by destructing a stream repeatedly, one obtains its elements one by one; at any point along the way, this deconstruction process may get stuck, a situation denoted by $\perp$. Conversely, flat domains correspond to computations which either fail to terminate, providing no information whatsoever, or return complete, total results. In synchronous functional languages in general and this thesis in particular, we are interested in intermediate points between these two extreme cases. More precisely, we would like to describe streams such that when the destruction process succeeds, it returns a whole pack of values (possibly empty) in one step. Understanding how we can *coarsen* a stream function will be the subject of the later sections of this chapter, but we first have to describe such streams where destruction is coarser.

The simplest idea to introduce streams whose destruction may return several elements while staying inside domain theory is to consider streams of strict lists. We sometimes call such lists *segments* to emphasize their role as containers for contiguous elements of a stream.

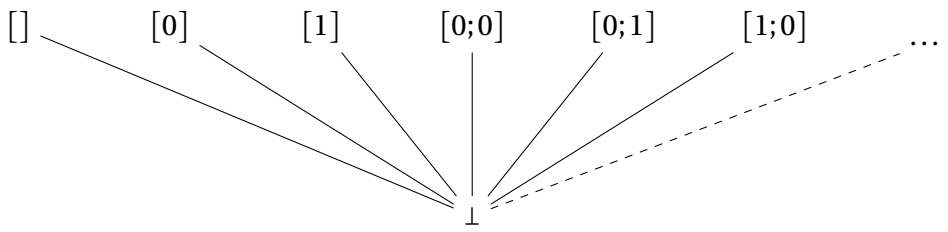


Figure 2.2: The domain $List(\mathbb{B}_\perp)$ of boolean lists

Definition 4 (The Domain of Lists)**.** *Given a domain D, the domain $List(D)$ of finite lists whose elements belong to D is the solution of the recursive domain equation*

$$List(D) \cong \mathbb{1}_\perp \oplus (D \otimes List(D))$$

The strictness of segments corresponds to the fact that list constructors force the evaluation of their elements. In particular, if D is a flat domain then so is $List(D)$, with $|List(D)|$ in bijection with $|D|^* \cup \{\perp\}$. Figure 2.2 shows part of the domain of boolean lists.

We now define syntactic shortcuts for list construction with a syntax à la ML. If i_l is the isomorphism solving the above recursive domain equation, we can write $[\,]$ for the empty list defined as $[\,] = i_l^{-1}(p_\oplus(\iota_1 *))$, with $p_\oplus$ the projection part of the embedding-projection pair $A \oplus B \lhd A + B$. We also write $x;l$ for the list with head x and tail l, defined as $x;l = i_l^{-1}(p_\oplus(\iota_2(p_\otimes(x,l))))$, and $[x_1;\ldots;x_n]$ for $x_1;\ldots;x_n;[\,]$, and index streams by natural numbers so that $xs[i]$ is the i-th value if it exists and $\perp$ otherwise. Finally, for any continuous function $f : A \Rightarrow_c B$ we write $List(f) : List(A) \Rightarrow_c List(B)$ for its unique lift to lists. This function is traditionally written `map` in functional languages.

2.4 An Informal Metalanguage

The previous sections use "raw" set-theoretic functions to build convenient functions for stream and list constructions. We feel that this is a way of explaining how things work in terms of pure domain theory. However, the direct use of isomorphism and embedding-projection pairs quickly becomes tedious and hard to read when one has to write bigger functions. Following established usage—such as in the book of Winskel [1993]—we will use a high-level, informal functional metalanguage for writing continuous functions between domains. Technically, this is justified by the fact that the category of domains is cartesian-closed and thus provides all the necessary combinators for interpreting a λ-calculus.

As an example, let us define a function that computes the size of a segment. We use a Haskell-like syntax, and rely on pattern-matching.

$$\begin{array}{lcl} \textbf{length} & : & List(D) \Rightarrow_c \mathbb{N}_\perp \\ \textbf{length}\,[\,] & = & 0 \\ \textbf{length}\,(x;l) & = & 1 + \textbf{length}\; l \end{array}$$

It is possible to translate, or "compile", this function to the raw combinators provided by domain theory in a systematic way. Pattern-matching can be implemented through the combined use of list destruction, itself relying on the isomorphism that solves the domain equation defining lists and on the projections from strict to lazy products or sums, and on the function $[f \mid g]$ that maps $\iota_1 x$ to $f x$ and $\iota_2 y$ to $g y$. List constructors were explained away in the previous sections. Recursive calls as usual correspond to least fixpoints. We overload the notation (x, y) and use it for both ordinary products and smashed ones. Functions acting on $\mathbb{N}$ such as addition are lifted to act on $\mathbb{N}_\perp$, and natural numbers themselves are injected into $\mathbb{N}_\perp$. This process is systematic but will not be detailed in the thesis.

Finally, as a side remark for readers fluent in Haskell, the domains of streams and segments can be described by the following Haskell data type definitions.

```
data Stream a = Cons !a (Stream a)
data List a = Nil
            | Cons !a !(List a)
```

The bang (!) annotation means that the first argument of `Cons` must be a value, i.e. that this constructor is strict in its first argument. These annotations correspond, at least intuitively, to "unlifting" in domain theory. The usual list data type of the Haskell standard library matches neither the domain of streams nor the one of segments but rather the solution of the more complex domain equation

$$HList(D) \cong 1 \oplus (D \times HList(D)_\perp)$$

while our segments model exactly to OCaml or SML lists.

2.5 Segmented Streams and Clocks

In the rest of this chapter, we consider given a flat domain D that will play the role of "scalars", that is basic elements of streams. Given this domain and the definition of segments given in the previous section, it is now possible to define formally streams whose elements are computed in a bursty way, segment per segment, as follows.

$$SStream(D) \stackrel{\text{def}}{=} Stream(List(D))$$

We call the elements of this domain *segmented streams*.

Let us study the relations between streams and segmented streams. There is an obvious way to transform a segmented stream into a stream, which consists in concatenating all the segments. The continuous function **unpack** implements this transformation.

$$\begin{array}{lcl} \textbf{unpack} & : & SStream(D) \Rightarrow_c Stream(D) \\ \textbf{unpack}\,([].xs) & = & \textbf{unpack}\,xs \\ \textbf{unpack}\,((x;l).xs) & = & x.(\textbf{unpack}\,(l.xs)) \end{array}$$

To create a segmented stream out of a raw stream, we need to know the size each segment should have. This information can be presented as a stream of integers, with the n-th integer describing the length of the n-th segment. We call *clocks* such streams of integers. Since clocks play a prominent role in this thesis, we give their domain an explicit name.

$$\mathbf{Ck} \stackrel{\text{def}}{=} \mathit{Stream}(\mathbb{N}_\perp)$$

Given a segmented stream, we can compute its clock as the stream of lengths of its segments.

$$\begin{array}{lcl} \mathbf{clock} & : & \mathit{SStream}(D) \Rightarrow_c \mathbf{Ck} \\ \mathbf{clock}\ xs & = & \mathit{Stream}(\mathbf{length}) \end{array}$$

Now, given a clock w and stream xs, the continuous function $\mathbf{pack}_w\ xs$ computes a segmented stream whose clock is given by w. It relies on an intermediate function *splitAt* that takes an integer n and a stream and splits the stream into a pair of a list of length n and the remainder of the stream by the iterated application of stream destruction.

$$\begin{array}{lcl} \mathbf{pack} & : & \mathbf{Ck} \Rightarrow_c \mathit{Stream}(D) \Rightarrow_c \mathit{SStream}(D) \\ \mathbf{pack}_{n.w}\ xs & = & (l.\mathbf{pack}_w\ xs') \\ & & \text{where } (l, xs') = \mathit{splitAt}\ n\ xs \\ & & \mathit{splitAt}\ 0\ xs = ([\,], xs) \\ & & \mathit{splitAt}\ (1+n)\ (x.xs) = (x; l, xs') \text{ where } (l, xs') = \mathit{splitAt}\ n\ xs \end{array}$$

Finally, one may compose the two functions to change the clock of a segmented stream to a new clock w using $\mathbf{repack}_w$.

$$\begin{array}{lcl} \mathbf{repack} & : & \mathbf{Ck} \Rightarrow_c \mathit{SStream}(D) \Rightarrow_c \mathit{SStream}(D) \\ \mathbf{repack}_w & = & \mathbf{pack}_w \circ \mathbf{unpack} \end{array}$$

The functions $\mathbf{pack}_w$, **unpack** and $\mathbf{repack}_w$ return partial streams in certain conditions. This is for example the case of **unpack** xs applied to a total segmented stream finishing with an infinite amount of empty segments. The case of $\mathbf{pack}_w\ xs$ is more complex, since the convergence of the output stream depends on w. In the three cases, we would like to describe the relation between the convergence of inputs and the convergence of outputs of the functions. For this we define the *cumulative function* $\mathcal{O}_w$ of a clock w,

$$\begin{array}{lcl} \mathcal{O} & : & \mathbf{Ck} \Rightarrow_c \mathbb{N} \Rightarrow_c \mathbb{N}_\perp \\ \mathcal{O}_w(0) & = & 0 \\ \mathcal{O}_{n.w}(1+i) & = & n + \mathcal{O}_w(i) \end{array}$$

This function may return $\perp$ when applied to a partial clock. When we use $\mathcal{O}_w(i)$ as a natural number, we implicitly suppose that w converges up to i.

Now, given a segmented stream xs that converges up to i, the total number of elements present in all segments up to i is given by $\mathcal{O}_{\mathbf{clock}\ xs}(i)$. We can use this fact to characterize the convergence of the functions **pack** and **unpack**, using the cumulative function of the clocks of their inputs or outputs.

Property 4 (Convergence of **pack**)**.** *The output of* $\mathbf{pack}_w$ *converges up to* i *if its input converges up to* $\mathcal{O}_w(i)$*.*

$$\forall w \in \mathbf{Ck}, \forall xs \in \mathit{Stream}(D), \forall i \in \mathbb{N}, \mathit{if}\ w \Downarrow_i\ \mathit{and}\ xs \Downarrow_{\mathcal{O}_w(i)}\ \mathit{then}\ \mathbf{pack}_w\ xs \Downarrow_i$$

Property 5 (Convergence of **unpack**)**.** *Let* w *be a clock. The output of* **unpack** *, applied to an input whose clock is equal to* w *up to* i *steps, converges up to* $\mathcal{O}_w(i)$*.*

$$\forall w \in \mathbf{Ck}, \forall xs \in \mathit{Stream}(D), \forall i \in \mathbb{N}, \mathit{if}\ \mathbf{clock}\ xs =_i\ w\ \mathit{then}\ \mathbf{unpack}\ xs \Downarrow_{\mathcal{O}_w(i)}$$

Let us now discuss what we obtain by composing **pack** and **unpack**. In the case of the composite $\mathbf{repack}_w$, the situation is clear—this function does nothing to an argument whose clock is already equal to w.

Property 6 (Behavior of $\mathbf{repack}_w$)**.** *The function* $\mathbf{repack}_w$ *acts as the identity function when its input has the same clock* w*. In other words, for any* i *one has*

$$\mathbf{repack}_w\ x =_i x \quad \mathit{provided}\ \mathbf{clock}\ x =_i\ w$$

For $\mathbf{unpack} \circ \mathbf{pack}_w$, the situation is slightly more involved since w might either diverge or be total but end with an infinite amount of zeroes. In both cases, the composite of the two functions is smaller (for the pointwise order) than the identity function. We could stop there, but it is easy to give finer characterization using the cumulative function of w.

Property 7 (Behavior of $\mathbf{unpack} \circ \mathbf{pack}$)**.** *Given clock* w *and stream xs, for any* $i \geq 0$ *one has*

$$(\mathbf{unpack} \circ \mathbf{pack}_w)\ xs =_{\mathcal{O}_w(i)}\ xs\ \mathit{provided}\ w \Downarrow_i$$

Finally, algorithmic manipulation on clocks will often make use of the *index function* $\mathcal{I}_w$, the quasi-inverse of the cumulative sum. We write $\mathcal{I}_w(j)$ for this function, which returns the smallest i such that $\mathcal{O}_w(i) \geq j$, if it exists.

Definition 5 (Index Function)**.**

$$\begin{array}{rcl} \mathcal{I}_{\_}(\_) & : & \mathbf{Ck} \Rightarrow_c \mathbb{N}_\perp \Rightarrow_c \mathbb{N}_\perp \\ \mathcal{I}_w(0) & = & 0 \\ \mathcal{I}_{n.w}(j) & = & 1 + \mathcal{I}_w(j-n) \end{array}$$

Note that $\mathcal{I}_w(j)$ may be equal to $\perp$ when the clock ends with an infinite amount of zeroes and j is too large. We will sometimes use $\mathcal{I}_w$ as a partial function from natural numbers to natural numbers, rather than a continuous function on domains.

Remark 2. The cumulative sum and index functions can be traced back to the *Thèse d'État* of Halbwachs [1984] and to his joint work with Caspi [Caspi and Halbwachs, 1986]. In these works, the cumulative sum $\mathcal{O}_w(\_)$ is called the *event counter* and the index function $\mathcal{I}_w(\_)$ the *time function.* The authors remark that, for a fixed clock w, these functions are Galois adjoint and thus quasi-inverses of each other. This is the case here as well; in particular, we have $\mathcal{O}_w(\mathcal{I}_w(j)) \leq j$. Let us remark that when w is binary, as in the thesis of Plateau [2010], the above inequality tightens to $\mathcal{O}_w(\mathcal{I}_w(j)) = j$.

2.6 Buffering and Clock Adaptability

The previous section did not directly discuss the convergence of **repack**. One may see this function as a denotational analogue to a buffering process, as the following example shows. The notation $0^2(1)^\omega$ is a shortcut for the clock beginning with two zeroes followed by an infinite amount of ones.

$$\begin{array}{lcl} xs & = & [0].[2].[4].[6].[8].[10].[12]\ldots \\ ys = \mathbf{repack}_{0^2(1)^\omega}\ xs & = & [].[].[0].[2].[4].[6].[8].[10]\ldots \end{array}$$

Imagine that there exists some global, discrete time scale. The stream *ys* carries the same values as *xs*, but they arrive two time steps later. This is what the following chronogram shows, with vertical alignment representing simultaneity.

xs	$[0]$	$[2]$	$[4]$	$[6]$	$[8]$	$[10]$	$[12]$	…
$ys = \mathbf{repack}_{0^2(1)^\omega}\ xs$	$[]$	$[]$	$[0]$	$[2]$	$[4]$	$[6]$	$[8]$	…
Number of buffered items	1	2	2	2	2	2	2	…

The lowest row of the table shows the amount of items from *xs* that should be stored inside the buffer.

This buffering process is well-behaved because at each time step i, the prefix of length i of *xs* has provided enough data to create the prefix of length i of *ys*. Here, the situation gives a bit more freedom: remark that at time step i, we have received enough elements from *xs* to produce a prefix of *ys* of length $i+2$. This gives rise to an order relation on clocks called *precedence*, as in the thesis of Plateau [2010].

Definition 6 (Precedence)**.** *Given clocks w and w' and $k \geq 0$, we say that w k-*precedes *w', denoted by $w \preceq_k w'$, when*

$$\forall i \in \mathbb{N},\ \mathcal{O}_w(i) \geq \mathcal{O}_{w'}(i+k)$$

Moreover, we say that w precedes w' and write $w \preceq w'$ when $w \preceq_0 w'$.

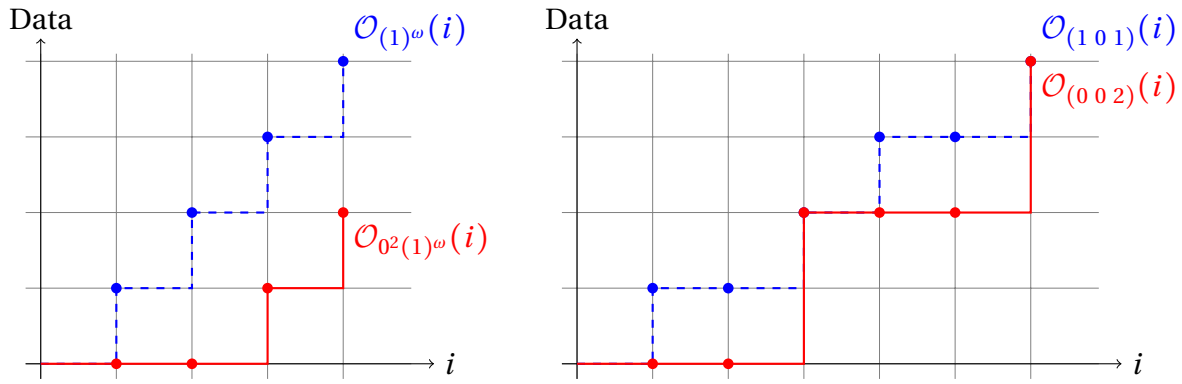
With this notion, one may characterize the convergence of repack when the clock of its input k-precedes the clock of its output.

Property 8 (Convergence of **repack**, precedence case)**.** *Let w and w' be clocks and $k \geq 0$ such that $w \preceq_k w'$. Then the output of* $\mathbf{repack}_{w'}$*, applied to an input whose clock is equal to w up to i steps, converges up to $i+k$.*

$$\forall w, w' \in \mathbf{Ck}, \forall xs \in \mathit{Stream}(D), \forall k, i \in \mathbb{N}, \textit{if } w \preceq_k w' \textit{ and } \mathbf{clock}\ xs =_i w \textit{ then } \mathbf{repack}_{w'}\ xs \Downarrow_{i+k}$$

Proof. One has the following chain of inferences.

$$\begin{array}{rll} xs & \Downarrow_i & \\ \mathbf{unpack}\ xs & \Downarrow_{\mathcal{O}_w(i)} & \text{(Property 5, hypothesis } \mathbf{clock}\ xs =_i w) \\ \mathbf{unpack}\ xs & \Downarrow_{\mathcal{O}_{w'}(i+k)} & (\mathcal{O}_w(i) \geq \mathcal{O}_{w'}(i+k),\ \Downarrow \text{ prefix-closed}) \\ \mathbf{pack}_{w'}\ (\mathbf{unpack}\ xs) & \Downarrow_{i+k} & \text{(Property 4)} \\ \mathbf{repack}_{w'}\ xs & \Downarrow_{i+k} & \text{(Definition of } \mathbf{repack}) \end{array}$$ □



(a) $(1)^\omega <:_2 0^2(1)^\omega$ (b) $(1\,0\,1)^\omega <:_0 (0\,0\,2)^\omega$

The precedence relation only characterizes *causality*: it ensures that the current outputs of the buffer only depend on the inputs received up to now. It does not force the buffer to be of finite size. For that, we introduce a relation expressing that the amount of data produced and consumed in a buffer are asymptotically equivalent, and thus that their difference stays bounded.

Definition 7 (Rate and Synchronizability)**.** *We call* rate *of a clock the limit of its cumulative function when times grows toward infinity.*

$$rate(w) = \lim_{i\to\infty} \frac{\mathcal{O}_w(i)}{i}$$

The limit might not exist, in which case the rate of the clock is not defined. Two clocks are synchronizable, denoted by $w \bowtie w'$, when $rate(w) = rate(w')$.

Definition 8 (Adapatability)**.** *Given clocks w and w' and $k \geq 0$, we say that w is k-*adaptable *to w' if they are synchronizable and that w k-precedes w'.*

$$w <:_k w' \stackrel{\text{def}}{=} w \bowtie w' \text{ and } w \preceq_k w'$$

Moreover, we say that w is adaptable *to w' when it is 0-adaptable to it.*

Figure 2.6 gives a graphical view of adaptability. The left side illustrates the example introducing this section $(1)^\omega <:_2 0^2(1)^\omega$; precedence corresponds to the fact that the dashed curve stays above the plain one, and synchronizability the fact that the difference between the two at each i stays bounded (by 2, in this case). It is furthermore 2-adaptable since the dashed curve shifted twice to the right would still be above the plain curve. The right side shows an example with integer clocks, $(1\,0\,1)^\omega <:_0 (0\,0\,2)^\omega$. This relation can be realized by a buffer of size one. The clock $(1)^\omega$ is at most 0-adaptable but not 1-adaptable since shifting the dashed curve to the right even once makes it lie below the plain one at $i = 2$.

2.7 Rescaling and Clock Composition

The previous sections defined clocks as streams of integers that describe a segmented stream. Through the use of the **repack** function, one may pass from one segmented stream to another; Property 8 gives a simple result on the convergence of the output in terms of both the input clock and desired output clock. This property, however, does not characterize all the changes of clocks we are interested in, because of the condition $\mathcal{O}_w(i) \geq \mathcal{O}_{w'}(i+k)$.

Example 1 Consider the following streams, where we write $(1)^\omega$ for the constant clock of infinitely many ones.

$$\begin{array}{lcl} xs & = & [0;1].[].[2;3].[].[4;5].[].[6;7].[]\ldots \\ ys = \mathbf{repack}_{(1)^\omega}\ xs & = & [0].[1].[2].[3].[4].[5].[6].[7]\ldots \end{array}$$

There are two ways to relate *xs* and *ys*.

The first one was explained in the previous section: imagine that there exists some global, discrete time scale, that one segment of *xs* is computed per time step, and that they have a length of two at even time steps and zero at odd time steps. In other words, the stream *xs* has clock $(2\ 0)^\omega$. Then, the above use of **repack** corresponds to some *buffering* process that stores two items at every even time step and releases one item at each time step. This use case is captured by Property 8: at any step i we have $\mathcal{O}_{(2\ 0)^\omega}(i) \geq \mathcal{O}_{(1)^\omega}(i)$ and $\mathbf{clock}\ xs =_i (2\ 0)^\omega$, thus $\mathbf{repack}_{(1)^\omega}\ xs$ converges up to i. This interpretation could be represented as follows, with vertical alignment denoting simultaneity.

xs	$[0;1]$	$[]$	$[2;3]$	$[]$	$[4;5]$	$[]$	$[6;7]$	$[]$	$\ldots$
$ys = \mathbf{repack}_{(1)^\omega}\ xs$	$[0]$	$[1]$	$[2]$	$[3]$	$[4]$	$[5]$	$[6]$	$[7]$	$\ldots$

Taking an operational view of process networks, we may think of the above situation as a case of simple *synchronization* between the producer of *xs* and the consumer of *ys*, with both running in lockstep. Yet, we could imagine a different scenario where several activations of the consumer of *ys* occur for one of *xs*, or vice versa. For example, *ys* could be consumed twice at even time steps, which means that its consumer would run twice faster than the producer of *xs*. At odd time steps, there is no value left, so the consumer of *ys* does not run at all. This is what the next chronogram depicts.

xs	$[0;1]$	$[]$	$[2;3]$	$[]$	$[4;5]$	$[]$	$[6;7]$	$[]$	$\ldots$
$ys = \mathbf{repack}_{(1)^\omega}\ xs$	$[0]$ $[1]$		$[2]$ $[3]$		$[4]$ $[5]$		$[6]$ $[7]$		$\ldots$

One may think of this example as a situation where time passes faster for the consumer of *ys* than for the producer of *xs*. We say that these two processes are in different time scales, or that a *change of scale*, or *rescaling*, occurs. This usage is not covered at all by Property 8: what we would like to say is that, given $xs \Downarrow_{2i}$, one obtains $\mathbf{repack}_{(1)^\omega}\ xs \Downarrow_{2i+1}$.

Example 2 Rather than trying to deduce a more general result now, we first continue our investigations. The situation above is quite peculiar because one can understand it either as buffering or as rescaling. This is not always the case. Consider the following two streams.

$$\begin{array}{lcl} xs & = & [0;1;1].[2;3;5].[8;13;21].[34;55;89]\ldots \\ ys = \mathbf{repack}_{(1\,2)^\omega}\ xs & = & [0].[1;1].[2].[3;5].[8].[13;21].[34].[55;89]\ldots \end{array}$$

Clearly, this clock transformation cannot be implemented via a finite-state buffer. Remembering the previous section, we observe that xs and ys are not synchronizable since they have different rates: $rate((3)^\omega) = 3$ but $rate((1\,2)^\omega) = \frac{3}{2}$. The following chronogram shows the unbounded growth of the amount of items stored in the buffer.

xs	$[0;1;1]$	$[2;3;5]$	$[8;13;21]$	$[34;55;89]$	$\ldots$
$ys = \mathbf{repack}_{(1\,2)^\omega}\ xs$	$[0]$	$[1;1]$	$[2]$	$[3;5]$	$\ldots$
Number of buffered items	2	3	5	6	$\ldots$

On the other hand, this describes a valid rescaling where the consumer of ys runs exactly twice faster than the producer of xs, and whose second activation consumes a segment of length two.

xs	$[0;1;1]$	$[2;3;5]$	$[8;13;21]$	$[34;55;89]$	$\ldots$
$ys = \mathbf{repack}_{(1\,2)^\omega}\ xs$	$[0]\,[1;1]$	$[2]\,[3;5]$	$[8]\,[13;21]$	$[34]\,[55;89]$	$\ldots$

Here, we would like a result on the convergence of **repack** expressing that for any $i \geq 0$, $xs \Downarrow_i$ implies $\mathbf{repack}_{(1\,2)^\omega}\ xs \Downarrow_{2i}$.

Example 3 We have, up to now, considered situations where the consumer of ys runs faster than the producer of xs, but the reverse situation also makes sense. Let us swap the roles of xs and ys.

$$\begin{array}{lcl} xs & = & [0].[1;1].[2].[3;5].[8].[13;21].[34].[55;89]\ldots \\ ys = \mathbf{repack}_{(3)^\omega}\ xs & = & [0;1;1].[2;3;5].[8;13;21].[34;55;89]\ldots \end{array}$$

Now, this clock transformation could not even be implemented with a buffer of *unbounded* size, since it would need to know the segment received at the second time step to produce its first segment. The chronogram below highlights the first items where this *non-causal* behavior occurs.

xs	$[0]$	$[1;1]$	$[2]$	$[3;5]$	$\ldots$
$ys = \mathbf{repack}_{(3)^\omega}\ xs$	$[0;1;1]$	$[2;3;5]$	$[8;13;21]$	$[34;55;89]$	$\ldots$

Yet, this makes sense as a change of scale where the producer of xs runs twice faster than the consumer of ys, in a completely symmetric manner compared to our previous example.

xs	$[0]\,[1;1]$	$[2]\,[3;5]$	$[8]\,[13;21]$	$[34]\,[55;89]$	$\ldots$
$ys = \mathbf{repack}_{(3)^\omega}\ xs$	$[0;1;1]$	$[2;3;5]$	$[8;13;21]$	$[34;55;89]$	$\ldots$

What are the rules governing changes of scale? They should be characterized through three pieces of information: the clock counting in *internal* time–that of ys in the first two

examples and that of *xs* in the last one–, the clock counting in *external time*–conversely, twice that of *xs* and then that of *ys*–and the *local time clock* describing the relation between internal and external time–respectively $(2\ 0)^\omega$ in example 1 and $(2)^\omega$ in examples 2 and 3. The intuition supported by the above examples is that *the external clock should be equal to the clock obtained by fusing the numbers present in the internal clock, with the amount of numbers to fuse described by the local time clock.* This fusion process is called *clock composition* and is defined mathematically by the following operator.

Definition 9 (Clock Composition)**.** *The composition of two clocks w and w' is denoted w on w' and defined as follows.*

$$\begin{array}{lcl} (n.w)\ on\ w' & = & (sum\ l).(w\ on\ w'')\ where\ (l,w'') = \mathbf{splitAt}\ n\ w' \\ & & sum\ [\,] = 0 \\ & & sum\ (x;l) = x + sum\ l \end{array}$$

In each example, let us call *we* the external clock, *w* the local time clock and *wi* the internal clock. The table below shows that in each example, we have precisely *we* = *w on wi*.

Example	External clock *we*	Internal Clock *wi*	Local Time Clock *w*
1	**clock** $xs = (2\ 0)^\omega$	$(2\ 0)^\omega$	**clock** $ys = (1)^\omega$
2	**clock** $xs = (3)^\omega$	$(2)^\omega$	**clock** $ys = (1\ 2)^\omega$
3	**clock** $ys = (3)^\omega$	$(2)^\omega$	**clock** $xs = (1\ 2)^\omega$

Clock composition has good distributivity properties with regard to cumulative functions. In fact, it corresponds exactly to their reverse composition.

Property 9 (Cumulative Function of Compound Clocks)**.** *Given clocks w and w', one has*

$$\mathcal{O}_{w\ on\ w'} = \mathcal{O}_{w'} \circ \mathcal{O}_w$$

This result leads to an immediate characterization of the action of $\mathbf{repack}_w\ xs$ in the case where either w or **clock** xs is a compound clock.

Property 10 (Convergence of **repack**, rescaling case)**.** *Let w and wi be clocks and write we for w on wi. We consider two applications of the function* **repack**.

First, the function $\mathbf{repack}_{wi}$ *maps a stream whose clock is equal to we up to i steps to a stream converging up to $\mathcal{O}_w(i)$.*

$$\forall xs \in Stream(D), \forall k, i \in \mathbb{N}, \textit{if } we =_i w\ on\ wi \textit{ and } \mathbf{clock}\ xs =_i we \textit{ then } \mathbf{repack}_{wi}\ xs \Downarrow_{\mathcal{O}_w(i)}$$

Second, the function $\mathbf{repack}_{we}$ *maps a stream whose clock is equal to wi up to $\mathcal{O}_w(i)$ steps to a stream converging up to i.*

$$\forall xs \in Stream(D), \forall k, i \in \mathbb{N}, \textit{if } we =_i w\ on\ wi \textit{ and } \mathbf{clock}\ xs =_{\mathcal{O}_w(i)} we \textit{ then } \mathbf{repack}_{wi}\ xs \Downarrow_i$$

Proof. The two properties are very similar. For the first case, we have

$$\begin{aligned}
xs &\Downarrow_i \\
\mathbf{unpack}\, xs &\Downarrow_{\mathcal{O}_{we}(i)} && \text{(Property 5, } \mathbf{clock}\, xs =_i we\text{)} \\
\mathbf{unpack}\, xs &\Downarrow_{\mathcal{O}_{w\ on\ wi}(i)} && \text{(Property 9)} \\
\mathbf{unpack}\, xs &\Downarrow_{\mathcal{O}_{wi}(\mathcal{O}_w(i))} && \text{(Hypothesis } we = w\ on\ wi\text{)} \\
\mathbf{pack}_{wi}\,(\mathbf{unpack}\, xs) &\Downarrow_{\mathcal{O}_w(i)} && \text{(Property 4)} \\
\mathbf{repack}_{wi}\, xs &\Downarrow_{\mathcal{O}_w(i)} && \text{(Definition of } \mathbf{repack}\text{)}
\end{aligned}$$

and for the second case

$$\begin{aligned}
xs &\Downarrow_{\mathcal{O}_w(i)} \\
\mathbf{unpack}\, xs &\Downarrow_{\mathcal{O}_{wi}(\mathcal{O}_w(i))} && \text{(Property 5, } \mathbf{clock}\, xs =_{\mathcal{O}_w(i)} wi\text{)} \\
\mathbf{unpack}\, xs &\Downarrow_{\mathcal{O}_{w\ on\ wi}(i)} && \text{(Property 9)} \\
\mathbf{unpack}\, xs &\Downarrow_{\mathcal{O}_{we}(i)} && \text{(Hypothesis } we = w\ on\ wi\text{)} \\
\mathbf{pack}_{we}\,(\mathbf{unpack}\, xs) &\Downarrow_i && \text{(Property 4)} \\
\mathbf{repack}_{wi}\, xs &\Downarrow_i && \text{(Definition of } \mathbf{repack}\text{)}
\end{aligned}$$

which concludes the proof. □

2.8 Properties of Clocks

In order to improve our understanding of clock composition, and the range of changes of scale that we may perform, let us study its algebraic properties. In what follows, we say that a clock is *binary* if it holds no natural number strictly larger than one, and that it is *strictly positive* if it holds no zero.

Property 11 (The Clock Monoid)**.** *The domain of clocks* **Ck** *equipped with the clock composition operator "on" and the clock* $(1)^\omega$ *as neutral element forms a monoid. Additionally, the clock* $(0)^\omega$ *is an absorbing element.*

However, this monoid is not an inverse monoid, and thus not a group. Given clocks w and w', w' is a left-inverse for w when $w'\ on\ w = (1)^\omega$ and a right-inverse when $w\ on\ w' = (1)^\omega$. Some clocks, such as $(2)^\omega$, have no left-inverse, and several right-inverses; for example, $(1\ 0)^\omega$ and $(0\ 1)^\omega$. Reciprocally, a clock such as $(1\ 0\ 1)^\omega$ has several left-inverses, for example $(2\ 1)^\omega$ and $(1\ 2)^\omega$, but no right-inverse. This is in fact a general result, stated below.

Property 12 (Inverses of Clocks)**.** *A clock w has left-inverses if and only if it is binary and of non-null rate, and has right-inverses if and only if it is strictly positive. Moreover, the left-inverses of binary clocks of non-null rate are strictly positive, and the right-inverses of strictly-positive clocks are binary.*

Proof. Consider two clocks w and w' such that $w\ on\ w' = (1)^\omega$. By definition of clock composition, any zero appearing in w must appear in $w\ on\ w'$, and thus here w is strictly positive. Conversely, since numbers appearing in $w\ on\ w' \leq (1)^\omega$ are sums of numbers appearing in w', w' may not contain integer larger than one. □

The monoid is not cancellative either: it has elements which are either not left cancellative or not right cancellative. A clock w is left cancellative if for any w' and w'', one has $w \text{ on } w' = w \text{ on } w''$ implies $w' = w''$. This is clearly not the case since, as explained above, $(2)^\omega$ has two right inverses. Binary clocks provide similar counter-examples for right cancellativity.

Property 13 (Factoring of Clocks)**.** *Any clock w factors as w_b on w_p, with w_b a binary clock and w_p a strictly positive one. Moreover, if w has non-null rate, this factorization is unique.*

Proof. The clock w_b is the minimum of w and $(1)^\omega$, and w_p is w with zeroes removed. In case w_b has null rate, it ends with an infinite sequence of zeroes, and in this case w_b may end with a sequence of arbitrary strictly positive integers. □

This situation with regard to inverses leads us to adopt the following relational notation for changes of scales.

Definition 10 (Clock Scattering and Gathering)**.** *We say that w_1* scatters into *w_2 up to w, which we write $w_1 \downarrow_w w_2$, when $w_1 = w \text{ on } w_2$. Conversely, w_1* gathers into *w_2 up to w, which we write $w_2 \uparrow_w w_1$, when $w \text{ on } w_2 = w_1$.*

At this point, the reader may find strange that we adopt two distinct notations for what appears to be the same relation, reversed. Our main motivation is that, when one considers only the first two words of each triple, scatter and gather are very different: there are several w_2 such that $w_1 \downarrow_w w_2$ while there is only one w_1 such that $w_2 \uparrow_w w_1$—because clock composition is a function. Let us make this remark precise by defining the two following functions on sets of clocks.

$$\begin{array}{lcl} \downarrow_w & : & \mathcal{P}(\mathbf{Ck}) \to \mathcal{P}(\mathbf{Ck}) \\ \downarrow_w X & \stackrel{\text{def}}{=} & \{y \mid x \downarrow_w y, x \in X\} \end{array}$$

$$\begin{array}{lcl} \uparrow_w & : & \mathcal{P}(\mathbf{Ck}) \to \mathcal{P}(\mathbf{Ck}) \\ \uparrow_w X & \stackrel{\text{def}}{=} & \{y \mid x \uparrow_w y, x \in X\} \end{array}$$

One can then show that these two functions bear a special relationship.

Property 14 (The Galois Surjection between Clock Gathering and Scattering)**.** *The function pair $(\uparrow_w, \downarrow_w)$ forms a Galois connection on $\mathcal{P}(\mathbf{Ck})$ ordered by inclusion. In other words, for any $X, Y \subseteq \mathbf{Ck}$, we have*

$$\uparrow_w X \subseteq Y \Leftrightarrow X \subseteq \downarrow_w Y$$

Since it is a Galois connection, we have $X \subseteq \downarrow_w \uparrow_w X$ for any X. Moreover, $\uparrow_w \downarrow_w X = X$ which means that it is a Galois surjection.

Proof. Let us first prove that $X \subseteq \downarrow_w \uparrow_w X$ and $\uparrow_w \downarrow_w X = X$ for any X. Showing that for any X one has $X \subseteq \downarrow_w \uparrow_w X$ can be done simply by unfolding the definition: since, $\downarrow_w \uparrow_w X = \{x' \mid w \text{ on } x = w \text{ on } x'\}$, clearly $x \in \downarrow_w \uparrow_w X$. Observe that in general $X \not\subseteq \downarrow_w \uparrow_w X$ since for example $(1\ 0)^\omega \in \downarrow_{(2)^\omega} \uparrow_{(2)^\omega} \{(0\ 1)^\omega\}$. Finally, $\uparrow_w \downarrow_w X = \{w \text{ on } y \mid x \in X, x = w \text{ on } y\} = X$.

These two results imply, and in fact are equivalent to, the existence of a Galois connection: $\uparrow_w X \subseteq Y$ implies $\downarrow_w \uparrow_w X \subseteq \downarrow_w Y$ by monotonicity of $\downarrow_w$ and thus $X \subseteq \downarrow_w Y$ by transitivity. The reverse direction is similar, $X \subseteq \downarrow_w Y$ implies $\uparrow_w X \subseteq \uparrow_w \downarrow_w Y = Y$. □

This result motivates the notation by showing an asymmetry between scattering and gathering. We give examples and intuitions for these relations at the end of Section 2.10. Another justification for the notation is that in the next chapters we will also consider scattering and gathering for objects more complex than streams, such as functions. In this case the two notions will be even more asymmetric.

2.9 Clocked Streams

The domain of segmented streams $SStream(A)$, introduced in Section 2.5, models queues in which one produces and consumes data by batch rather than one-by-one. Each segmented stream can be mapped to its clock, which describes how it grows as time passes. However, we will often be interested in the reverse direction: given a clock w, can we describe a domain $CStream_w(A)$ of streams compatible with w in some sense? It turns out that there is a simple way to define this domain using the following lemma.

Lemma 1 (Strict Idempotents and Sub-Domains)**.** *Let f be a strict and idempotent continuous function from a domain D to itself. Then $Im(f)$ is a sub-domain of D.*

Proof. Recall that $Im(f) = \{y \mid \exists x \in D, f(x) = y\}$. First, since $f(\bot) = \bot$ one has $\bot \in Im(f)$. Then, suppose $Y \subseteq Im(f)$ is a directed set; we want to show that $\bigsqcup Y \in Im(f)$. Observe that because f is continuous, $f(\bigsqcup Y) = \bigsqcup f(Y)$; but $f(Y) = Y$ since f is idempotent. Hence $f(\bigsqcup Y) = \bigsqcup Y$ and thus $\bigsqcup Y \in Im(f)$. □

Now, consider the function $\mathbf{cut}_w : SStream(D) \Rightarrow_c SStream(D)$ defined as follows, with w a clock.

$$\begin{array}{lcl} \mathbf{cut}\_\_ & : & \mathbf{Ck} \Rightarrow_c SStream(D) \Rightarrow_c SStream(D) \\ \mathbf{cut}_{n.w}\ (l.xs) & = & \begin{cases} l.(\mathbf{cut}_w\ xs) & \text{when } \mathbf{length}\ l = n \\ \bot & \text{otherwise} \end{cases} \end{array}$$

This function maps its argument to its longest prefix whose clock is a prefix of w. For a fixed w, $\mathbf{cut}_w$ is a deflation, and hence is strict and idempotent. Its image consists of all streams whose clocks are prefixes of w, and by virtue of Lemma 1, it is a domain. We adopt it as definition of $CStream_w(A)$.

Definition 11 (Clocked Streams)**.** *Given a clock $w \in \mathbf{Ck}$, the domain $CStream_w(D)$ of w-*clocked streams *is defined as*

$$CStream_w(D) = Im(\mathbf{repack}_w)$$

Figure 2.3 shows the full domain $CStream_{2.0.1.\bot}(\mathbb{B}_\bot)$. It corresponds to the first four levels of the domain $CStream_{(2\,0\,1)^\omega}(\mathbb{B}_\bot)$. This example shows that $w \sqsubseteq w'$ implies $CStream_w(D) \triangleleft$

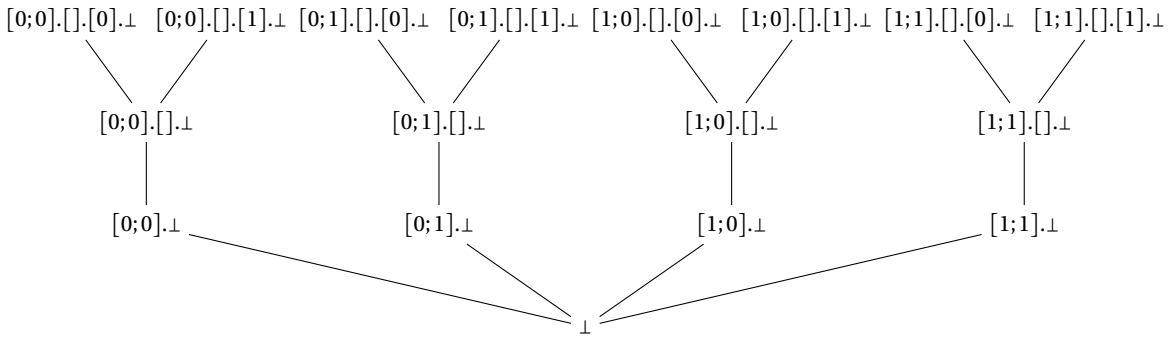


Figure 2.3: The domain $CStream_{2.0.1.\bot}(\mathbb{B}_\bot)$

$CStream_{w'}(D)$. On the other hand, when studying the semantics of functional synchronous languages, we will often be interested in comparing/embedding the domain of w-clocked streams, for a certain w, with/into other stream domains, not necessarily clocked. The two following properties describe this kind of relation between domains.

Property 15 (Clocked Streams and Segmented Streams)**.** *For any fixed clock w, the domain of w-clocked streams is a retract of the domain of segmented streams. The embedding is given by the inclusion map $CStream_w(D) \hookrightarrow SStream(D)$ and the projection by* $\mathbf{cut}_w$.

$$(\hookrightarrow, \mathbf{cut}_w) : CStream_w(D) \triangleleft SStream(D)$$

The next property is more interesting, and will be useful later on when relating the typed and untyped semantics of a language.

Property 16 (Clocked Streams and Streams)**.** *For any fixed clock w, the domain of w-clocked streams is a retract of the domain of streams. The embedding is given by* **unpack** *and the projection by* $\mathbf{pack}_w$.

$$(\mathbf{unpack}, \mathbf{pack}_w) : CStream_w(D) \triangleleft Stream(D)$$

Additionally, if $rate(w) > 0$, then this retraction is in fact an isomorphism.

$$(\mathbf{unpack}, \mathbf{pack}_w) : CStream_w(D) \cong Stream(D) \text{ when } rate(w) > 0$$

2.10 Ultimately Periodic Clocks

As elements of a domain, clocks are ideal objects that model in an abstract way the communications of Kahn networks. As such, most interesting relations on them, including equality and adaptability, are at best semi-decidable. In our later investigations of clock-related type systems, we will sometimes need to restrict ourselves to a more syntactic class of clocks where

relations of interest are decidable. This section describes such a set of clocks, the *ultimately periodic* ones.

Ultimately periodic clocks have been studied in detail in the PhD thesis of Plateau [2010, Chapter 4] in the binary case. She outlined how to extend the binary case to the more general setting of integers greater than one in Chapter 11. We continue this extension here, following her notations and general style.

2.10.1 Definitions

Definition 12 (Ultimately Periodic Words)**.** *An* ultimately periodic word *is a pair of sequences of natural numbers* $(u, v) \in \mathbb{N}^* \times \mathbb{N}^+$*. Note that* v *is non-empty.*

We write $u(v)$ for this pair, or (v) when u is empty. We say that u is the *prefix part* and v the *periodic part* of $u(v)$.

Definition 13 (Ultimately Periodic Clocks)**.** *Each ultimately periodic word* $u(v)$ *gives rise to a clock* $u(v)^\omega$*, described by the following function, considering* u *and* v *as elements of List*$(\mathbb{N})$*.*

$$\begin{array}{lcl} \_(\_)^\omega & : & List(\mathbb{N}_\perp) \times List(\mathbb{N}_\perp) \Rightarrow_c Stream(\mathbb{N}_\perp) \\ [](v)^\omega & = & v(v)^\omega \\ (n;u)(v)^\omega & = & n.u(v)^\omega \end{array}$$

We call ultimately periodic clocks *the clocks obtained in this way. Additionally, clocks of the form* $(v)^\omega$ *are said to be* (strictly) periodic.

We write $|s|$ for the length of a sequence of natural numbers s and $|s|_1 = \sum_{1 \le i \le |s|} s[i]$ for the sum of its elements. Because v is non-empty, a clock $u(v)^\omega$ is always total, and we thus often write $u(v)^\omega[i]$ for its i-th integer. By convention, $u(v)^\omega[1]$ is the first number in $u(v)^\omega$ and we take $u(v)^\omega[0] = 0$. Finally, given a sequence s, we write s^i for s concatenated i times with itself.

Lemma 2. *A clock* w *is ultimately periodic if and only if there exists two numbers* a *and* b *with* $a > 0$ *such that* $w[i + aj + b] = w[i + b]$ *for all numbers* i *and* j *such that* $i > 0$*. We call* a *the* period *and* b *the* phase*.*

Proof. For the left to right direction, assuming $w = u(v)^\omega$, take $a = |v|$ and $b = |u|$. First, show that for any k one has $u(v)^\omega[k + |v| + |u|] = u(v)^\omega[k + |u|]$. Conclude by induction on j that $u(v)^\omega[i + |v|j + |u|] = u(v)^\omega[i + |u|]$.

For the right to left one, form the words $u = w[1] \dots w[b]$ and $v = w[b+1] \dots w[b+a]$ and show $w = u(v)^\omega$. □

2.10.2 Equality of Clocks and Equivalence of Words

We have seen that an ultimately periodic word (u, v) describes a unique clock $u(v)^\omega$. The converse is not true however, for the same ultimately periodic clock arises from an infinite

number of words. For example, one has $(2\,0)^\omega = (2\,0\,2\,0)^\omega = 2(0\,2)^\omega = 2(0\,2\,0\,2)^\omega$ and so on. Words denoting the same clock will be said to be *equivalent*.

Definition 14 (Equivalence of Words)**.** *Two ultimately periodic words $u_1(v_1)$ and $u_2(v_2)$ are said to be* equivalent *whenever $u_1(v_1)^\omega = u_2(v_2)^\omega$. We then write $u_1(v_1) \equiv u_2(v_2)$.*

The examples above exposed two transformations on words that preserve equivalence. They will turn out to be relevant to the algorithmic manipulation of ultimately periodic words.

Definition 15 (Rotation and Expansion)**.** *We define as follows the* rotation *function rot and, given $i > 1$, the* expansion *function exp_i.*

$$rot(u(nv)) = un(vn)$$
$$exp_i(u(v)) = u(v^i)$$

As an example, the earlier equivalences come from $exp_2(2\,0) = (2\,0\,2\,0)$, $rot(2\,0) = 2(0\,2)$ and $exp_2(rot(2\,0)) = 2(0\,2\,0\,2)$. This also shows that these functions preserve equivalence.

Property 17. *For any ultimately periodic word p and integer $i > 0$,*

$$p \equiv rot(p) \text{ and } p \equiv exp_i(p)$$

In addition, rotation and expansion are in fact essential when one studies the equivalence of words. Indeed, observe that for any two words p_1 and p_2 such that $p_1 \equiv p_2$, one can always obtain one from the other through a sequence of applications of *rot* and exp_i. This suggests that an equivalence class of words always has a smallest element from which all the other ones can be obtained through rotations and expansions. Given a word p, we will call $nf(p)$ the smallest word to which it is equivalent, found by applying the inverses of rotations and expansions as much as possible. Then, to decide whether $p_1 \equiv p_2$, simply check whether $nf(p_1) = nf(p_2)$.

Remark 3. Another solution is to check whether the clocks $u_1(v_1)^\omega$ and $u_2(v_2)^\omega$ are equal up to the point where both reach their periodic behavior. This can be done by computing their prefixes of length $max(|u_1|,|u_2|) + lcm(|v_1|,|v_2|)$

2.10.3 Composition

In order to prove that ultimately periodic clocks are closed by composition, we now build up a series of results on cumulative sums and compound clocks. Because we will apply these results to ultimately periodic clocks, we now assume that all the clocks involved are total. The following properties and lemmas could be extended to partial clocks by working with domains instead of plain sets, but we do not feel that the increased generality would be worth the additional formal noise.

Lemma 3. *For any total clocks w_1 and w_2, one has.*

$$(w_1 \text{ on } w_2)[i] = \sum_k w_2[k] \text{ for } \mathcal{O}_{w_1}(i-1) < k \leq \mathcal{O}_{w_1}(i)$$

Property 18. *Let us call* **drop** *the following function.*

$$\begin{array}{lcl} \textbf{drop} & : & \mathbb{N}_\perp \Rightarrow_c Stream(D) \Rightarrow_c Stream(D) \\ \textbf{drop}\ n\ xs & = & xs'\ where\ (\_, xs') = \textbf{splitAt}\ n\ xs \end{array}$$

Now, one can decompose the cumulative sum of a clock through the following equation.

$$\mathcal{O}_w(a+b) = \mathcal{O}_w(a) + \mathcal{O}_{\textbf{drop}\ a\ w}(b)$$

Lemma 4. *Let $u_1(v_1)^\omega$ and $u_2(v_2)^\omega$ be two ultimately periodic clocks such that $|u_1|_1 = |u_2|$ and $|v_1|_1 = |v_2|$. Then their composition is ultimately periodic.*

Proof. Call w the clock $u_1(v_1)^\omega$ *on* $u_2(v_2)^\omega$ to be shown ultimately periodic. Let us show that for any $i > 0$ and j, $w[i + |v_1| \times j + |u_1|] = w[i + |u_1|]$. On the one hand, we have

$$\begin{array}{lll} & w[i+|u_1|] & \\ = & \sum_{\mathcal{O}_{u_1(v_1)^\omega}(i+|u_1|-1)+1 \le k \le \mathcal{O}_{u_1(v_1)^\omega}(i+|u_1|)} u_2(v_2)^\omega[k] & \text{(Lemma 3)} \\ = & \sum_{\mathcal{O}_{(v_1)^\omega}(i-1)+\mathcal{O}_{u(v)^\omega}(|u_1|)+1 \le k \le \mathcal{O}_{(v_1)^\omega}(i)+\mathcal{O}_{u(v)^\omega}(|u_1|)} u_2(v_2)^\omega[k] & \text{(Property 18)} \\ = & \sum_{\mathcal{O}_{(v_1)^\omega}(i-1)+|u_1|_1+1 \le k \le \mathcal{O}_{(v_1)^\omega}(i)+|u_1|_1} u_2(v_2)^\omega[k] & \text{(Definition of } |s|_1) \\ = & \sum_{\mathcal{O}_{(v_1)^\omega}(i-1)+1 \le k \le \mathcal{O}_{(v_1)^\omega}(i)} u_2(v_2)^\omega[k+|u_1|_1] & \\ = & \sum_{\mathcal{O}_{(v_1)^\omega}(i-1)+1 \le k \le \mathcal{O}_{(v_1)^\omega}(i)} u_2(v_2)^\omega[k+|u_2|] & \text{(Hyp. } |u_1|_1 = |u_2|) \\ = & \sum_{\mathcal{O}_{(v_1)^\omega}(i-1)+1 \le k \le \mathcal{O}_{(v_1)^\omega}(i)} (v_2)^\omega[k] & \end{array}$$

and on the other

$$\begin{array}{lll} & w[i+|v_1|j+|u_1|] & \\ = & \sum_{\mathcal{O}_{u_1(v_1)^\omega}(i+|v_1|j+|u_1|-1)+1 \le k \le \mathcal{O}_{u_1(v_1)^\omega}(i+|v_1|j+|u_1|)} u_2(v_2)^\omega[k] & \text{(Lemma 3)} \\ = & \dots & \\ = & \sum_{\mathcal{O}_{(v_1)^\omega}(i+|v_1|j-1)+1 \le k \le \mathcal{O}_{(v_1)^\omega}(i+|v_1|j)} (v_2)^\omega[k] & \text{(As above)} \\ = & \sum_{\mathcal{O}_{(v_1)^\omega}(i-1)+\mathcal{O}_{(v_1)^\omega}(|v_1|j)+1 \le k \le \mathcal{O}_{(v_1)^\omega}(i)+\mathcal{O}_{(v_1)^\omega}(|v_1|j)} (v_2)^\omega[k] & \text{(Property 18 and } i>0) \\ = & \sum_{\mathcal{O}_{(v_1)^\omega}(i-1)+|v_1|_1 j+1 \le k \le \mathcal{O}_{(v_1)^\omega}(i)+|v_1|_1 j} (v_2)^\omega[k] & \text{(Definition of } |s|_1) \\ = & \sum_{\mathcal{O}_{(v_1)^\omega}(i-1)+1 \le k \le \mathcal{O}_{(v_1)^\omega}(i)} (v_2)^\omega[k+|v_1|_1 j] & \\ = & \sum_{\mathcal{O}_{(v_1)^\omega}(i-1)+1 \le k \le \mathcal{O}_{(v_1)^\omega}(i)} (v_2)^\omega[k+|v_2|j] & \text{(Hyp. } |v_1|_1 = |v_2|) \\ = & \sum_{\mathcal{O}_{(v_1)^\omega}(i-1)+1 \le k \le \mathcal{O}_{(v_1)^\omega}(i)} (v_2)^\omega[k] & \end{array}$$

thus the equality holds. We conclude by Lemma 2. □

To show that arbitrary ultimately periodic clocks are closed by composition, we build upon this lemma. We will show that its hypotheses are not restrictive since they can always be achieved in all non-trivial cases.

Theorem 3. *The composition of two ultimately periodic clocks $p_1 = u_1(v_1)^\omega$ and $p_2 = u_2(v_2)^\omega$ is ultimately periodic.*

Proof. Call w the clock p_1 *on* p_2. In the case where $|v_1|_1$ is 0, this clock is ultimately periodic since then $rate(w) = 0$ and hence $w = u(0)^\omega$ for some u by definition. Now, assuming $|v_1|_1 > 0$, the goal is to find $p'_1 = u'_1(v'_1)$ and $p'_2 = u'_2(v'_2)$ satisfying the hypotheses of Lemma 4 and such that $p_1 \equiv p'_1$ and $p_2 \equiv p'_2$.

First, consider the prefixes u_1 and u_2. If $|u_1|_1 < |u_2|$, increase $|u_1|$ until it is larger than $|u_2|$, obtaining a new prefix u'_1; because $|v_1| > 0$, this is always possible. Then, if $|u'_1|_1 > |u_2|$, increase $|u_2|$ by rotating $|u_2| - |u'_1|_1$ times the word p_1, obtaining a sequence u'_2. Second, the periodic parts v_1 and v_2 should be expanded into v'_1 and v'_2 so that $|v'_1|_1 = |v'_2|$. This is always possible because of the assumption that $|v_1|_1 > 0$ and because $|v_2| > 0$ by definition.

We thus look for words $u'_1(v'_1)$ and $u'_2(v_2)'$ with the following lengths for their prefixes and periodic parts.

$$\begin{array}{rclcrcl} |u'_1| & = & max(|u_1|, \mathcal{I}_{u_1(v_1)^\omega}(|u_2|)) & & |v'_1| & = & \frac{|v_1| \times lcm(|v_1|_1, |v_2|)}{|v_1|_1} \\ |u'_2| & = & max(|u_2|, \mathcal{O}_{u_1(v_1)^\omega}(|u'_1|)) & & |v'_2| & = & lcm(|v_1|_1, |v_2|) \end{array}$$

This words can be obtained using the following operations, for $i \in [1,2]$.

$$p'_i = exp_{a_i}(rot^{b_i} p_i) \text{ with } a_i = \frac{|v'_i|}{|v_i|} \text{ and } b_i = |u'_i| - |u_i|$$

These two words have been constructed such that $|u'_1|_1 = |u'_2|$ and $|v'_1|_1 = |v'_2|$, and are respectively equivalent to p_1 and p_2 by virtue of Property 17. Lemma 4 thus proves that w is ultimately periodic. □

Remark 4. The proof of Theorem 3 actually gives an algorithm to compute the word $u_3(v_3)$ such that $u_3(v_3)^\omega = u_1(v_1)^\omega$ *on* $u_2(v_2)^\omega$. Since $|u_3| = |u'_1|$ and $|v_3| = |v'_1|$, the formulas present in the proof give the size of the prefix and periodic part of the composition. One can then compute the composition, for example by applying Lemma 3 for $i \in [1, |u_3| + |v_3|]$.

2.10.4 Adaptability

We finally turn to the question of k-adaptability of ultimately periodic clocks. Remember that according to Definition 8 two clocks are k-adaptable if they are both synchronizable and if the first k-precedes the second. These two notions are decidable for ultimately periodic clocks, and thus so is k-adaptability.

Synchronizability Let us begin with synchronizability. Two clocks are synchronizable if they have the same rate. In the case of an ultimately periodic clock presented by a word $u(v)$, the rate is a computable rational number, and synchronizability is thus decidable.

Property 19. *The rate of an ultimately periodic clock* $u(v)^\omega$ *is characterized by*

$$rate(u(v)^\omega) = \frac{|v|_1}{|v|}$$

Precedence For the precedence relation, the reasoning is similar to that of Remark 3: to check whether a clock $w_1 = u_1(v_1)^\omega$ k-precedes a clock $w_2 = u_2(v_2)^\omega$, one checks that

$$\mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i+k)$$

for all i up to $max(|u_1|,|u_2|) + lcm(|v_1|,|v_2|)$.

Examples We finish with some examples, in addition to the ones already given in Section 2.7.

Example 1. The inequation $(1)^\omega <:_2 0^2(1)^\omega$ models a two-place buffer. No instantaneous communication happens; in fact, this buffer is even able to satisfy the demands of the consumer for the current and next time step without consuming its current input: the input is 2-adaptable to the output.

Example 2. While $(1\,1\,0\,0\,1)^\omega <:_0 (0\,0\,1\,1\,1)^\omega$ also models a two-place buffer, there is no constant decoupling between the producer and the consumer. The two clocks are 0-adaptable but not 1-adaptable. This comes from the fact that at the fifth time step there is an instanataneous dependency between production and consumption.

Example 3. The inequation $(1)^\omega <:_0 (0\,0\,3)^\omega$ models a buffer which stores its input for the first two time steps out of three. At the third time step, the accumulated values are transmitted to the consumer. This buffer implements a form of trade-off between latency and throughput.

2.10.5 Scattering and Gathering

We have seen in Section 2.8 how the gathering and scattering relations model how clocks evolve when leaving and entering a local time scale. When the clocks involved are all ultimately periodic, these relations are decidable: one may decide $p_1 \uparrow_p p_2$ or $p_2 \downarrow_p p_1$, with p, p_1 and p_2 ultimately periodic words, by checking whether p *on* $p_1 \equiv p_2$. Let us give some examples that should help the reader build intuitions.

Example 4. The examples of gathering that do not fuse data are arguably the simplest ones. This corresponds to local time scales that hide time steps at which nothing was computed. From the point of view of clocks, this can also be seen as the case where gathering *removes* zeroes. Such examples include $(1\,0)^\omega \uparrow_{(2)^\omega} (1)^\omega$ and $(0\,1)^\omega \uparrow_{(2)^\omega} (1)^\omega$ which model data going out of a local time scale driven by $(2)^\omega$. The internal clock is not necessarily binary, as we have $(0\,2\,0)^\omega \uparrow_{(3)^\omega} (2)^\omega$ and $(3\,0)^\omega \uparrow_{(2)^\omega} (3)^\omega$ for instance. The external clock may still retain some zeroes, as in $(1\,0)^\omega \uparrow_{(1\,2)^\omega} (1\,1\,0\,1)^\omega$ or $(0\,1\,0\,0)^\omega \uparrow_{(2)^\omega} (1\,0)^\omega$. Note that in all these cases, the driving clock is striclty positive, since it must not insert any zero.

Example 5. We know that $w_1 \uparrow_w w_2$ holds iff $w_2 \downarrow_w w_1$. Applying this principle to the previous examples gives us scattering relations that do not split data. This is uninteresting when the external clock is binary, since there is striclty speaking no data to be split. Consider for instance $(2)^\omega \downarrow_{(3)^\omega} (0\,2\,0)^\omega$. The chunk of size two is consumed as a whole at the second local time step out of three, rather than split into several ones. From the point of view of clocks, these scattering relations are the ones that only *add* zeroes.

Example 6. Some gathering relations fuse several chunks together, and some scattering relations divide them apart. This is the case in $0^2(1\,0\,1)^\omega \uparrow_{(3)^\omega} 1(2)^\omega$ and $1(2)^\omega \downarrow_{(3)^\omega} 0^2(1\,0\,1)^\omega$, for instance. Note that this cannot happen when scattering a binary clock, since in this case there are no chunks to divide.

Example 7. Local time scales driven by binary clocks are interesting, as they correspond to the *activation conditions* of Lustre and SCADE. They are however expressed at the level of clocks, while activation conditions are untyped operators. Gathering by a binary clock inserts some zeroes, as in $(3\,2)^\omega \uparrow_{1(1\,0)^\omega} 3(2\,0\,3\,0)^\omega$ or $(1\,0)^\omega \uparrow_{(0\,1)^\omega} (0\,1\,0\,0)^\omega$. This models the fact that during some time steps the local time scale is idle, and thus produces no data.

Example 8. It is important to understand the effect of scattering by a binary clock, which is slightly counterintuitive. Consider how a stream of clock $(2\,0)^\omega$ may enter a local time scale driven by $(1\,0)^\omega$. At the first external step, the two values are used in the local time scale. At the second external step, no data is availabe, which is fine since *there is no corresponding local step*. This shows that the internal clock is $(2)^\omega$ and explains why $(2\,0)^\omega \downarrow_{(1\,0)^\omega} (2)^\omega$ holds. In contrast, there is no clock *wi* such that $(1)^\omega \downarrow_{(1\,0)^\omega} wi$ holds, since the value arriving at the second external step cannot be processed by the time scale, which is idle at this point. Thus, from the point of view of clocks, this kind of scattering can be understood as an operation which filters zeroes from the input clock when they occur at the same position in the driving clock. In operational terms, this corresponds to removing empty chunks that arrive when the time scale is idle. Moreover, the relation guarantees that when no local step is performed no input data may be present.

2.11 Bibliographic notes

We have described basic properties of streams and segmented streams in a denotational fashion, and shown how to use clocks to characterize the relation between these two domains. By studying the convergence of the rescaling function, we had a taste of how clocks can be used to describe temporal phenomena. The next chapter builds upon these ideas to describe a concrete language. Before that, we discuss some bibliographical references.

Synchronous languages As explained in the introduction, the ideas of the present work can be traced back to the seminal paper of Kahn [1974] on deterministic parallel programs. The contribution of this chapter is to allow the presence of integers greater than one in clocks, or, equivalently, of allowing clocks that assign the same time step to several consecutive elements of a stream. The notion of rescaling goes hand-in-hand with integer clocks, and was a motivation for their introduction.

The use of clocks to compile synchronous languages comes from Lustre [Caspi et al., 1987]. They became vital to the compilation process with Lucid Synchrone [Pouzet, 2006] which pioneered the compilation techniques used in SCADE6 [Biernacki et al., 2008]. This chapter adopts the point of view of n-synchronous Kahn networks [Cohen et al., 2006] as implemented in the language Lucy-n [Mandel et al., 2010] by Plateau and Mandel, including their description

of the monoid of clocks and of the adaptability relation. In particular, we pay heavy debt to the exposition of binary clocks found in chapter 3 of the PhD thesis of Plateau [2010]. This latter thesis itself builds upon the work of Caspi and Halbwachs [1986].

From a more semantic point of view, we feel that the work of Caspi [1992] on the nature and role of clocks is of paramount importance. In this paper, Caspi shows how clocks characterize subsets of the inputs of a process network where the network can be implemented as a finite-state machine. Interestingly, he explicitly ponders the case of integer clocks but chooses not to investigate it further. Quoting Caspi:

> We may now understand why the oversampling case is more complex [...] the corresponding [soundness] theorem would be more difficult to state, as the mgsm [multiple generalized sequential machine] property depends now on the position of the fbys, along the cycles of the network, with respect to the position of corresponding lasts and muxs.

The foundational aspects of the relation between binary clocks and Kahn networks are studied and developed in the second half of the PhD thesis of Gérard [2013]. A similar treatment of integer clocks has not yet been achieved.

Regarding the rescaling operation, this is to our knowledge the first time it has been described in a synchronous functional language. In synchronous languages in general, let us mention the work of Mandel et al. [2013] on adding clock domains to ReactiveML, the work of Gemünde et al. [2013] on clock refinement in Quartz, and the work of Benveniste et al. [1992] on the foundations of the relational language Signal. More generally, one may find analogues to special cases of rescaling in the literature. On the one hand, rescaling by binary clocks broadly corresponds to the activation conditions of Lustre and SCADE (as described by Halbwachs [2005, Section 4], for example), and to the guards inserted by modern compilers for synchronous functional languages during compilation [Biernacki et al., 2008]. On the other hand, rescaling by strictly positive constant clocks corresponds to loop unrolling or circuit unfolding (e.g., [Parhi and Messerschmitt, 1991]).

Static scheduling Another related line of work lies in models for static, cyclic scheduling, such as the well-known *Synchronous Dataflow Graphs* from Lee and Messerschmitt [1987] or the older *Computation Graphs* of Karp and Miller [1966]. These models typically consider a sub-class of Kahn process networks where communication rates between processes are statically known. Hence, one can apply powerful scheduling techniques at compile-time to check properties—for example the absence of deadlocks–or look for schedules optimizing various criteria—for example throughput, or buffer sizes. We will see a modest analogue to these techniques in our setting in Chapter 5.

In contrast to most of these works, (n-)synchronous languages offer the ability to compose programs into bigger ones. The essential tool is clock composition, which makes it possible to use clocks for describing both communication patterns *and* schedules. In particular, if we see a clock w_1 as a schedule and a clock w_2 as the communication pattern of some component, then w_1 *on* w_2 can be seen as the updated communication pattern of the same component

after it has been scheduled according to w_1. Also, while these models describe processes communicating in a bursty manner, their schedules remain binary clocks, and they do not consider time scale changes.

Another difference is that the scheduling community only considers schedules that are finitely presentable (typically periodic or ultimately periodic), in contrast to the theory of this chapter where clocks are arbitrary integer streams. We believe that one of the main contributions of the programming language point of view on scheduling, as introduced in the original n-synchronous paper [Cohen et al., 2006], is the separation it enables between algorithms and semantics. It is possible to describe the nature and properties of clocks in a general setting and state results once and for all. The algorithmic side can then reuse the general properties and focus on clever scheduling techniques.

Chapter 3

Language

We now turn to the description of a concrete language for writing stream functions. This language is equipped with *clock types*, in order to ensure that programs execute within bounded time and space. Its description relies on the basic semantic tools introduced in Chapter 2. More precisely, this chapter introduces and studies a minimal language, named μAS, where semantic questions can be studied with as little formal baggage as possible. As such, it is quite restricted compared to a more usable language, yet exhibits all the interesting issues arising from integer clocks and the scaling operation. We study more expressive languages in Chapter 5.

This chapter goes as follows. First, we introduce the basic features of the language. It is a variant of the linear λ-calculus with fixpoints and ad hoc operators for stream manipulation. We describe a simple untyped semantics for the language in terms of a universal domain; it serves as an ideal, clock-free, semantics where programs manipulate streams of scalars. Then, we give a clock type system for the language, and a typed semantics interpreting typing derivations as functions on segmented streams. The type system includes rules for buffering and scaling. We discuss and prove its soundness using a time-indexed realizability predicate. We conclude by discussing how the typed semantics can be seen as witnesses for properties of the untyped one, and more generally the relationship between the two.

3.1 Syntax and Untyped Semantics

Clocks, as presented in the previous chapter, are a denotational tool for describing operational properties of streams and stream functions. It should always be possible, however, to understand the meaning of a program in a way that is independent of clocks. In technical terms, we wish to have a *naive* semantics where functions act upon streams of scalars.

What are the main ingredients of a functional language for stream manipulation? We claim that they are twofold. First, one should have the ability to manipulate streams and apply various operations to them. We can restrict ourselves to three elementary operations: scalar-wise transformations, with constant streams as a special case; stream sampling, where one removes some elements from a stream; and stream merging, where one interleaves elements from two streams. Both sampling and merging take a boolean stream describing whether to keep an

e	::=	x	Variable
	$\mid$	$\texttt{fun}\, x.e$	Function
	$\mid$	$e\, e$	Application
	$\mid$	(e,e)	Pair constructor
	$\mid$	$\texttt{let}\,(x,x) = e\,\texttt{in}\,e$	Pair destructor
	$\mid$	$\texttt{fix}\, e$	Recursive definition
	$\mid$	s	Constant stream
	$\mid$	op	Lifted stream operator
	$\mid$	$\texttt{merge}\, p$	Stream merging
	$\mid$	$\texttt{when}\, p$	Stream sampling
s	::=	$n \in Int$	Integer literal
	$\mid$	$b \in \mathbb{B}$	Boolean literal
	$\mid$	$\ldots$	
op	::=	$+ \mid * \mid \ldots$	Scalar function
p	::=	$u(v)$	Ultimately periodic word
u	::=	s^*	List of scalar literals
v	::=	s^+	Non-empty list of scalar literals

Figure 3.1: Syntax of μAS

element or not (for sampling) or from which stream to take (for merging). Second, one should be able to use the usual features of the λ-calculus to define and combine functions as usual in functional programming. We see anonymous functions, applications and the manipulation of pairs as *wiring combinators* enabling the modular construction of programs. In addition, the language should have a fixpoint operator for introducing feedback loops in the Kahn network.

In addition to these general principles, one should design the language so that we can equip it with a clock type system. As usual, capturing a large class of correct programs requires an expressive type system with a complex metatheory. Because in this chapter we want the latter to be as simple as possible, we restrict ourselves to a simplistic programming language. In particular, we would like to have a simple one-to-one correspondence between clocks and clock types, a principle which does not hold in languages such as Lucid Synchrone which feature clock polymorphism.

For all these reasons, we study a language where programs have a *ultimately periodic* behavior. This is a quite restrictive choice since, for example, it makes it impossible to write functions for which the convergence of outputs depends on the *value* of inputs. It is not completely inexpressive however, as one can write non-trivial programs such as the examples found in the thesis in Plateau [2010].

Syntax The syntax of the language is given in Figure 3.1. We suppose given an infinite countable set of variables $V = \{x, y, z \ldots\}$. The first six constructs of the language describe

a λ-calculus with pairs and recursive definitions. We also have constant streams s, with s a scalar literal, pointwise liftings of scalar operators *op* such as addition or multiplication, stream merging and sampling. Integer literals belong to a finite set *Int* of bounded natural numbers. This bound is arbitrary; we take $2^{64}-1$ in order to model unsigned 64 bit integers. Merging and sampling are parametrized by a *condition*, which is an ultimately periodic word $p = u(v)$, with u the (possibly empty) prefix and v a periodic pattern.

The syntax of this language and its untyped semantics make it a straightforward variation on Lustre and its descendants, such as the functional core of Lucid Synchrone or Lucy-n. It shares with Lucid Synchrone its higher-order nature, and with Lucy-n the restriction to ultimately periodic conditions. There is one small syntactic difference however: we choose to write `when` and `merge` with their condition on the side. Readers familiar with synchronous functional languages might find this formulation unappealing in the case of `when`, which is generally written in an infix manner—in our syntax one writes `when` p e rather than e `when` p. This difference is purely stylistic however, and it will pay off by lowering the amount of bureaucracy in later definitions, statements and proofs.

Semantic combinators We now turn to the description of the untyped semantics of the language. It is a function from the syntax of programs to some mathematical model, here domains, which makes it a denotational semantics. Since our semantics is completely untyped, it attributes meanings to all syntactically valid programs, even nonsensical ones such as $1 + (\texttt{fun}\ x.x)$. We again rely on domain theory and its built-in handling of partiality to build a space where this kind of operation can be described. Let us describe the traditional solution to this problem that we adopt here.

Our language includes streams of scalar values, pairs and functions. We suppose given a domain $\mathbb{V}$ of scalar values such that the e-p pairs

$$\begin{array}{lll} (bool, unbool) & \in & \mathbb{B}_\perp \triangleleft \mathbb{V} \\ (int, unint) & \in & Int_\perp \triangleleft \mathbb{V} \end{array}$$

exist. Note that the streams of the language cannot transport complex values such as functions but only simple scalars. Now, what would be a good domain $\mathbb{K}$ for interpreting the language? Since it includes streams of scalars, singletons, pairs, and functions, we want the e-p pairs

$$\begin{array}{lllll} (stream, unstream) & \in & \mathbb{V} & \triangleleft & \mathbb{K} \\ (unit, ununit) & \in & \mathbb{1}_\perp & \triangleleft & \mathbb{K} \\ (pair, unpair) & \in & \mathbb{K}\times\mathbb{K} & \triangleleft & \mathbb{K} \\ (fun, unfun) & \in & \mathbb{K}\Rightarrow_c\mathbb{K} & \triangleleft & \mathbb{K} \end{array}$$

to exist. Since we need nothing more, we define $\mathbb{K}$ as the solution of the following recursive domain equation.

$$\mathbb{K} \quad \cong \quad Stream(\mathbb{V}) \oplus (\mathbb{K}\times\mathbb{K}) \oplus (\mathbb{K}\Rightarrow_c\mathbb{K})$$

ML programmers may think of this domain as a (denotational analogue to a) recursive variant type, with one variant for each kind of value in our semantics.

The untyped semantics of the language is unremarkable since it is a λ-calculus with ad hoc constructs for stream processing. The only definitions that are not standard in the theory of functional languages are those concerning stream-related constructs. We define the denotational counterpart to each of these four constructs. They appeared first in Caspi and Pouzet [1996].

$$\begin{array}{l} \mathbf{const} \quad \in \mathbb{S} \Rightarrow_c \mathit{Stream}(\mathbb{S}) \\ \mathbf{const}\ s = s.\mathbf{const}\ s \end{array}$$

The **const** is used to interpret literal constant streams. It repeats a scalar constant s an unbounded number of times.

$$\begin{array}{lcl} \mathbf{merge} & \in & \mathit{Stream}(\mathbb{S}) \Rightarrow_c \mathit{Stream}(\mathbb{S}) \times \mathit{Stream}(\mathbb{S}) \Rightarrow_c \mathit{Stream}(\mathbb{S}) \\ \mathbf{merge}\ ((\mathit{bool}\ 1).cs)\ ((x.xs), ys) & = & x.(\mathbf{merge}\ cs\ (xs, ys)) \\ \mathbf{merge}\ ((\mathit{bool}\ 0).cs)\ (xs, (y.ys)) & = & y.(\mathbf{merge}\ cs\ (xs, ys)) \end{array}$$

The function $\mathbf{merge}\ cs\ (xs, ys)$ intersperses the elements of xs and ys according to a condition, the stream cs corresponding to p in the syntax of Figure 3.1. When the first element of cs is one, the next output is the first element of xs, and symmetrically for zero and ys. The condition is a stream of scalars, all of which should be booleans; hence the use of the embedding *bool* in the left-hand side of the above definition to express an implicit use of the *unbool* projection in the definition of the function.

$$\begin{array}{lcl} \mathbf{when} & \in & \mathit{Stream}(\mathbb{S}) \Rightarrow_c \mathit{Stream}(\mathbb{S}) \Rightarrow_c \mathit{Stream}(\mathbb{S}) \\ \mathbf{when}\ ((\mathit{bool}\ 0).cs)\ (x.xs) & = & \mathbf{when}\ cs\ xs \\ \mathbf{when}\ ((\mathit{bool}\ 1).cs)\ (x.xs) & = & x.(\mathbf{when}\ cs\ xs) \end{array}$$

The sampling function **when** filters its input according to a condition cs. More precisely, the function removes elements of the stream xs when the boolean of same rank in cs is false.

$$\begin{array}{lcl} \mathbf{map}_n & \in & (\mathbb{S} \times \cdots \times \mathbb{S} \Rightarrow_c \mathbb{S}) \Rightarrow_c (\mathit{Stream}(\mathbb{S}) \times \cdots \times \mathit{Stream}(\mathbb{S})) \Rightarrow_c \mathit{Stream}(\mathbb{S}) \\ \mathbf{map}_n & = & \lambda f.\lambda(x_1.xs_1, \ldots, x_n.xs_n).(f\ (x_1, \ldots, x_n)).(\mathbf{map}_n\ f\ (xs_1, \ldots, xs_n)) \end{array}$$

The family of function $\mathbf{map}_{n \in \mathbb{N}}$ applies an n-ary function f pointwise to n streams. Following categorical spirit, we sometimes write $\mathit{Stream}(f)$ for $\mathbf{map}_1\ f$.

$$\begin{array}{lcl} \mathbf{unroll} & \in & \mathit{List}(\mathbb{S}) \Rightarrow_c \mathit{List}(\mathbb{S}) \Rightarrow_c \mathit{Stream}(\mathbb{S}) \\ \mathbf{unroll}\ (n; u)\ v & = & n.(\mathbf{unroll}\ u\ v) \\ \mathbf{unroll}\ []\ v & = & \mathbf{unroll}\ v\ v \end{array}$$

Finally, the function **unroll** builds the ultimately periodic stream $\mathbf{unroll}\ u\ v$ from its syntactic representation as a pair of lists $u(v)$.

Environments An expression e may contain free variables, the set of which is written $FV(e)$ and defined as in Figure 3.2. The interpretation must be parametrized by an *environment* assigning a value to each free variable. We adopt a simple solution: an environment simply is a

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\texttt{fun}\, x.e) &= FV(e) \setminus \{x\} \\
FV(e\ e') = FV((e,e')) &= FV(e) \cup FV(e') \\
FV(\texttt{let}\ (x,y) = e_1\ \texttt{in}\ e_2) &= FV(e_1) \cup (FV(e_2) \setminus \{x,y\}) \\
FV(\texttt{fix}\, e) &= FV(e) \\
FV(s) = FV(op) = FV(\texttt{merge}\ p) = FV(\texttt{when}\ p) &= \varnothing
\end{aligned}
$$

Figure 3.2: Syntax of μAS - free variables

continuous function from V, seen as a discretely ordered predomain, to $\mathbb{K}$. We abbreviate the domain of $\mathbb{K}$-valued environment as

$$Env(\mathbb{K}) \stackrel{\text{def}}{=} V \Rightarrow_c \mathbb{K}$$

An environment $\sigma \in Env(\mathbb{K})$ is extended with a new binding $(x,k) \in V \times \mathbb{K}$ via

$$\sigma[x \mapsto k] \stackrel{\text{def}}{=} \lambda y.\text{if } y = x \text{ then } k \text{ else } \sigma(y)$$

and the value of a variable x can be obtained by applying σ to x.

Interpretation functions Interpretation functions map syntactic objects to their denotations. We define four functions, one for each distinct syntactic category. The first two categories, *s* and *op*, are left abstract since the precise language of scalars and operators is mostly independent from the rest of the language. We assume that their interpretations have been given Overloading notations, we write both of these functions $[\![\_]\!]$, relying on context to disambiguate them. These functions should have type

$$
\begin{aligned}
[\![s]\!] &\in \mathbb{S} \\
[\![op]\!] &\in \mathbb{S} \times \mathbb{S} \Rightarrow_c \mathbb{S}
\end{aligned}
$$

since scalar literals correspond to elements of $\mathbb{S}$ and operators to binary functions on $\mathbb{S}$.

We later reuse the functions $[\![\_]\!]$ directly in the typed semantics. In contrast, the interpretation functions for conditions and expressions are specific to the untyped semantics and, overloading notation again, we write them $\mathbf{K}[\![\_]\!]$. We use the letter **K** to recall the terms *Kahn semantics*, sometimes used informally to describe the untyped semantics of synchronous functional languages [Pouzet, 2002]. The semantics $\mathbf{K}[\![p]\!]$ for ultimately periodic words $p = u(v)$, with u and v lists of scalars, consists in using the **unroll** function defined above. It is given in Figure 3.3 (a).

The interpretation $\mathbf{K}[\![e]\!]$ of an expression e is a map from environments $Env(\mathbb{K})$ to $\mathbb{K}$. Its definition is given in Figure 3.3 (b). It is mainly routine that injects and extracts values from or to the universal domain $\mathbb{K}$, as needed. The first six clauses are standard in interpretations of the untyped (or *pure*) λ-calculus. The remaining ones dedicated to stream processing constructs rely on the previously defined functions, wrapped with the appropriate embedding and projections to make them inhabitants of $\mathbb{K}$.

$$\boxed{\mathbf{K}[\![p]\!]}$$

$$\begin{array}{lcl}
\mathbf{K}[\![p]\!] & \in & \mathit{Stream}(\mathbb{V}) \\
\mathbf{K}[\![u(v)]\!] & = & \mathbf{unroll}\,(\mathit{List}(\mathbf{K}[\![_]\!])\;u)\;(\mathit{List}(\mathbf{K}[\![_]\!])\;v)
\end{array}$$

(a) Semantics of Ultimately Periodic Words

$$\boxed{\mathbf{K}[\![e]\!]}$$

$$\begin{array}{lcl}
\mathbf{K}[\![e]\!] & \in & \mathit{Env}(\mathbb{K}) \Rightarrow_c \mathbb{K} \\
\mathbf{K}[\![x]\!]\;\sigma & = & \sigma(x) \\
\mathbf{K}[\![\mathtt{fun}\;x.e]\!]\;\sigma & = & \mathit{fun}(\lambda v.\mathbf{K}[\![e]\!]\;\sigma[x \mapsto v]) \\
\mathbf{K}[\![e\;e']\!]\;\sigma & = & (\mathit{unfun}\;(\mathbf{K}[\![e]\!]\;\sigma))(\mathbf{K}[\![e']\!]\;\sigma) \\
\mathbf{K}[\![(e_1,e_2)]\!]\;\sigma & = & \mathit{pair}\;(\mathbf{K}[\![e_1]\!]\;\sigma,\mathbf{K}[\![e_2]\!]\;\sigma) \\
\mathbf{K}[\![\mathtt{let}\;(x,y) = e_1\;\mathtt{in}\;e_2]\!]\;\sigma & = & (\lambda(d_x,d_y).\mathbf{K}[\![e_2]\!]\;\sigma[x \mapsto d_x, y \mapsto d_y])\;(\mathit{unpair}\;(\mathbf{K}[\![e_1]\!]\;\sigma)) \\
\mathbf{K}[\![\mathtt{fix}\;e]\!]\;\sigma & = & \mathit{fix}(\mathit{unfun}\;(\mathbf{K}[\![e]\!]\;\sigma)) \\
\mathbf{K}[\![s]\!]\;\sigma & = & \mathit{stream}\;(\mathbf{const}\;[\![s]\!]) \\
\mathbf{K}[\![op]\!]\;\sigma & = & \mathit{fun}(\mathit{stream} \circ \mathbf{map_2}\;[\![op]\!] \circ (\mathit{unstream} \times \mathit{unstream}) \circ \mathit{unpair}) \\
\mathbf{K}[\![\mathtt{merge}\;p]\!]\;\sigma & = & \mathit{fun}(\mathit{stream} \circ \mathbf{merge}\;[\![p]\!] \circ (\mathit{unstream} \times \mathit{unstream}) \circ \mathit{unpair}) \\
\mathbf{K}[\![\mathtt{when}\;p]\!]\;\sigma & = & \mathit{fun}\;(\mathit{stream} \circ \mathbf{when}\;[\![p]\!] \circ \mathit{unstream})
\end{array}$$

(b) Semantics of Expressions

Figure 3.3: Untyped semantics

Discussion The language as defined above can be seen as an untyped fragment of Haskell with a unique base type, that of streams. In particular, it allows programs of dubious behavior in the context of real-time stream processing. Let us describe three classes of behaviors that we wish to rule out.

1. Programs having the usual "type errors" where one tries to combine data having incompatible shapes, such as for example $1\;(\mathtt{fun}\;x.x)$ and $1+(4,2)$. In general their interpretation in the untyped semantics defined above is $\bot_{\mathbb{K}}$.

2. Programs with recursive definitions that produce partial rather than total streams, such as $\mathtt{fix}\;(\mathtt{fun}\;x.x)$. We may think of these programs as process networks that deadlock.

3. Programs whose memory usage grows without bound as time passes. They correspond to process networks with buffers that cannot be bounded without introducing artificial deadlocks, or that have an unbounded number of processes.

The first issue is addressed by traditional type systems. Let us discuss the ways in which memory usage may increase over time.

First, there are programs that combine streams in ways that inherently use an unbounded amount of memory for buffering intermediate results. Consider for example the function

$$\texttt{fun}\ x.\,(x + \texttt{when}\ (1\ 0)\ x)$$

which requires buffering an unbounded amount of elements of the stream x as time passes. Such programs are well-known in functional synchronous programming and it is the role of ordinary clock types to reject them.

Second, recursive functions may also use an unbounded amount of memory for storing intermediate results. A practical consequence of this behavior in general-purpose functional languages is the possibility of stack overflow. Real-time synchronous languages such as Lustre generally forbid such functions.

Third, higher-order functions bring their own set of problems. The standard modular scheme for compiling higher-order functions assumes both dynamic memory management and the presence of code pointers in the target language. While the removal of certain features, such as streams of functions, may help avoid dynamic memory allocation, the need for code pointers remains.

We tackle all these problems in the next section through the use of a dedicated type system ensuring that programs are free of type errors, produce infinite streams, and work within bounded memory. Our final objective is to compile well-typed programs to static hardware.

3.2 Type System

3.2.1 Motivation

The problems outlined above stem from the unrestricted use of three features: sampling and merging stream combinators, recursive definitions, and higher-order functions. We would like to reject programs that use these features in a way that is either unsafe, or that cannot be compiled to circuits. Since these questions are undecidable in general, we settle for a sufficient syntactic condition; since we wish to allow for separate compilation, we look for modular criteria. These goals suggest the use of some sort of type system, which we outline in the next paragraphs before giving its formal description.

Why clock types? Memory usage problems described above are linked with the growth of streams along program execution. Say that a stream function is *length-preserving* if, given inputs converging up to n steps, its outputs converge up to n steps. Such functions are well-behaved in that they map total streams to total streams; it is also easier to characterize whether they can be implemented as finite state machines. In particular, if one sees the action of running a state machine on an input sequence as a stream function, then this function is length-preserving. This intuition suggests taking length-preservation as a criterion for functions that can be turned into state machines.

Unfortunately, many interesting programs sample or merge streams in ways that make their denotations fall outside of this class of functions. Yet, the notions of clock and clocked

streams introduced in Chapter 2 offer a way out. We have shown in Chapter 2 that, given two clocks w and w', there exists an embedding-retraction pair

$$(\mathit{desync}, \mathit{sync}) : \mathit{CStream}_w(\mathbb{N}_\perp) \Rightarrow_c \mathit{CStream}_{w'}(\mathbb{N}_\perp) \triangleleft \mathit{Stream}(\mathbb{N}_\perp) \Rightarrow_c \mathit{Stream}(\mathbb{N}_\perp)$$

between functions on clocked streams and functions on streams. Thus, one can project a function f on streams to a function $\mathit{sync}(f)$ on clocked streams. The interesting thing is that, under certain conditions, $\mathit{sync}(f)$ is better behaved than f itself; in particular, $\mathit{sync}(f)$ might preserve lengths even when f does not. One of the main roles of the clock type system is precisely to give a syntactic characterization of these conditions and to enforce additional properties of $\mathit{sync}(f)$, such as deadlock-freedom.

From a more technical point of view, the fact that a closed program e has type t in our system ensures that $\mathbf{K}[\![e]\!]$ has certain good properties. Because such properties are independent from t, the existence of *any* type t and typing derivation d whose conclusion is $\square \vdash e : t$ implies that they are satisfied by $\mathbf{K}[\![e]\!]$. One can then run the program using any implementation of the untyped semantics, such as a shallow embedding into Haskell. In this case, types can be considered as *irrelevant* to execution. On the other hand, we will see in the later parts of this chapter that a typing derivation d gives rise to a function $\mathbf{S}[\![d]\!]$ that precisely corresponds to the function $\mathit{sync}(f)$ from above. This function and the typing derivation that underlies it give precious information that we will use to give a more efficient implementation of the program e. Moreover, and in contrast with the reasoning above where one is interested in d and t themselves but in their mere existence, distinct derivations d get compiled to distinct state machines. We then say that types are *relevant* to the compilation process[1].

Thus, clock types are not only used to reject unsafe programs, but also provide powerful hints to describe how the same program may be compiled to different implementations. In particular, we will see how the two clock transformations studied in the previous chapter, buffers and scatter/gather, make it possible to describe and implement various space/time trade-offs in an abstract manner.

Because μAS features only ultimately periodic conditions, in this chapter clock types denote ultimately periodic clocks. This makes the metatheory much simpler while still exhibiting most of the interesting features and issues related to integer clocks. In particular, a clock type denotes exactly one clock, which will no longer be the case in some of the type systems of Chapter 5.

Why linearity? We have explained that higher-order functions are problematic when one wants to statically bound memory usage. This problem hinges upon the conceptual difference between functions as first- and second-class objects.

In the case of first-order languages, functions are second-class citizens: they cannot be passed around or returned, and thus a function name always refers to some fixed piece of code. Calls to such functions can be compiled by *instantiating* the callee inside its caller, either in hardware or software.

[1] This philosophy is similar to that of Proof Theory, where one is interested not only in *provability* but in *proofs* themselves. It also corresponds to what type theorists call *proof-relevance*, the idea that the proofs of certain propositions carry non-trivial computational content.

- Hardware Description Languages such as VHDL or Verilog offer built-in facilities for instantiating closed circuits, typically called *components*. Synthesis tools replace each instance declaration with a copy of the corresponding component early in the compilation process. The resulting flat description is then turned into a physical object or a configuration file for an FPGA.

- In software, this inlining process is unnecessary: a stream function is compiled to a description of its internal state and to a transition function that can be reused freely. Each call to a function f in g gives rise to a new copy of the state of f in that of g, passed to its transition function.

In the case of higher-order functions, this approach is no longer sufficient: even if f is a function from, for example, streams of integers to streams of integers, its body may refer to variables whose values are provided by its environment at definition time. Consider for example the two stream functions below, written in the metalanguage of Chapter 2.

$$\begin{array}{lcl} g & \in & (\mathit{Stream}(\mathbb{N}_\perp) \Rightarrow_c \mathit{Stream}(\mathbb{N}_\perp)) \Rightarrow_c \mathit{Stream}(\mathbb{B}_\perp) \\ g\ f & = & \ldots \end{array}$$

$$\begin{array}{lcl} h & \in & \mathit{Stream}(\mathbb{N}_\perp) \Rightarrow_c \mathit{Stream}(\mathbb{B}_\perp) \\ h\ x & = & g\ f \text{ where } f\ y = (o \text{ where } o = x + y + 0.o) \end{array}$$

The function g is a second-order function; h receives an integer stream x and calls g with a function f that computes the cumulative sum o of the stream $x + y$, where y is the stream passed to f by g. From the perspective of g, f is a function that, given a stream of natural numbers, returns a stream of natural numbers; but the value computed by f depends on x, which g knows nothing about. The variable x is free in f and bound in h. Thus, one cannot simply instantiate the body of f inside g, since it would need to be passed with the value of x which is not available there. There are two standard solutions to this difficulty.

1. There are several whole-program techniques that statically transform a program with higher-order functions into one that is completely first-order. For example, one can replace g with its definition everywhere, and similarly for every higher-order function. Another possibility that is less expensive in terms of code size is to use the *defunctionalization* technique of Reynolds [1972].

2. As alluded to in the previous section, one can perform *closure conversion* [Landin, 1964; Appel, 2006] to implement higher-order functions in a language such as C or assembly. In our example, the function f would be represented in the compiled code of g as a pair formed of a (pointer to) the compiled body of f next to a structure holding the value of its free variables. This pair, or *closure*, would have to be created by h, storing the value of x in its second component.

Unfortunately, both approaches appear unsatisfactory to us. Whole-program transformations are incompatible with separate compilation. Closure-conversion assumes that the

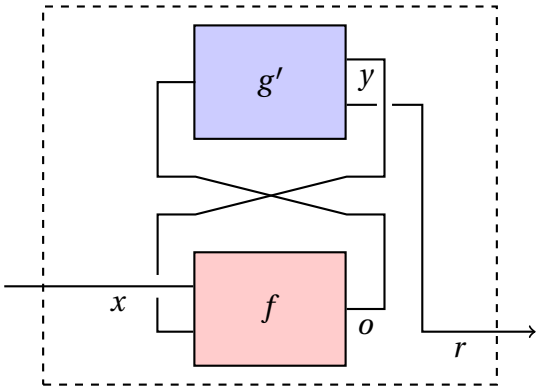


Figure 3.4: Circuit composition in h'

target language is able to represent code pointers. This is not the case in hardware: functions are compiled to circuits, which are—at least conceptually—physical objects that cannot be abstracted over.

Now, imagine that the behavior of g is to test whether the natural number 42 is mapped to 27 by f. The body of g would actually be the following.

$$\begin{array}{lcl} g & \in & (\mathit{Stream}(\mathbb{N}_\perp) \Rightarrow_c \mathit{Stream}(\mathbb{N}_\perp)) \Rightarrow_c \mathit{Stream}(\mathbb{B}_\perp) \\ g\ f & = & (f\ 42) = 27 \end{array}$$

This function g calls f exactly once. Following the terminology introduced by Girard [1987], we say that g is *linear*. In this case, the only dialogue possible between g and f is that g sends to f one input, and f sends to g one output. In the general, non-linear case, this dialogue may involve an unbounded number of exchanges between the two functions. The important point is that a linear g does not need to know the body of f, only its input and output. This suggests that g can actually be rewritten into a first-order function g' that takes as an argument the *output* of f and returns as an additional result the *input* passed to f by g.

$$\begin{array}{lcl} g' & \in & \mathit{Stream}(\mathbb{N}_\perp) \Rightarrow_c \mathit{Stream}(\mathbb{N}_\perp) \times \mathit{Stream}(\mathbb{B}_\perp) \\ g'\ o & = & (42, o = 27) \end{array}$$

Now, knowing that g is linear, h can be rewritten into a function h' adapted to g'. The function h' relies on the additional output of g' as a substitute for the variable y that was previously bound by f, making the code completely first-order.

$$\begin{array}{lcl} h' & \in & \mathit{Stream}(\mathbb{N}_\perp) \Rightarrow_c \mathit{Stream}(\mathbb{B}_\perp) \\ h'\ x & = & r \text{ where } \begin{array}{rcl} (y, r) & = & g'\ o \\ o & = & f\ y \\ f\ y & = & o \text{ where } o = x + y + 0.o \end{array} \end{array}$$

Because the resulting functions are first-order, they can be compiled as usual to digital circuits. Figure 3.4 gives an intuitive depiction of the result in the case of g' and h'. Note that

the variable x free in f has here been represented as an additional input to the function. The underlying principle can be extended to functions of arbitrary order, and will serve as a basis for the compilation of our full language.

We have seen that a higher-order function g using its functional argument f in a linear fashion can be understood as a first-order function receiving the output of f and returning the input to f. Application can then be translated to a kind of parallel feedback between f and g, as shown in Figure 3.4. This translation process will be made systematic when we compile our programs to state machine; but first, we need to make sure that each higher-order argument used by a function is used linearly. This is why our type system uses linear arrow types to check that a function is used exactly once.

As a side note, let us add that it is possible to design more flexible systems that count the number of times a function is called, and make it possible to reuse closed functions an arbitrary number of times. We will propose such a type system in Chapter 5. The simple system proposed offers a more pedagogical introduction and will serve as a basis to more expressive extensions.

3.2.2 Formal Definition

We now describe the type system based on the principles outlined above, mixing (integer) clock types and linear higher-order functions. It is based on three type constructors, a main well-typedness judgment for expressions, and a number of additional judgments that either reflect relations on clocks at the level of clock types or manage the linear aspects of the system.

Types The following grammar defines the syntax of types.

t	$::=$	$dt :: ct$	Clocked stream of scalars
	$\mid$	$t \otimes t$	Product
	$\mid$	$t \multimap t$	Function
dt	$::=$	**bool** $\mid$ **int** $\mid \ldots$	Scalar type
ct	$::=$	p	Ultimately periodic clock type
	$\mid$	ct **on** ct	Compound clock type

Types t classify expressions e. We allow expressions to denote either streams of scalars with a specified clock, products $t \otimes t'$ or functions $t \multimap t'$. We use $\otimes$ and $\multimap$ as type constructors rather than the more common $\times$ and $\to$ to insist on the linear aspect of the type system, following notations dating back from Girard [1987]. The type of streams $dt :: ct$ describes both the data type dt of the scalars held in the stream, and the clock type ct. Intuitively, data types dt should classify elements of the domain $\mathbb{S}$, and so must contain at least integer and boolean types. As explained above, clock types denote ultimately periodic clocks and can be either a literal constant p or a compound clock ct **on** ct'.

Contexts The typing judgment relies on a notion of context Γ to describe the types of free variables of an expression. Our definition is standard.

$$\begin{array}{rcll} \Gamma & ::= & \square & \text{Empty context} \\ & | & \Gamma, x : t & \text{Augmented context} \end{array}$$

Contexts are lists of bindings. A binding is a pair of a variable x and a type t. A variable x appearing in a binding of Γ is said to be bound in Γ; the set $dom(\Gamma)$ of all such variables is defined as follows.

$$\begin{array}{lcl} dom(\square) & = & \varnothing \\ dom(\Gamma, x : t) & = & dom(\Gamma) \cup \{x\} \end{array}$$

Typings Finally, we sometimes need to manipulate pairs (Γ, t) of context and type. Such a pair is called *a typing* and written $\Gamma \vdash t$ for clarity. Typings with empty contexts are often written $\vdash t$ instead of $\square \vdash t$.

Typing expressions Figure 3.5 defines the judgment $\Gamma \vdash e : t$ stating that the expression e has type t under context Γ, or that $\Gamma \vdash t$ is a typing for e. We detail each rule.

- Rule VAR assigns to a variable the type given by its rightmost binding in the context. Since the type system is linear, the rest of the context should be erasable.

$$\frac{\vdash \Gamma \text{ value}}{\Gamma, x : t \vdash x : t}\ \textsc{Var}$$

  The premise $\vdash \Gamma$ value ensures that Γ is a value context, which can be duplicated or erased. There is a corresponding judgment $\vdash t$ value for types. Both judgments are defined in Figure 3.6. A value type can only be a clocked stream or a product of value types. A value context is a context which holds only value types.

- Rule WEAKEN removes useless bindings from the context, when they bind value types.

$$\frac{\Gamma \vdash e : t \qquad \vdash t' \text{ value} \qquad x \notin FV(e)}{\Gamma, x : t' \vdash e : t}\ \textsc{Weaken}$$

- Rule LAMBDA introduces function parameters in the context.

$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \texttt{fun}\, x.\, e : t \multimap t'}\ \textsc{Lambda}$$

  This is the usual rule of typed λ-calculi.

$$\boxed{\Gamma \vdash e : t}$$

$$\textsc{Var}\ \frac{\vdash \Gamma \text{ value}}{\Gamma, x : t \vdash x : t} \qquad \textsc{Weaken}\ \frac{\Gamma \vdash e : t \qquad \vdash t' \text{ value} \qquad x \notin FV(e)}{\Gamma, x : t' \vdash e : t} \qquad \textsc{Lambda}\ \frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \texttt{fun}\, x.\, e : t \multimap t'}$$

$$\textsc{App}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \qquad \Gamma_1 \vdash e : t \multimap t' \qquad \Gamma_2 \vdash e' : t}{\Gamma \vdash e\ e' : t'} \qquad \textsc{Pair}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \qquad \Gamma_1 \vdash e_1 : t_1 \qquad \Gamma_2 \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2}$$

$$\textsc{LetPair}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \qquad \Gamma_1 \vdash e : t_1 \otimes t_2 \qquad \Gamma_2, x : t_1, y : t_2 \vdash e' : t}{\Gamma \vdash \texttt{let}\ (x, y) = e\ \texttt{in}\ e' : t}$$

$$\textsc{Fix}\ \frac{\Gamma \vdash e : t \multimap t' \qquad \vdash t' <:_1 t \qquad \vdash t' \text{ value}}{\Gamma \vdash \texttt{fix}\, e : t'}$$

$$\textsc{Const}\ \frac{}{\square \vdash s : dtof(s) :: ct} \qquad \textsc{Op}\ \frac{}{\square \vdash op : (\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct) \multimap (\mathbf{int} :: ct)}$$

$$\textsc{Merge}\ \frac{p \leq (1)}{\square \vdash \texttt{merge}\ p : (dt :: ct\ \mathbf{on}\ p) \otimes (dt :: ct\ \mathbf{on}\ \overline{p}) \multimap (dt :: ct)}$$

$$\textsc{When}\ \frac{p \leq (1)}{\square \vdash \texttt{when}\ p : (dt :: ct) \multimap (dt :: ct\ \mathbf{on}\ p)} \qquad \textsc{Sub}\ \frac{\Gamma \vdash e : t \qquad \vdash t <:_k t'}{\Gamma \vdash e : t'}$$

$$\textsc{Rescale}\ \frac{\vdash \Gamma \downarrow_{ct} \Gamma' \qquad \Gamma' \vdash e : t' \qquad \vdash t' \uparrow_{ct} t}{\Gamma \vdash e : t}$$

Figure 3.5: Typing - main judgment

$$\boxed{\vdash t \text{ value}} \text{ and } \boxed{\vdash \Gamma \text{ value}}$$

$$\textsc{ValStream}\ \frac{}{\vdash dt :: ct \text{ value}} \qquad \textsc{ValProd}\ \frac{\vdash t_1 \text{ value} \qquad \vdash t_2 \text{ value}}{\vdash t_1 \otimes t_2 \text{ value}} \qquad \textsc{ValCtxEmpty}\ \frac{}{\vdash \square \text{ value}} \qquad \textsc{ValCtxCons}\ \frac{\vdash \Gamma \text{ value} \qquad \vdash t \text{ value}}{\vdash \Gamma, x : t \text{ value}}$$

Figure 3.6: Typing - value judgment

$$\boxed{\Gamma \vdash \Gamma_1 \otimes \Gamma_2}$$

$$\textsc{SepEmpty}\ \frac{}{\square \vdash \square \otimes \square} \qquad \textsc{SepContract}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \qquad \vdash t\ \text{value}}{\Gamma, x : t \vdash \Gamma_1, x : t \otimes \Gamma_2, x : t}$$

$$\textsc{SepLeft}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \qquad x \notin dom(\Gamma_2)}{\Gamma, x : t \vdash \Gamma_1, x : t \otimes \Gamma_2} \qquad \textsc{SepRight}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \qquad x \notin dom(\Gamma_1)}{\Gamma, x : t \vdash \Gamma_1 \otimes \Gamma_2, x : t}$$

Figure 3.7: Typing - splitting judgment

- Rule APP expresses that a function of type $t \multimap t'$ can be applied to an argument of type t to obtain a result of type t'.

$$\textsc{App}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \qquad \Gamma_1 \vdash e : t \multimap t' \qquad \Gamma_2 \vdash e' : t}{\Gamma \vdash e\ e' : t'}$$

  The function and its argument are typed in contexts Γ_1 and Γ_2 obtained by splitting Γ in two. This is necessary since one cannot use Γ in both premises. This would duplicate Γ, which may contain non-value types.

  The judgment $\Gamma \vdash \Gamma_1 \otimes \Gamma_2$ describes how Γ is split in two contexts Γ_1 and Γ_2 that may only share value types. Figure 3.7 gives its definition. There is nothing to split in an empty context (SEPEMPTY). A value type can be duplicated, and hence may go in both contexts (SEPCONTRACT). In general, a binding can only go into either Γ_1 or Γ_2, not both (SEPLEFT and SEPRIGHT). Note that when a binding goes to Γ_1 via SEPLEFT, it cannot be bound in Γ_2, and conversely for SEPRIGHT. Without this restriction, it would be possible to make a variable that has been shadowed by a binder visible again, breaking the lexical scope of the language.

- Rules PAIR and LETPAIR are standard except for the splitting of contexts.

$$\textsc{Pair}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \qquad \Gamma_1 \vdash e_1 : t_1 \qquad \Gamma_2 \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2} \qquad \textsc{LetPair}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \qquad \Gamma_1 \vdash e : t_1 \otimes t_2 \qquad \Gamma_2, x : t_1, y : t_2 \vdash e' : t}{\Gamma \vdash \texttt{let}\ (x, y) = e\ \texttt{in}\ e' : t}$$

- Rule FIX allows recursive definitions as long as they are productive. The expression e should have type $t \multimap t'$. The type t' should be a value, which forbids recursive functions. More importantly, the premise $\vdash t' <:_1 t$ enforces that the output of e does not depend *instantaneously* on its input.

$$\textsc{Fix}\ \frac{\Gamma \vdash e : t \multimap t' \qquad \vdash t' <:_1 t \qquad \vdash t'\ \text{value}}{\Gamma \vdash \texttt{fix}\ e : t'}$$

We describe the judgment $\vdash t <:_k t'$ later, for now, think of it as reflecting at the level of clock types the notion of adaptability between clocks defined in Definition 8.

- Rule CONST expresses that constant literals denote streams of arbitrary clocks.

$$\textsc{Const}\ \frac{}{\square \vdash s : dtof(s) :: ct}$$

The function *dtof* maps a scalar s to its data type $dtof(s)$. Constants may inhabit any clock type ct because, intuitively, **const** s has no input stream and thus no dependencies that could constrain the clock of its output.

- Rule OP checks that the input of an operator is a pair of integer streams. The output is also a stream of integers. Inputs and output should have the same clock type.

$$\textsc{Op}\ \frac{}{\square \vdash op : (\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct) \multimap (\mathbf{int} :: ct)}$$

Intuitively, this is sound because the untyped semantics of *op* gives rise to a length-preserving function.

- Rule MERGE forces the parameter p of `merge` p to be binary. Assuming this holds, the rule expresses that `merge` is a function receiving a pair of inputs. The two inputs have type ct **on** p and ct **on** $\overline{p}$, with $\overline{p}$ denoting the elementwise negation of p.

$$\textsc{Merge}\ \frac{p \leq (1)}{\square \vdash \texttt{merge}\ p : (dt :: ct\ \mathbf{on}\ p) \otimes (dt :: ct\ \mathbf{on}\ \overline{p}) \multimap (dt :: ct)}$$

Note that, in general, the clock denoted by ct holds larger integers than both ct **on** p and ct **on** $\overline{p}$. This reflects the fact that the output of **merge** p generally converges more than its inputs.

- Rule WHEN is similar to rule MERGE.

$$\textsc{When}\ \frac{p \leq (1)}{\square \vdash \texttt{when}\ p : (dt :: ct) \multimap (dt :: ct\ \mathbf{on}\ p)}$$

Here, in general, the output clock ct **on** p features integers that are smaller than the ones in ct. This reflects the fact that the output of **when** p converges less than its input.

- Rule SUB adds a bounded buffer to an expression.

$$\textsc{Sub}\ \frac{\Gamma \vdash e : t \qquad \vdash t <:_k t'}{\Gamma \vdash e : t'}$$

$$\boxed{\vdash t <:_k t'}$$

$$\textsc{AdaptStream}\ \frac{nf(ct) <:_k nf(ct')}{\vdash dt :: ct <:_k dt :: ct'} \qquad \textsc{AdaptProd}\ \frac{\vdash t_1 <:_k t_1' \qquad \vdash t_2 <:_k t_2'}{\vdash t_1 \otimes t_2 <:_k t_1' \otimes t_2'} \qquad \textsc{AdaptArrow}\ \frac{\vdash t_1' <:_0 t_1 \qquad \vdash t_2 <:_k t_2'}{\vdash t_1 \multimap t_2 <:_k t_1' \multimap t_2'}$$

Figure 3.8: Typing - adaptability judgment

The buffer is modeled by the judgment $\vdash t <:_k t'$ that we have already encountered in the rule for recursive definitions. The type t characterizes the input of the buffer and t' its output.

- Rule RESCALE wraps an expression e in a local time scale driven by the clock type ct.

$$\textsc{Rescale}\ \frac{\vdash \Gamma \downarrow_{ct} \Gamma' \qquad \Gamma' \vdash e : t' \qquad \vdash t' \uparrow_{ct} t}{\Gamma \vdash e : t}$$

One first scatters the variables present in Γ according to ct in order to obtain a context Γ' in the local time scale. This is expressed by the premise $\vdash \Gamma \downarrow_{ct} \Gamma'$. Then, the computation of e happens in Γ', leading to a result of type t' in the local time scale. In order to move this result into the external time scale, one gathers it according to ct, obtaining t. This is expressed by the premise $\vdash t' \uparrow_{ct} t$.

Remark 5. As mentioned in the introduction, previous synchronous functional languages did not check productivity at the level of clock types but as an independent static analysis called *causality analysis* (e.g., Cuoq and Pouzet [2001]). This analysis runs after clock typing. We compare our approach to the usual causality analysis in Section 3.5.

Adaptability The judgment $\vdash t <:_k t'$ expresses that a computation behaving according to type t up to a given time step n can be transformed into a computation that behaves according to type t' up to a time step $n+k$ through the introduction of bounded buffers. Its rules are given in Figure 3.8 and explained below.

- Rule ADAPTSTREAM checks the k−adaptability of clocks by computing the ultimately periodic clocks $nf(ct)$ and $nf(ct')$ corresponding to the clock types ct and ct'. As explained in Chapter 2, on such clocks the k-adaptability relation is decidable. The function $nf(ct)$ reduces a clock type to an ultimately periodic clock.

$$\begin{array}{lcl} nf(p) & = & p \\ nf(ct\ \mathbf{on}\ ct') & = & nf(ct)\ on\ nf(ct') \end{array}$$

- Rule ADAPTPROD is traditional. It introduces buffers on both components.

$\boxed{\vdash t \uparrow_{ct} t'}$ and $\boxed{\vdash t \downarrow_{ct} t'}$

$$\textsc{UpStream}\ \frac{nf(ct') \uparrow_{nf(ct)} nf(ct'')}{\vdash dt :: ct' \uparrow_{ct} dt :: ct''} \qquad \textsc{DownStream}\ \frac{nf(ct') \downarrow_{nf(ct)} nf(ct'')}{\vdash dt :: ct' \downarrow_{ct} dt :: ct''}$$

$$\textsc{UpProd}\ \frac{\vdash t_1 \uparrow_{ct} t_1' \qquad \vdash t_2 \uparrow_{ct} t_2'}{\vdash t_1 \otimes t_2 \uparrow_{ct} t_1' \otimes t_2'} \qquad \textsc{DownProd}\ \frac{\vdash t_1 \downarrow_{ct} t_1' \qquad \vdash t_2 \downarrow_{ct} t_2'}{\vdash t_1 \otimes t_2 \downarrow_{ct} t_1' \otimes t_2'}$$

$$\textsc{UpArrow}\ \frac{\vdash t_1' \downarrow_{ct} t_1 \qquad \vdash t_2 \uparrow_{ct} t_2'}{\vdash t_1 \multimap t_2 \uparrow_{ct} t_1' \multimap t_2'} \qquad \textsc{DownArrow}\ \frac{\vdash t_1' \uparrow_{ct} t_1 \qquad \vdash t_2 \downarrow_{ct} t_2' \qquad ct \leq (1)}{\vdash t_1 \multimap t_2 \downarrow_{ct} t_1' \multimap t_2'}$$

$$\textsc{UpOn}\ \frac{\vdash t \uparrow_{ct'} t'' \qquad \vdash t'' \uparrow_{ct} t'}{\vdash t \uparrow_{ct\,\mathbf{on}\,ct'} t'} \qquad \textsc{DownOn}\ \frac{\vdash t \downarrow_{ct} t'' \qquad \vdash t'' \downarrow_{ct'} t'}{\vdash t \downarrow_{ct\,\mathbf{on}\,ct'} t'}$$

$$\textsc{UpInv}\ \frac{\vdash t \downarrow_{ct'} t' \qquad ct\,\mathbf{on}\,ct' \equiv (1)}{\vdash t \uparrow_{ct} t'} \qquad \textsc{DownInv}\ \frac{\vdash t \uparrow_{ct'} t' \qquad ct\,\mathbf{on}\,ct' \equiv (1)}{\vdash t \downarrow_{ct} t'}$$

$$\textsc{DownCtxEmpty}\ \frac{}{\vdash \Box \downarrow_{ct} \Box} \qquad \textsc{DownCtxCons}\ \frac{\vdash \Gamma \downarrow_{ct} \Gamma' \qquad \vdash t \downarrow_{ct} t'}{\vdash \Gamma, x : t \downarrow_{ct} \Gamma', x : t'}$$

Figure 3.9: Typing - gathering/scattering judgments

- $\boxed{\text{Rule } \textsc{AdaptArrow}}$ is also traditional, except that we have to handle the delay k. Because we seek the most general rule, we want the delay in the conclusion as large as possible and the ones in the premises as small as possible. Thus, the only delay that matters is the one on the output of the function.

Scattering and gathering types Finally, let us describe the judgments related to local time scales. The gathering judgment $\vdash t \uparrow_{ct} t'$ states that all the inhabitants of a type t computed in a local time scale driven by ct can be seen as inhabitants of the type t' in the external time scale. Conversely, the scattering judgment $\vdash t \downarrow_{ct} t'$ states that all the inhabitants of a type t computed in the external time scale can be seen as inhabitants of the type t' in the local time scale driven by ct. Their rules are defined in Figure 3.9 and explained below.

- The first group of rules handle type constructors.
  - $\boxed{\text{Rules } \textsc{UpStream} \text{ and } \textsc{DownStream}}$ use the gathering and scattering relations on clocks defined in Section 2.8. These relations are decidable when the clocks

involved are ultimately periodic, as is the case here.

$$\textsc{UpStream}\ \frac{nf(ct') \uparrow_{nf(ct)} nf(ct'')}{\vdash dt :: ct' \uparrow_{ct} dt :: ct''} \qquad \textsc{DownStream}\ \frac{nf(ct') \downarrow_{nf(ct)} nf(ct'')}{\vdash dt :: ct' \downarrow_{ct} dt :: ct''}$$

– Rules UpProd and DownProd are standard, lifting the components pairwise

$$\textsc{UpProd}\ \frac{\vdash t_1 \uparrow_{ct} t_1' \qquad \vdash t_2 \uparrow_{ct} t_2'}{\vdash t_1 \otimes t_2 \uparrow_{ct} t_1' \otimes t_2'} \qquad \textsc{DownProd}\ \frac{\vdash t_1 \downarrow_{ct} t_1' \qquad \vdash t_2 \downarrow_{ct} t_2'}{\vdash t_1 \otimes t_2 \downarrow_{ct} t_1' \otimes t_2'}$$

– Rule UpArrow and DownArrow make functions leave and enter a local time scale, respectively. The former scatters the input of the function, making it enter the time scale, and gather its output, making it leave the time scale. The latter is symmetric, except that it only applies in time scales driven by binary clocks.

$$\textsc{UpArrow}\ \frac{\vdash t_1' \downarrow_{ct} t_1 \qquad \vdash t_2 \uparrow_{ct} t_2'}{\vdash t_1 \multimap t_2 \uparrow_{ct} t_1' \multimap t_2'} \qquad \textsc{DownArrow}\ \frac{\vdash t_1' \uparrow_{ct} t_1 \qquad \vdash t_2 \downarrow_{ct} t_2' \qquad ct \leq (1)}{\vdash t_1 \multimap t_2 \downarrow_{ct} t_1' \multimap t_2'}$$

We discuss why this rule is unsound without the $ct \leq (1)$ premise in the example section below.

- The second group of rules express that gathering and scattering are compatible with certain algebraic properties of clocks.

  – Rules DownOn and UpOn relate gathering and scattering to clock composition. A time scale driven by a compound clock ct **on** ct' can be decomposed into two time scales. The first one, which is driven by ct', is nested into the second one, which is driven by ct. Thus, for gathering, one leaves the innermost time scale before leaving the outermost one. Conversely, for scattering, one enters the outermost time scale before entering the innermost one.

$$\textsc{UpOn}\ \frac{\vdash t \uparrow_{ct'} t'' \qquad \vdash t'' \uparrow_{ct} t'}{\vdash t \uparrow_{ct\ \mathbf{on}\ ct'} t'} \qquad \textsc{DownOn}\ \frac{\vdash t \downarrow_{ct} t'' \qquad \vdash t'' \downarrow_{ct'} t'}{\vdash t \downarrow_{ct\ \mathbf{on}\ ct'} t'}$$

  – Rules DownInv and UpInv relate clock inverses and the duality between gathering and scattering. They express that gathering by ct is equivalent to scattering by a right inverse ct' of ct, and symmetrically.

$$\textsc{UpInv}\ \frac{\vdash t \downarrow_{ct'} t' \qquad ct\ \mathbf{on}\ ct' \equiv (1)}{\vdash t \uparrow_{ct} t'} \qquad \textsc{DownInv}\ \frac{\vdash t \uparrow_{ct'} t' \qquad ct\ \mathbf{on}\ ct' \equiv (1)}{\vdash t \downarrow_{ct} t'}$$

$$\textsc{Fun}\ \dfrac{\textsc{Var}\ \dfrac{\textsc{ValCtxEmpty}\ \dfrac{}{\vdash \square\ \text{value}}}{\mathtt{x} : (1) \vdash \mathtt{x} : (1)}}{\vdash \mathtt{fun}\,\mathtt{x}.\mathtt{x} : (1) \multimap (1)}$$

(A)

$$\textsc{Fun}\ \dfrac{\textsc{Sub}\ \dfrac{\textsc{Var}\ \dfrac{\textsc{ValCtxEmpty}\ \dfrac{}{\vdash \square\ \text{value}}}{\mathtt{x} : (1) \vdash \mathtt{x} : (1)} \qquad \textsc{AdaptStream}\ \dfrac{(1) <:_1 0(1)}{\vdash (1) <:_1 0(1)}}{\mathtt{x} : (1) \vdash \mathtt{x} : 0(1)}}{\vdash \mathtt{fun}\,\mathtt{x}.\mathtt{x} : (1) \multimap 0(1)}$$

(B)

Figure 3.10: Example 9 - typing the identity function

Examples We omit data types, for clarity, considering that they are all equal.

Example 9 (Identity function). Consider the identity function on streams, *id*, defined below.

$$id \stackrel{\text{def}}{=} \mathtt{fun}\,\mathtt{x}.\mathtt{x}$$

This function may receive several distinct clock types.

- Perhaps the most intuitive clock type is $(1) \multimap (1)$. The corresponding derivation is given in Figure 3.10 (A). More generally, *id* might receive any clock type of the form $p \multimap p$, with p a fixed ultimately periodic word.
- One may add a buffer between input and output, conceptually "shifting" the outputs to the right in the ambient time scale. A simple example would be the clock type $(1) \multimap 0(1)$, with the corresponding derivation given in Figure 3.10 (B). Compared to the clock type in Figure 3.10 (A), here the output has been delayed for one step.

Example 10 (Linearity). Consider the typing derivations given in Figure 3.11.

- Derivation (a) types $\mathtt{fun}\,\mathtt{x}.\mathtt{fun}\,\mathtt{f}.\mathtt{f}\,\mathtt{x}$. The context splitting judgment sends $\mathtt{x}$ to the left premise, which is the function, and $\mathtt{f}$ to the right one, which is the argument. No contraction is needed in this example, as $\mathtt{x}$ occurs only once in the body of the function.
- Derivation (b) types the same expression but differs from (a). It features a useless contraction, making x appear in the context where the body of the function applied is typed. But since $\mathtt{x}$ denotes a stream, which is a value, it can be erased in the VAR rule.
- Derivation (c) types $\mathtt{fun}\,\mathtt{x}.\mathtt{fun}\,\mathtt{f}.(\mathtt{f}\,\mathtt{x},\mathtt{x})$. In this case, $\mathtt{x}$ appears twice and thus contraction is actually needed. The sub-derivation of $\mathtt{f} : (1) \multimap (2), \mathtt{x} : (1) \vdash \mathtt{f}\,\mathtt{x} : (2)$ has been elided since it is similar to the one in derivation (a).

Example 11 (Sampling). Consider the function *half* which filters out the elements at even indices from its input stream.

$$half \stackrel{\text{def}}{=} \mathtt{fun}\,\mathtt{x}.\mathtt{when}\,(1\ 0)\,\mathtt{x}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\square \vdash \square \otimes \square}\,\textsc{SepEmpty}}{\mathtt{x}:(1) \vdash \square \otimes \mathtt{x}:(1)}\,\textsc{SepRight}}{\mathtt{x}:(1), \mathtt{f}:(1) \multimap (2) \vdash \mathtt{f}:(1) \multimap (2) \otimes \mathtt{x}:(1)}\,\textsc{SepLeft} \quad \dfrac{\dfrac{}{\vdash \square \text{ value}}\,\textsc{ValCtxEmpty}}{\mathtt{f}:(1) \multimap (2) \vdash \mathtt{f}:(1) \multimap (2)}\,\textsc{Var} \quad \dfrac{\dfrac{}{\vdash \square \text{ value}}\,\textsc{ValEmpty}}{\mathtt{x}:(1) \vdash \mathtt{x}:(1)}\,\textsc{Var}}{\mathtt{x}:(1), \mathtt{f}:(1) \multimap (2) \vdash \mathtt{f}\ \mathtt{x}:(2)}\,\textsc{App}$$
$$\dfrac{\dfrac{\mathtt{x}:(1), \mathtt{f}:(1) \multimap (2) \vdash \mathtt{f}\ \mathtt{x}:(2)}{\mathtt{x}:(1) \vdash \mathtt{fun\,f.f\ x}:((1) \multimap (2)) \multimap (2)}\,\textsc{Fun}}{\vdash \mathtt{fun\,x.fun\,f.f\ x}:(1) \multimap ((1) \multimap (2)) \multimap (2)}\,\textsc{Fun}$$

(a) - Typing derivation for `fun x. fun f. f x`

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\square \vdash \square \otimes \square}\,\textsc{SepEmpty}}{\mathtt{x}:(1) \vdash \mathtt{x}:(1) \otimes \mathtt{x}:(1)}\,\textsc{SepContract}}{\mathtt{x}:(1), \mathtt{f}:(1) \multimap (2) \vdash \mathtt{x}:(1), \mathtt{f}:(1) \multimap (2) \otimes \mathtt{x}:(1)}\,\textsc{SepLeft} \quad \dfrac{\dfrac{\dfrac{}{\vdash \square \text{ value}}\,\textsc{ValCtxEmpty} \quad \dfrac{}{\vdash (1) \text{ value}}\,\textsc{ValStream}}{\vdash x:(1) \text{ value}}\,\textsc{ValCtxCons}}{\mathtt{x}:(1), \mathtt{f}:(1) \multimap (2) \vdash \mathtt{f}:(1) \multimap (2)}\,\textsc{Var} \quad \dfrac{\dfrac{}{\vdash \square \text{ value}}\,\textsc{ValEmpty}}{\mathtt{x}:(1) \vdash \mathtt{x}:(1)}\,\textsc{Var}}{\mathtt{x}:(1), \mathtt{f}:(1) \multimap (2) \vdash \mathtt{f}\ \mathtt{x}:(2)}\,\textsc{App}$$
$$\dfrac{\dfrac{\mathtt{x}:(1), \mathtt{f}:(1) \multimap (2) \vdash \mathtt{f}\ \mathtt{x}:(2)}{\mathtt{x}:(1) \vdash \mathtt{fun\,f.f\ x}:((1) \multimap (2)) \multimap (2)}\,\textsc{Fun}}{\vdash \mathtt{fun\,x.fun\,f.f\ x}:(1) \multimap ((1) \multimap (2)) \multimap (2)}\,\textsc{Fun}$$

(b) - Typing derivation for `fun x. fun f. f x`, featuring useless contraction

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\square \vdash \square \otimes \square}\,\textsc{SepLeft}}{\mathtt{f}:(1) \multimap (2) \vdash \mathtt{f}:(1) \multimap (2) \otimes \square}\,\textsc{SepEmpty} \quad \dfrac{}{\vdash (1) \text{ value}}\,\textsc{ValStream}}{\mathtt{f}:(1) \multimap (2), \mathtt{x}:(1) \vdash \mathtt{f}:(1) \multimap (2), \mathtt{x}:(1) \otimes \mathtt{x}:(1)}\,\textsc{SepContract} \quad \dfrac{\ldots}{\mathtt{f}:(1) \multimap (2), \mathtt{x}:(1) \vdash \mathtt{f\ x}:(2)}\,\textsc{App} \quad \dfrac{\dfrac{}{\vdash \square \text{ value}}\,\textsc{ValCtxEmpty}}{x:(1) \vdash x:(1)}\,\textsc{Var}}{\mathtt{f}:(1) \multimap (2), \mathtt{x}:(1) \vdash (\mathtt{f\ x}, \mathtt{x}):(2) \otimes (1)}\,\textsc{Pair}$$
$$\dfrac{\dfrac{\mathtt{f}:(1) \multimap (2), \mathtt{x}:(1) \vdash (\mathtt{f\ x}, \mathtt{x}):(2) \otimes (1)}{\mathtt{f}:(1) \multimap (2) \vdash \mathtt{fun\,x.}\,(\mathtt{f\ x}, \mathtt{x}):(1) \multimap (2) \otimes (1)}\,\textsc{Fun}}{\vdash \mathtt{fun\,f.fun\,x.}\,(\mathtt{f\ x}, \mathtt{x}):((1) \multimap (2)) \multimap (1) \multimap (2) \otimes (1)}\,\textsc{Fun}$$

(c) - Typing derivation for `fun f. fun x. (f x, x)`

Figure 3.11: Example 10 - typing derivations

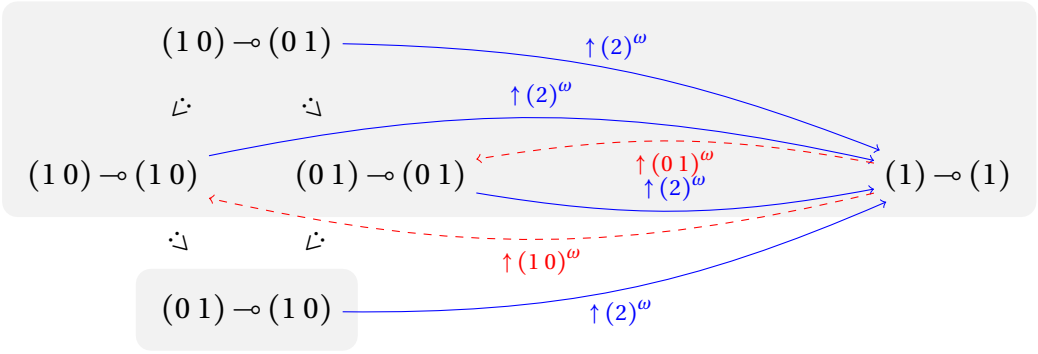


Figure 3.12: Example 13 - gathering, adaptability, and equivalence classes

1. A valid clock type is $(1) \multimap (1\ 0)$. According to this binary clock type, the function consumes one element per time step but only produces at even time steps. A corresponding derivation is shown in Figure 3.13 (A). In the instance of the WHEN rule, we have reasoned up to the equivalence of the clock types (1) **on** $(1\ 0)$ and $(1\ 0)$.

2. Another possibility is $(2) \multimap (1\ 0)$. Here the function produces one element of the output stream per time step, at the price of needing two input elements. A possible derivation for this type is shown in Figure 3.13 (B). It is essentially the same as Figure 3.13 (A), the only difference being that the output clock type is (2) **on** $(1\ 0)$ which is equivalent to (1).

3. Figure 3.13 (C) gives a conceptually different derivation of $\vdash \mathit{half} : (2) \multimap (1)$. In contrast with the derivation (B), in (C) we rescale the derivation (A) by (2) to obtain the desired clock type from $\vdash \mathit{half} : (1) \multimap (1\ 0)$.

Remark 6. Rescaling, like buffering, is generic in the sense that it does not need to peer inside the body of the rescaled (or buffered) expression. Example 11 (3) shows that $(1) \multimap (1\ 0)$ is *at least as general* as $(2) \multimap (1)$ since one can go from the former to the latter in such a generic manner. It is in fact *strictly more general* because one cannot transform $(2) \multimap (1)$ into $(1) \multimap (1\ 0)$ via rescaling or buffering alone. This comes from the fact that $(2) \multimap (1)$ describes computations in which the first element of the output stream might depend on the first element of the input stream, while in $(1) \multimap (1\ 0)$ this can never be the case.

Remark 7. Chapter 4 will make clear that the typing derivations (B) and (C) from Figure 3.13 compile to different circuits. Intuitively, the derivation (C) leads to a "large" circuit, since it contains two copies of the circuit obtained by compiling the derivation (A).

Example 12 (Buffering and Integer Clocks). Consider the function *sumhalf* defined below.

$$\mathit{sumhalf} \stackrel{\text{def}}{=} \texttt{fun x.+(when (1 0) x, when (0 1) x)}$$

Informally, the element y_n in the output stream is the sum of x_{2n} and x_{2n+1} in the input stream.

$$\textsc{App}\ \frac{\dfrac{\dfrac{\dots}{\mathtt{x}:(1)\vdash \square\otimes \mathtt{x}:(1)}\quad \textsc{When}\ \dfrac{\vdash \mathtt{when}\,(1\,0):(1)\multimap(1\,0)}{\mathtt{x}:(1)\vdash \mathtt{when}\,(1\,0)\,\mathtt{x}:(1\,0)}\quad \textsc{Var}\ \dfrac{\vdash \square\ \mathrm{value}}{\mathtt{x}:(1)\vdash \mathtt{x}:(1)}}{\textsc{Fun}\ \dfrac{\mathtt{x}:(1)\vdash \mathtt{when}\,(1\,0)\,\mathtt{x}:(1\,0)}{\vdash \mathtt{fun}\,\mathtt{x}.\,\mathtt{when}\,(1\,0)\,\mathtt{x}:(1)\multimap(1\,0)}}}{\vdash \mathtt{fun}\,\mathtt{x}.\,\mathtt{when}\,(1\,0)\,\mathtt{x}:(2)\multimap(1)}$$

(A)

$$\textsc{App}\ \frac{\dfrac{\dfrac{\dots}{\mathtt{x}:(2)\vdash \square\otimes \mathtt{x}:(2)}\quad \textsc{When}\ \dfrac{\vdash \mathtt{when}\,(1\,0):(2)\multimap(1)}{\mathtt{x}:(2)\vdash \mathtt{when}\,(1\,0)\,\mathtt{x}:(1)}\quad \textsc{Var}\ \dfrac{\vdash \square\ \mathrm{value}}{\mathtt{x}:(2)\vdash \mathtt{x}:(2)}}{\textsc{Fun}\ \dfrac{\mathtt{x}:(2)\vdash \mathtt{when}\,(1\,0)\,\mathtt{x}:(1)}{\vdash \mathtt{fun}\,\mathtt{x}.\,\mathtt{when}\,(1\,0)\,\mathtt{x}:(2)\multimap(1)}}}{\vdash \mathtt{fun}\,\mathtt{x}.\,\mathtt{when}\,(1\,0)\,\mathtt{x}:(2)\multimap(1)}$$

(B)

$$\textsc{Rescale}\ \frac{\dfrac{\vdash \square\downarrow_{(2)}\square}{\vdash \mathtt{fun}\,\mathtt{x}.\,\mathtt{when}\,(1\,0)\,\mathtt{x}:(1)\multimap(1\,0)}\ (\mathrm{A})\qquad \textsc{DownStream}\ \dfrac{\textsc{DownStream}\ \dfrac{(2)^{\omega}=(2)^{\omega}\ on\ (1)^{\omega}}{\vdash (2)\downarrow_{(2)}(1)}\quad \textsc{UpArrow}\ \dfrac{(2)^{\omega}\ on\ (1\,0)^{\omega}=(1)^{\omega}}{\vdash (1\,0)\uparrow_{(2)}(1)}}{\vdash (1)\multimap(1\,0)\uparrow_{(2)}(2)\multimap(1)}}{\vdash \mathtt{fun}\,\mathtt{x}.\,\mathtt{when}\,(1\,0)\,\mathtt{x}:(2)\multimap(1)}$$

(C)

Figure 3.13: Example 11 - typing derivations

$$\dfrac{\dfrac{\dfrac{}{\mathtt{x}:(1)\vdash \square\otimes\mathtt{x}:(1)}\ \ldots}{\mathtt{x}:(1)\vdash +:(0\,1)\otimes(0\,1)\multimap(0\,1)}\ \ldots \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{x}:(1)\vdash \mathtt{x}:(1)\otimes\mathtt{x}:(1)}\ \ldots}{\mathtt{x}:(1)\vdash \mathtt{when}\,(1\,0)\,\mathtt{x}:(1\,0)}\ \ldots \qquad \dfrac{}{\vdash (1\,0)\lhd_1(0\,1)}\ \ldots}{\mathtt{x}:(1)\vdash \mathtt{when}\,(1\,0)\,\mathtt{x}:(0\,1)}\ \ldots}{\mathtt{x}:(1)\vdash(\mathtt{when}\,(1\,0)\,\mathtt{x},\,\mathtt{when}\,(0\,1)\,\mathtt{x}):(0\,1)\otimes(0\,1)}}{\mathtt{x}:(1)\vdash +(\mathtt{when}\,(1\,0)\,\mathtt{x},\,\mathtt{when}\,(0\,1)\,\mathtt{x}):(0\,1)}$$
$$\vdash \mathtt{fun}\,\mathtt{x}.+(\mathtt{when}\,(1\,0)\,\mathtt{x},\,\mathtt{when}\,(0\,1)\,\mathtt{x}):(1)\multimap(0\,1)$$

(A)

$$\dfrac{\dfrac{\dfrac{}{\mathtt{x}:(2)\vdash \square\otimes\mathtt{x}:(2)}\ \ldots}{\vdash +:(1)\otimes(1)\multimap(1)}\ \ldots \qquad \dfrac{\dfrac{\dfrac{}{\mathtt{x}:(2)\vdash \mathtt{x}:(2)\otimes\mathtt{x}:(2)}\ \ldots}{\mathtt{x}:(2)\vdash \mathtt{when}\,(1\,0)\,\mathtt{x}:(1)}\ \ldots \qquad \dfrac{}{\mathtt{x}:(2)\vdash \mathtt{when}\,(0\,1)\,\mathtt{x}:(1)}\ \ldots}{\mathtt{x}:(2)\vdash(\mathtt{when}\,(1\,0)\,\mathtt{x},\,\mathtt{when}\,(0\,1)\,\mathtt{x}):(1)\otimes(1)}}{\mathtt{x}:(2)\vdash +(\mathtt{when}\,(1)\,\mathtt{x},\,\mathtt{when}\,(1)\,\mathtt{x}):(1)}$$
$$\vdash \mathtt{fun}\,\mathtt{x}.+(\mathtt{when}\,(1\,0)\,\mathtt{x},\,\mathtt{when}\,(0\,1)\,\mathtt{x}):(2)\multimap(1)$$

(B)

Figure 3.14: Example 12 - typing derivations

1. A possible clock type is $(1) \multimap (0\ 1)$. Figure 3.14 (A) gives a possible derivation, in which rule names and easily guessable subderivations have been elided. This derivation buffers the output of `when` $(1\ 0)$ for one step, using the adaptability judgment $\vdash (1\ 0) <:_1 (0\ 1)$. A value cannot be produced at the first time step since the second input has not been received yet. Thus, all the derivations for this clock type necessarily involve such a buffering process.

2. Another possibility is $(2) \multimap (1)$. Figure 3.14 (B) gives a derivation, which works in the same way as Figure 3.13 (B). No buffering is involved here since, at each time step, the function has exactly the data needed to produce its next output.

This example shows how different choices of clock types may influence the amount of memory needed in a typed program.

Example 13 (Gathering). We now show the results of gathering for four first-order functions, with respect to a local time scale where time passes twice faster than in the global context.

$$\begin{array}{llll} (1\ 0) \multimap (0\ 1) & \uparrow_{(2)} & (1) \multimap (1) & (a) \\ (1\ 0) \multimap (1\ 0) & \uparrow_{(2)} & (1) \multimap (1) & (b) \\ (0\ 1) \multimap (0\ 1) & \uparrow_{(2)} & (1) \multimap (1) & (c) \\ (0\ 1) \multimap (1\ 0) & \uparrow_{(2)} & (1) \multimap (1) & (d) \end{array}$$

These examples show that, in general, gathering by an integer clock hides implementation details, transforming complex types into simpler ones. In contrast, gathering by a binary clock only adds zeroes for the time steps at which the time scale is idle. This is what the example below shows.

$$\begin{array}{llll} (1) \multimap (1) & \uparrow_{(1\ 0)} & (1\ 0) \multimap (1\ 0) & (e) \\ (1) \multimap (1) & \uparrow_{(0\ 1)} & (0\ 1) \multimap (0\ 1) & (f) \end{array}$$

The gathering relations (e) and (f) above invert the (b) and (c) ones.

Remark 8. The types and gathering relations of Example 13 are represented in Figure 3.12. This figure also depicts the adaptability relations that exist between these types. The combination of gathering adaptability and gathering defines a preorder whose equivalence classes are represented in light grey. Remark that $(0\ 1) \multimap (1\ 0)$ is, here, the *most general* type: the five other types can be obtained from it, but not reciprocally. Indeed, its inhabitants have the lowest amount of dependencies between inputs and outputs.

Example 14 (Unsound DOWNARROW). The DOWNARROW rule in Figure 3.9 is restricted to local time scales driven by binary clocks. We may now explain why this hypothesis is needed. Consider the scattering relations shown in Example 13. If scattering arrow types by non-binary clocks is allowed, we can reverse them all.

$$\begin{array}{llll} (1) \multimap (1) & \downarrow_{(2)} & (1\ 0) \multimap (0\ 1) & (A) \\ (1) \multimap (1) & \downarrow_{(2)} & (1\ 0) \multimap (1\ 0) & (B) \\ (1) \multimap (1) & \downarrow_{(2)} & (0\ 1) \multimap (0\ 1) & (C) \\ (1) \multimap (1) & \downarrow_{(2)} & (0\ 1) \multimap (1\ 0) & (D) \end{array}$$

$$\textsc{DownArrow}\ \dfrac{\textsc{UpStream}\ \dfrac{}{\vdash (1)\uparrow_{(1\,0\,1)}(1\,0\,1)} \qquad \textsc{DownStream}\ \dfrac{}{\vdash (2\,0\,1)\downarrow_{(1\,0\,1)}(2\,1)}}{\vdash (1\,0\,1)\multimap(2\,0\,1)\downarrow_{(1\,0\,1)}(1)\multimap(2\,1)}$$

(A)

$$\textsc{DownInv}\ \dfrac{\textsc{UpArrow}\ \dfrac{\textsc{DownStream}\ \dfrac{}{\vdash (0\,1)\downarrow_{(0\,1)}(1)} \qquad \textsc{UpStream}\ \dfrac{}{\vdash (2\,1)\uparrow_{(0\,1)}(0\,2\,0\,1)}}{\vdash (1)\multimap(2\,1)\uparrow_{(0\,1)}(0\,1)\multimap(0\,2\,0\,1)} \qquad (3\,1)\ \mathbf{on}\ (1\,0\,0\,1)\equiv(1)}{\vdash (1)\multimap(2\,1)\downarrow_{(3\,1)}(0\,1)\multimap(0\,2\,0\,1)}$$

(B)

$$\textsc{DownOn}\ \dfrac{\dfrac{(A)}{\vdash (1\,0\,1)\multimap(2\,0\,1)\downarrow_{(1\,0\,1)}(1)\multimap(2\,1)} \qquad \dfrac{(B)}{\vdash (1)\multimap(2\,1)\downarrow_{(3\,1)}(0\,1)\multimap(0\,2\,0\,1)}}{\vdash (1\,0\,1)\multimap(2\,0\,1)\downarrow_{(1\,0\,1)\ \mathbf{on}\ (3\,1)}(0\,1)\multimap(0\,2\,0\,1)}$$

(C)

Figure 3.15: Example 15 - gathering by $(3\,0\,1)$

Unfortunately, the relation (D) is incorrect. Take for example the identity function. One may assign it the type $(1)\multimap(1)$, but never $(0\,1)\multimap(1\,0)$, since the latter classifies functions whose first output never depends on the first input.

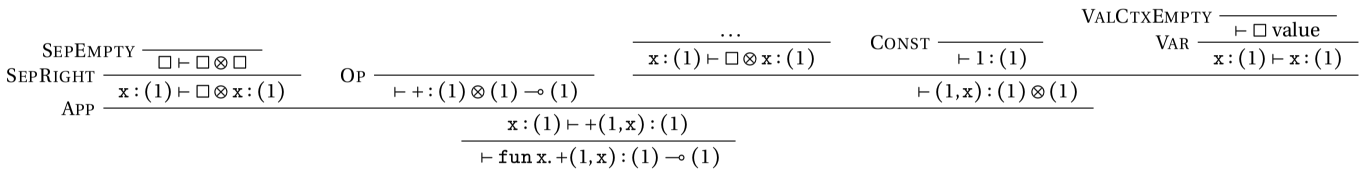
Remark 9. The fact that one cannot transform the type $(1)\multimap(1)$ into $(0\,1)\multimap(1\,0)$ is consistent with Figure 3.12 and Remark 8: the latter is strictly more general than the former.

Example 15. The previous example suggests that one cannot scatter a function by a local time scale containing integers larger than one. While there is no primitive rule to do so, this can actually be achieved by combining several rules and using algebraic properties of clocks.

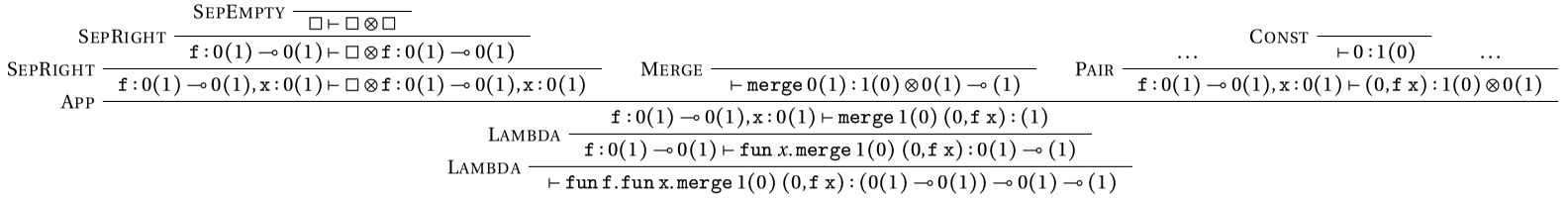
1. We know by Property 13 that any clock factors (uniquely) as $w_b\ on\ w_p$, with w_b binary and w_p strictly positive. Thus, factoring the clock type and applying rule DownOn cuts the function scattering problem in two.

2. We already have a rule for scattering functions by binary clocks. For strictly positive ones, Property 12 states that they have right-inverses. This means that we can apply rule DownInv to replace scattering with gathering. Gathering functions is simple.

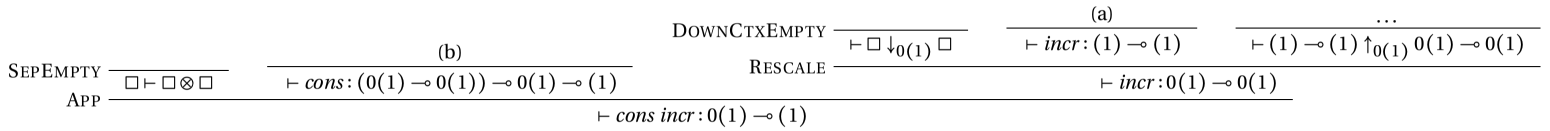
Figure 3.15 gives an example using the decomposition given above. The function type is scattered by $(3\,0\,1)$, which factors as $(1\,0\,1)$ **on** $(3\,1)$. Derivation (A) first gathers the function by $(1\,0\,1)$. The resulting type is then gathered by $(3\,1)$ in derivation (B) using its inverse $((0\,1))^\omega$. Derivation (C) combines (A) and (B) to finally gather by $(3\,0\,1)$.



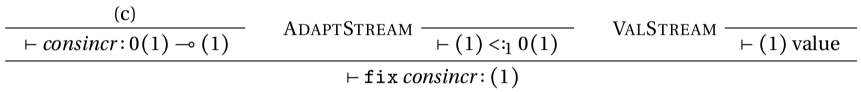
(a) - Derivation for *incr* = `fun x. +(1, x)`



(b) - Derivation for *cons* = `fun f. fun x. merge 1(0) (0, f x)`



(c) - Derivation for *consincr* = *cons incr*



(d) - Derivation for *nat* = `fix` *consincr*

Figure 3.16: Example 16 - typing derivations

Example 16 (Natural numbers). In this example we implement the classic synchronous program *nat*, which denotes the stream of natural numbers. We do it in an incremental way in order to illustrate some features of the type system. The relevant typing derivations are given in Figure 3.16—this time, the elided data types are all equal to **int**. Additionally, we do not repeat identical or equivalent sub-derivations.

1. Consider the expression *incr* below.

$$incr \stackrel{\text{def}}{=} \texttt{fun x.+}(1,\texttt{x})$$

   This expression denotes a stream function which increments all the elements of its input stream by one. A possible type for this function is $(1) \multimap (1)$, as shown by the derivation given in Figure 3.16 (a). This derivations shows a simple program with constants and the application of an operator, here +.

2. The expression *cons* defined below denotes a higher-order function.

$$cons \stackrel{\text{def}}{=} \texttt{fun f.fun x.merge}\ 1(0)\ (0,\texttt{f x})$$

   This function receives a function `f` and a stream `x`. It applies `f` to `x` and adds a zero in front of the resulting stream. One may give this function the type $(0(1) \multimap 0(1)) \multimap 0(1) \multimap (1)$ with the corresponding derivation given in Figure 3.16 (b).

3. The expression *consincr* is the application of *cons* to *incr*.

$$consincr \stackrel{\text{def}}{=} cons\ incr$$

   The function *incr* is of type $(1) \multimap (1)$, but should be of type $0(1) \multimap 0(1)$ to match the type of *cons*. This mismatch is solved through the use of a binary local time scale which skips the first time step. This gives *cons* the proper type, as shown in Figure 3.16 (c).

4. Finally, the stream *nat* can be obtained as the fixpoint of *consincr*.

$$consincr \stackrel{\text{def}}{=} \texttt{fix}\ consincr$$

   Since the output of this function does not depend instantaneously on its input, this is safe. The derivation Figure 3.16 (d) shows the corresponding adaptability premise.

Example 17 (Composition). Suppose we are given derivations for two closed expressions that denote functions. Furthermore, suppose that the first function, f, has type $(2) \multimap (2)$, while the second function, g, has type $(3) \multimap (3)$. We discuss two ways in which f and g can be composed using local time scales and, if needed, buffering. In this example, we elide rule names and separation judgments in order to keep the size of derivations managable.

1. Figure 3.17 describes an approach where we use local time steps to assign to both f and g the type $(6) \multimap (6)$. To reach this type, f and g have to perform respectively three and two local steps per global step. No buffering is needed, but in a circuit implementation we have to replicate the bodies of f and g respectively three and two times.

$$\dfrac{\dfrac{}{\vdash \square \downarrow_{(2)} \square} \quad \dfrac{\ldots}{\vdash g : (3) \multimap (3)} \quad \dfrac{\dfrac{}{\vdash (6) \downarrow_{(2)} (3)} \quad \dfrac{}{\vdash (3) \uparrow_{(2)} (6)}}{\vdash (3) \multimap (3) \uparrow_{(2)} (6) \multimap (6)}}{\vdash g : (6) \multimap (6)}$$

(A) - gathering g by $(2)^\omega$

$$\dfrac{\dfrac{}{\vdash \square \downarrow_{(2)} \square} \quad \dfrac{\ldots}{\vdash f : (2) \multimap (2)} \quad \dfrac{\dfrac{}{\vdash (6) \downarrow_{(3)} (2)} \quad \dfrac{}{\vdash (2) \uparrow_{(3)} (6)}}{\vdash (2) \multimap (2) \uparrow_{(3)} (6) \multimap (6)}}{\vdash f : (6) \multimap (6)}$$

(B) - gathering f by $(3)^\omega$

$$\dfrac{\ldots \quad \dfrac{\dfrac{\text{(A)}}{\vdash g : (6) \multimap (6)} \quad \ldots \quad \dfrac{\dfrac{\text{(B)}}{\vdash f : (6) \multimap (6)} \quad \dfrac{\vdash \square \text{ value}}{\mathtt{x} : (6) \vdash \mathtt{x} : (6)}}{\mathtt{x} : (6) \vdash f\ x : (6)}}{\mathtt{x} : (6) \vdash g\ (f\ x) : (6)}}{\vdash \mathtt{fun}\ x.\, g\ (f\ x) : (6) \multimap (6)}$$

(C) - composing f and g

Figure 3.17: Example 17 - first derivation

2. Now, if our functions contain expensive operators, such as multipliers, this may be too costly. The derivation in Figure 3.18 shows a way to work around this problem. We use local time scales again, but this time to slow down the functions. The function f becomes twice as slow, now inhabiting $(2\ 0) \multimap (2\ 0)$, and g becomes three times as slow, now inhabiting $(0\ 0\ 3) \multimap (0\ 0\ 3)$. The gain is that the output of f is now adaptable to the input of g via a buffer of size two, since $\vdash (2\ 0) <:_0 (0\ 0\ 3)$. This buffer is periodically empty: at every sixth step out of six all the stored data has been consumed.

Remark 10. We may take the derivation Figure 3.18 (C) and wrap it into a local time scale driven by $(6)^\omega$. The gathering judgment below shows the type that is obtained through the RESCALE rule.

$$\vdash (2\ 0) \multimap (0\ 0\ 3) \uparrow_{(6)} (6) \multimap (6)$$

This is the same type as in Figure 3.17 (C). The underlying derivation still contains the adaptability judgment introduced in Figure 3.18 (C), and the compiled code would thus contain a buffer. But this buffer is now virtually useless, as it is always empty after each step.

$$\dfrac{\dfrac{}{\vdash \Box \downarrow_{(0\,0\,1)} \Box} \quad \dfrac{\ldots}{\vdash g : (3) \multimap (3)} \quad \dfrac{\dfrac{}{\vdash (0\,0\,3) \downarrow_{(0\,0\,1)} (3)} \quad \dfrac{}{\vdash (3) \uparrow_{(0\,0\,1)} (0\,0\,3)}}{\vdash (3) \multimap (3) \uparrow_{(0\,0\,1)} (0\,0\,3) \multimap (0\,0\,3)}}{\vdash g : (0\,0\,3) \multimap (0\,0\,3)}$$

(A) - gathering g by $(0\,0\,1)^\omega$

$$\dfrac{\dfrac{}{\vdash \Box \downarrow_{(2)} \Box} \quad \dfrac{\ldots}{\vdash f : (2) \multimap (2)} \quad \dfrac{\dfrac{}{\vdash (2\,0) \downarrow_{(1\,0)} (2)} \quad \dfrac{}{\vdash (2) \uparrow_{(1\,0)} (2\,0)}}{\vdash (2) \multimap (2) \uparrow_{(1\,0)} (2\,0) \multimap (2\,0)}}{\vdash f : (2\,0) \multimap (2\,0)}$$

(B) - gathering f by $(1\,0)^\omega$

$$\dfrac{\ldots \quad \dfrac{\dfrac{\ldots}{\vdash g : (0\,0\,3) \multimap (0\,0\,3)}\,(A) \quad \dfrac{\dfrac{\ldots}{\vdash f : (2\,0) \multimap (2\,0)}\,(B) \quad \dfrac{}{x : (2\,0) \vdash x : (2\,0)\ \text{value}}}{x : (2\,0) \vdash f\ x : (2\,0)} \quad \dfrac{}{\vdash (2\,0) <:_0 (0\,0\,3)}}{x : (2\,0) \vdash f\ x : (0\,0\,3)}}{x : (2\,0) \vdash g\ (f\ x)) : (0\,0\,3)}$$
$$\vdash \mathtt{fun}\, x.\, g\ (f\ x) : (2\,0) \multimap (0\,0\,3)$$

(C) - composing f and g

Figure 3.18: Example 17 - second derivation

3.3 Typed Semantics

The semantics defined in Section 3.1 applies to all programs, typed or not. This semantics gives meaning to programs of dubious behavior. We now define a *typed semantics* based on the domain of clocked streams introduced in Chapter 2. This typed semantics can be seen as a variant of the untyped semantics annotated with additional information; it models the fact that every Kahn network corresponding to a well-typed term can be *synchronized* by putting buffers on its inputs and outputs. We will then show that in such well-typed programs, all clocked streams are total.

From a technical point of view, we achieve this by defining a *type directed* interpretation function mapping typing derivations to their meanings. The untyped semantics relied on a universal domain $\mathbb{K}$ in which all expressions are interpreted, regardless of their types. In contrast, in the typed expression, a closed expression of type t is interpreted as an element of the domain $\mathbf{S}[\![t]\!]$. This domain is defined by induction for t.

The typed semantics relies on four families of interpretation functions, all written $\mathbf{S}[\![\_]\!]$. First, we map clock types to clocks. Then, we define the interpretation of types and contexts; this mainly consists in mapping stream types to the domain of clocked streams. Once this is done, we can express the semantics of typing derivation $\Gamma \vdash e : t$ as a dependently-typed

map from the interpretation of Γ to the one of t. This function relies on an interpretation of the adaptability, scattering and gathering judgments as coercions between their source and destination types.

3.3.1 Types and Synchronization

Let us begin with the interpretation of types, and its relation with the universal domain $\mathbb{K}$. For any t, the domain $\mathbf{S}[\![t]\!]$ is a retract of $\mathbb{K}$. In other words, $\mathbf{S}[\![t]\!]$ is "smaller" than $\mathbb{K}$ in the sense that there exists an injection This makes it possible to compare the typed and untyped semantics, and move between the two.

Interpreting types and contexts Any clock type ct denotes exactly one clock $\mathbf{S}[\![ct]\!]$.

$$\begin{array}{lcl} \mathbf{S}[\![ct]\!] & \in & \mathbf{Ck} \\ \mathbf{S}[\![u(v)]\!] & = & u(v)^\omega \\ \mathbf{S}[\![ct\ \mathbf{on}\ ct']\!] & = & \mathbf{S}[\![ct]\!]\ on\ \mathbf{S}[\![ct']\!] \end{array}$$

We can now define the interpretation of types. A stream type $dt :: ct$ is interpreted as the domain $CStream_{\mathbf{S}[\![ct]\!]}([\![dt]\!])$ of clocked stream, following Definition 11. Other type constructors are interpreted in a standard way.

$$\begin{array}{lcl} \mathbf{S}[\![dt :: ct]\!] & = & CStream_{\mathbf{S}[\![ct]\!]}([\![dt]\!]) \\ \mathbf{S}[\![t_1 \otimes t_2]\!] & = & \mathbf{S}[\![t_1]\!] \times \mathbf{S}[\![t_2]\!] \\ \mathbf{S}[\![t_1 \multimap t_2]\!] & = & \mathbf{S}[\![t_1]\!] \Rightarrow_c \mathbf{S}[\![t_2]\!] \end{array}$$

The interpretation of contexts is straightforward: they are conceptually (nested) pairs of types, and thus correspond to products.

$$\begin{array}{lcl} \mathbf{S}[\![\square]\!] & = & \varnothing_\perp \\ \mathbf{S}[\![\Gamma, x : t]\!] & = & \mathbf{S}[\![\Gamma]\!] \times \mathbf{S}[\![t]\!] \end{array}$$

The interpretation of a typing $\Gamma \vdash t$ is a function from Γ to t.

$$\mathbf{S}[\![\Gamma \vdash t]\!] = \mathbf{S}[\![\Gamma]\!] \Rightarrow_c \mathbf{S}[\![t]\!]$$

Remark 11. The domain $\varnothing_\perp$, which serves as the interpretation for empty context, is the "smallest" domain (up to isomorphism), in the sense that there is a unique continuous function from this domain to any other one. Because of this, for any domain D we have the isomorphism $(\varnothing_\perp \Rightarrow_c D) \cong D$, and thus the definition above gives $\mathbf{S}[\![\square \vdash t]\!] \cong \mathbf{S}[\![t]\!]$. Thus, we sometimes identify the two domains and treat an element of $\square \vdash t$ as one of t.

Synchronizing types Before defining the semantics of typed programs, we need to establish that the domain interpretation of any type t can be injected into the universal domain $\mathbb{K}$. We already have our main ingredient, given in Section 2.9 and in particular by Property 16: the domain of clocked streams is a retract of the domain of streams, which itself is a retract of $\mathbb{K}$.

Thus we only have to show that e-p pairs compose and can be lifted to products and functions. The definitions below give these constructions.

$$\begin{array}{lcl}(B \lhd C) \circ (A \lhd B) & \in & A \lhd C \\ (e_2, p_2) \circ (e_1, p_1) & = & (e_2 \circ e_1, p_1 \circ p_2)\end{array}$$

$$\begin{array}{lcl}(A \lhd A') \times (B \lhd B') & \in & (A \times B) \lhd (A' \times B') \\ (e_1, p_1) \times (e_2, p_2) & = & (e_1 \times e_2, p_1 \times p_2)\end{array}$$

$$\begin{array}{lcl}(A \lhd A') \Rightarrow_c (B \lhd B') & \in & (A \Rightarrow_c B) \lhd (A' \Rightarrow_c B') \\ (e_1, p_1) \Rightarrow_c (e_2, p_2) & = & (\lambda f.p_2 \circ f \circ e_1, \lambda f'.e_2 \circ f' \circ p_1)\end{array}$$

These combinators make it possible to see the interpretations of all the types of μAS as retracts of the universal domain. We call the corresponding family of e-p pairs $(desync_t, sync_t)$, and define it as follows.

$$\begin{array}{lcl}(desync_t, sync_t) & \in & \mathbf{S}[\![t]\!] \lhd \mathbb{K} \\ (desync_{dt :: ct}, sync_{dt :: ct}) & = & (\mathbf{unpack}, \mathbf{pack}_{\mathbf{S}[\![ct]\!]}) \circ (unstream, stream) \\ (desync_{t_1 \otimes t_2}, sync_{t_1 \otimes t_2}) & = & ((desync_{t_1}, sync_{t_1}) \times (desync_{t_2}, sync_{t_2})) \circ (unprod, prod) \\ (desync_{t_1 \multimap t_2}, sync_{t_1 \multimap t_2}) & = & ((desync_{t_1}, sync_{t_1}) \Rightarrow (desync_{t_2}, sync_{t_2})) \circ (unfun, fun)\end{array}$$

Synchronizing contexts Now, we would like to have a family $(desync_\Gamma, sync_\Gamma)$ to move back-and-forth between the domains $\mathbf{S}[\![\Gamma]\!]$ and $Env(\mathbb{K})$. The natural definition is given below, by induction over Γ.

$$\begin{array}{lcl}(desync_\Gamma, sync_\Gamma) & \in & (\mathbf{S}[\![\Gamma]\!] \Rightarrow_c Env(\mathbb{K})) \times (Env(\mathbb{K}) \Rightarrow_c \mathbf{S}[\![\Gamma]\!]) \\ (desync_\square, sync_\square) & = & (\lambda\_.\bot, \lambda\sigma.\bot) \\ (desync_{\Gamma,x:t}, sync_{\Gamma,x:t}) & = & (\lambda(\gamma, v).(desync_\Gamma \gamma)[x \mapsto desync_t\, v], \lambda\sigma.(sync_\Gamma \sigma, sync_t(\sigma x)))\end{array}$$

Technically, $(desync_\Gamma, sync_\Gamma)$ is not an e-p pair since $desync_\Gamma$ is not injective. To see why, take $\Gamma = x : \mathbf{int} :: (1), x : \mathbf{int} :: (1)$, in which case $\mathbf{S}[\![\Gamma]\!]$ is isomorphic to the product domain $CStream_{(1)^\omega}(\mathbb{N}_\bot) \times CStream_{(1)^\omega}(\mathbb{N}_\bot)$, and consider the following γ and σ.

$$\begin{array}{lcllcl}\gamma & \in & \mathbf{S}[\![\Gamma]\!] & \sigma & \in & Env(\mathbb{K}) \\ \gamma & = & ([0]^\omega, [1]^\omega) & \sigma & = & \bot[y \mapsto stream(2^\omega)][x \mapsto stream(1^\omega)]\end{array}$$

We have $sync_\Gamma(desync_\Gamma \gamma) = ([1]^\omega, [1]^\omega)$ which is not even comparable to γ, and thus the functions do not form an e-p pair. On the other hand, the expected inequality $desync_\Gamma(sync_\Gamma \sigma) = \bot[stream(x \mapsto (1)^\omega)] \sqsubseteq \sigma$ still holds. More generally, the following result shows that the functions satisfy only the second property of an e-p pair $desync_\Gamma \circ sync_\Gamma$ is only a deflation.

Property 20. *For any Γ, $desync_\Gamma \circ sync_\Gamma$ is a deflation.*

Proof. Let us first prove that the function $desync_\Gamma \circ sync_\Gamma \sqsubseteq id$ by induction on Γ. The case for empty contexts is immediate because $\bot$ is minimal; for $\Gamma, x:t$, we have, for any σ,

$$\begin{array}{lll} desync_{\Gamma,x:t}(sync_{\Gamma,x:t}\sigma) = desync_\Gamma(sync_\Gamma\sigma)[x \mapsto desync_t(sync_t\sigma(x))] & \text{(def.)} \\ \sqsubseteq desync_\Gamma(sync_\Gamma\sigma)[x \mapsto \sigma(x)] & (desync_t \circ sync_t \sqsubseteq id) \\ \sqsubseteq \sigma[x \mapsto \sigma(x)] & \text{(ind. hyp.)} \\ = \sigma & \end{array}$$

which concludes the proof that the function is smaller than id.

Now, we prove the stronger result that $sync_\Gamma \circ desync_\Gamma \circ sync_\Gamma = sync_\Gamma$ by induction on Γ. The case for empty contexts is once again immediate. The induction case is, for any σ,

$$\begin{array}{lll} & sync_{\Gamma,x:t}(desync_{\Gamma,x:t}(sync_{\Gamma,x:t}\sigma)) & \\ = & (sync_\Gamma(desync_\Gamma(sync_\Gamma\sigma)), sync_t(desync_t(sync_t\sigma(x))) & \text{(def.)} \\ = & (sync_\Gamma\sigma, sync_t(desync_t(sync_t\sigma(x))) & \text{(ind. hyp.)} \\ = & (sync_\Gamma\sigma, sync_t\sigma(x)) & (desync_t \circ sync_t = id) \\ = & sync_{\Gamma,x:t}\sigma & \text{(def.)} \end{array}$$

and thus $desync_\Gamma \circ sync_\Gamma \circ desync_\Gamma \circ sync_\Gamma = desync_\Gamma \circ sync_\Gamma$ which concludes the proof that the function is idempotent. □

We will come back to this discrepancy between $Env(\mathbb{K})$ and $\mathbf{S}[\![\Gamma]\!]$ once we have defined the typed semantics. Let us mention that it is related to the fact that the untyped semantics follows lexical scope *by construction*, as the definition of $\mathbf{K}[\![\texttt{fun}\, x.\, e]\!](\sigma) = \lambda k.\mathbf{K}[\![e]\!]\sigma[x \mapsto k]$ overrides any previous value for x present in σ, while for the typed semantics this is a result that needs to be established.

Synchronizing typings The synchonization of typings establishes an e-p pair between the domain of interpretation of typed programs, $\mathbf{S}[\![\Gamma \vdash t]\!]$, and the domain of interpretation of untyped programs, $Env(\mathbb{K}) \Rightarrow_c \mathbb{K}$. This is built using the function space e-p pair.

$$\begin{array}{lll} (desync_{\Gamma\vdash t}, sync_{\Gamma\vdash t}) & \in & (\mathbf{S}[\![\Gamma \vdash t]\!] \Rightarrow_c (Env(\mathbb{K}) \Rightarrow_c \mathbb{K})) \times ((Env(\mathbb{K}) \Rightarrow_c \mathbb{K}) \Rightarrow_c \mathbf{S}[\![\Gamma \vdash t]\!]) \\ (desync_{\Gamma\vdash t}, sync_{\Gamma\vdash t}) & = & (desync_\Gamma, sync_\Gamma) \Rightarrow_c (desync_t, sync_t) \end{array}$$

As in the case of contexts, this family of functions gives rise to a deflation.

Property 21. *For any Γ, $desync_{\Gamma\vdash t} \circ sync_{\Gamma\vdash t}$ is a deflation.*

Proof. The proof is similar to the above case and general considerations on idempotents and deflations. For instance, one has

$$\begin{array}{lll} desync_{\Gamma\vdash t} \circ sync_{\Gamma\vdash t} = \lambda f.desync_\Gamma \circ sync_t \circ f \circ desync_\Gamma \circ sync_\Gamma & \text{(def.)} \\ \sqsubseteq \lambda f.desync_\Gamma \circ sync_t \circ f & \text{(Property 20)} \\ \sqsubseteq \lambda f.f & (desync_t \circ sync_t \sqsubseteq id) \end{array}$$

which shows that $desync_{\Gamma\vdash t} \circ sync_{\Gamma\vdash t} \sqsubseteq id$. □

$$\boxed{\mathbf{S}[\![\Gamma \vdash e : t]\!]}$$

$$\mathbf{S}[\![\Gamma \vdash e : t]\!] \in \mathbf{S}[\![\Gamma \vdash t]\!]$$

$$\begin{array}{ll} & \mathbf{S}[\![\Gamma, x : t \vdash x : t]\!] \\ = & \pi_r \end{array}$$

$$\begin{array}{ll} & \mathbf{S}[\![\Gamma, x : t' \vdash e : t]\!] \\ = & \mathbf{S}[\![\Gamma \vdash e : t]\!] \circ \pi_l \end{array}$$

$$\begin{array}{ll} & \mathbf{S}[\![\Gamma \vdash \mathtt{fun}\, x.e : t \multimap t']\!] \\ = & \lambda\gamma.\lambda v.(\mathbf{S}[\![\Gamma, x : t \vdash e : t']\!](\gamma, v)) \end{array}$$

$$\begin{array}{ll} & \mathbf{S}[\![\Gamma \vdash e\ e' : t']\!] \\ = & \lambda\gamma.(\mathbf{S}[\![\Gamma_1 \vdash e : t \multimap t']\!]\gamma_1(\mathbf{S}[\![\Gamma_2 \vdash e : t]\!]\ \gamma_2) \text{ where } (\gamma_1, \gamma_2) = \mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!]\gamma) \end{array}$$

$$\begin{array}{ll} & \mathbf{S}[\![\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2]\!] \\ = & (\mathbf{S}[\![\Gamma_1 \vdash e_1 : t_1]\!] \times \mathbf{S}[\![\Gamma_2 \vdash e_2 : t_2]\!]) \circ \mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!] \end{array}$$

$$\begin{array}{ll} & \mathbf{S}[\![\Gamma \vdash \mathtt{let}\ (x, y) = e\ \mathtt{in}\ e' : t]\!] \\ = & \lambda\gamma.(\mathbf{S}[\![\Gamma_2, x : t_1, y : t_2 \vdash e' : t]\!]((\gamma_2, v_1), v_2) \text{ where } (\gamma_1, \gamma_2) = \mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!]\gamma \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (v_1, v_2) = \mathbf{S}[\![\Gamma_1 \vdash e : t_1 \otimes t_2]\!]\gamma_1) \end{array}$$

$$\begin{array}{ll} & \mathbf{S}[\![\Gamma \vdash \mathtt{fix}\, e : t']\!] \\ = & \lambda\gamma.\mathbf{fix}\,(\mathbf{S}[\![\Gamma \vdash e : t \multimap t']\!]\gamma \circ \mathbf{S}[\![\vdash t' <:_1 t]\!]) \end{array}$$

$$\begin{array}{ll} & \mathbf{S}[\![\Box \vdash s : dtof(s) :: ct]\!] \\ = & sync_{\Box \vdash dtof(s) :: ct}\, \mathbf{K}[\![s]\!] \end{array}$$

$$\begin{array}{ll} & \mathbf{S}[\![\Box \vdash op : (\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct) \multimap (\mathbf{int} :: ct)]\!] \\ = & sync_{\Box \vdash (\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct) \multimap (\mathbf{int} :: ct)}\, \mathbf{K}[\![op]\!] \end{array}$$

$$\begin{array}{ll} & \mathbf{S}[\![\Box \vdash \mathtt{merge}\ p : (dt :: ct\ \mathbf{on}\ p) \otimes (dt :: ct\ \mathbf{on}\ \overline{p}) \multimap (dt :: ct)]\!] \\ = & sync_{\Box \vdash (dt :: ct\ \mathbf{on}\ p) \otimes (dt :: ct\ \mathbf{on}\ \overline{p}) \multimap (dt :: ct)}\, \mathbf{K}[\![\mathtt{merge}\ p]\!] \end{array}$$

$$\begin{array}{ll} & \mathbf{S}[\![\Box \vdash \mathtt{when}\ p : (dt :: ct) \multimap (dt :: ct\ \mathbf{on}\ p)]\!] \\ = & sync_{\Box \vdash (dt :: ct) \multimap (dt :: ct\ \mathbf{on}\ p)}\, \mathbf{K}[\![\mathtt{when}\ p]\!] \end{array}$$

$$\begin{array}{ll} & \mathbf{S}[\![\Gamma \vdash e : t']\!] \\ = & \mathbf{S}[\![\vdash t <:_k t']\!] \circ \mathbf{S}[\![\Gamma \vdash e : t]\!] \end{array}$$

$$\begin{array}{ll} & \mathbf{S}[\![\Gamma \vdash e : t]\!] \\ = & \mathbf{S}[\![\vdash t' \uparrow_{ct} t]\!] \circ \mathbf{S}[\![\Gamma' \vdash e : t']\!] \circ \mathbf{S}[\![\vdash \Gamma \downarrow_{ct} \Gamma']\!] \end{array}$$

Figure 3.19: Typed semantics - main function

3.3.2 Interpreting Typing Judgments

Interpreting typed expressions The typed semantics of expressions can be found in Figure 3.19. As expected, a derivation of conclusion $\Gamma \vdash e : t$ is interpreted as an inhabitant of $\mathbf{S}[\![\Gamma \vdash t]\!]$, defined by induction on the structure of typing derivations. Each case of the definition corresponds to a typing rule present in Figure 3.5.

1. We interpret the VAR rule as the second projection. This picks the rightmost component of the input context, in accordance with the definition of $\mathbf{S}[\![\Gamma, x : t]\!]$.

2. In contrast, the WEAKEN rule uses the left projection to remove the rightmost binding in (the interpretation of) Γ.

3. The interpretation of FUN is currying.

4. The rule APP should be interpreted as application in the metalanguage of domains. There is, however, a slight difficulty: the premises of the rule involve contexts different from the one in its conclusion. We thus need to interpret the context splitting judgment $\Gamma \vdash \Gamma_1 \otimes \Gamma_2$ as a function from $\mathbf{S}[\![\Gamma]\!]$ to $\mathbf{S}[\![\Gamma_1]\!]$ and $\mathbf{S}[\![\Gamma_2]\!]$. We give its definition in the next paragraph.

5. The rules PAIR and LETPAIR are similar to the previous one, except that we use pair construction and destruction rather than application.

6. The FIX rule handles recursive definition using the fixpoint functional from Kleene's theorem. However, we need a way to transform an inhabitant of t' into an inhabitant of t. We do this by interpreting the adaptability judgment $\vdash t' <:_0 t$ as a continuous function from $\mathbf{S}[\![t']\!]$ into $\mathbf{S}[\![t]\!]$. Its definition is given later in this section.

7. Consider the interpretation of the four rules CONST, OP, MERGE and WHEN. These rules are syntax-directed, and were interpreted in Section 3.1 using dedicated combinators. We would like to reuse this untyped interpretation in the typed semantics, or in other words to turn an inhabitant of $\mathbb{K}$ inside an element of $\mathbf{S}[\![\Gamma \vdash t]\!]$. This can be done through the functions $(desync_{\Gamma \vdash t}, sync_{\Gamma \vdash t})$ defined in the previous section. Since all four rules have a conclusion of the form $\square \vdash e : t$ and no premises, we interpret them as $sync_{\square \vdash t}\, \mathbf{K}[\![e]\!]$.

8. For the interpretation of the SUB rule we compose the interpretation of $\Gamma \vdash e : t$ with that of the adaptability judgment $\vdash t <:_k t'$.

9. The final rule, RESCALE, also involves interpreting auxiliary judgments, here of gathering/scattering. As in the case of adaptability, a judgment $\vdash t \uparrow_{ct} t'$ or $\vdash t \downarrow_{ct} t'$ will be interpreted as an element of $\mathbf{S}[\![t]\!] \Rightarrow_c \mathbf{S}[\![t']\!]$. The same is true for context scattering. We give these interpretations later. Thus, we can interpret the RESCALE rule in three steps. First, scatter the input γ inhabiting Γ into γ' which inhabits Γ', using $\mathbf{S}[\![\vdash \Gamma \downarrow_{ct} \Gamma']\!]$. Then, apply the interpretation of $\Gamma' \vdash e : t'$ to γ'. Finally, gather the resulting inhabitant of t' into one of t', using $\mathbf{S}[\![\vdash t' \uparrow_{ct} t]\!]$. Hence we obtain a function in $\mathbf{S}[\![\Gamma \vdash t]\!]$ from one int $\mathbf{S}[\![\Gamma' \vdash t']\!]$.

$$\boxed{\mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!]}$$

$$\begin{array}{lcl}
\mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!] & \in & \mathbf{S}[\![\Gamma]\!] \Rightarrow_c \mathbf{S}[\![\Gamma_1]\!] \times \mathbf{S}[\![\Gamma_2]\!] \\
\mathbf{S}[\![\square \vdash \square \otimes \square]\!] & = & \lambda_.(\bot, \bot) \\
\mathbf{S}[\![\Gamma, x : t \vdash \Gamma_1, x : t \otimes \Gamma_2, x : t]\!] & = & \lambda(\gamma, v).((\gamma_1, v), (\gamma_2, v)) \text{ where } (\gamma_1, \gamma_2) = \mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!]\ \gamma \\
\mathbf{S}[\![\Gamma, x : t \vdash \Gamma_1, x : t \otimes \Gamma_2]\!] & = & \lambda(\gamma, v).((\gamma_1, v), \gamma_2) \text{ where } (\gamma_1, \gamma_2) = \mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!]\ \gamma \\
\mathbf{S}[\![\Gamma, x : t \vdash \Gamma_1 \otimes \Gamma_2, x : t]\!] & = & \lambda(\gamma, v).(\gamma_1, (\gamma_2, v)) \text{ where } (\gamma_1, \gamma_2) = \mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!]\ \gamma
\end{array}$$

Figure 3.20: Typed semantics - context splitting judgment

$$\boxed{\mathbf{S}[\![\vdash t <:_k t']\!]}$$

$$\begin{array}{lcl}
\mathbf{S}[\![\vdash t <:_k t']\!] & \in & \mathbf{S}[\![t]\!] \Rightarrow_c \mathbf{S}[\![t']\!] \\
\mathbf{S}[\![\vdash dt :: ct <:_k dt :: ct']\!] & = & \mathbf{repack}_{\mathbf{S}[\![ct']\!]} \\
\mathbf{S}[\![\vdash t_1 :: t_2 <:_k t_1' :: t_2']\!] & = & \mathbf{S}[\![\vdash t_1 <:_k t_1']\!] \times \mathbf{S}[\![\vdash t_2 <:_k t_2']\!] \\
\mathbf{S}[\![\vdash t_1 \multimap t_2 <:_0 t_1' \multimap t_2']\!] & = & \lambda f.(\mathbf{S}[\![\vdash t_2 <:_k t_2']\!] \circ f \circ \mathbf{S}[\![\vdash t_1' <:_k t_1]\!])
\end{array}$$

Figure 3.21: Typed semantics - adaptability judgment

Remark 12. The case of rules VAR, WEAKEN and LAMBDA correspond to the standard interpretation of the λ-calculus in a cartesian-closed category. They can be found for instance in the book of Amadio and Curien [1998, Chapter 4].

Remark 13. At this point, the reader may feel that this use of projections looks a bit mysterious. Indeed, what happens if we use the wrong projection, or equivalently if we chose the wrong type t? We will show later in this chapter that the elements obtained in this way are *total*, showing that the types chosen are in fact far from arbitrary.

Interpreting context splitting The intepretation of the context splitting judgment is given in Figure 3.20. For rule SEPEMPTY, we return a pair of unit values. For SEPLEFT, we take the rightmost component of the input context and send it to the left output. This is reversed in the case of rule SEPRIGHT. For rule SEPCONTRACT, we duplicate the rightmost component of the input context and sends it to both outputs.

Interpreting adaptability The interpretation is given in Figure 3.21. For rule ADAPTSTREAM, we rely on the ideas outlined in Section 2.6. The function $\mathbf{repack}_w$ acts a coercion from arbitrary clocked streams to clocked streams of clock w. The interpretation of rule ADAPTPROD handles product by putting buffers in parallel. For rule ADAPTFUN we put buffers on the input and output of the function inhabiting $\mathbf{S}[\![t_1 \multimap t_2]\!]$ to turn it into an inhabitant of $\mathbf{S}[\![t_1' \multimap t_2']\!]$.

Interpreting gathering/scattering The interpretations of the gathering and scattering judgments for types are given in Figure 3.22. They follow the intuitions given in Section 3.2.

$$\boxed{\mathbf{S}[\![\vdash t \uparrow_{ct} t']\!]} \text{ and } \boxed{\mathbf{S}[\![\vdash t \downarrow_{ct} t']\!]} \text{ and } \boxed{\mathbf{S}[\![\vdash \Gamma \downarrow_{ct} \Gamma']\!]}$$

$$\begin{array}{lcl}
\mathbf{S}[\![\vdash t \uparrow_{ct} t']\!] & \in & \mathbf{S}[\![t]\!] \Rightarrow_c \mathbf{S}[\![t']\!] \\
\mathbf{S}[\![\vdash dt :: ct' \uparrow_{ct} dt :: ct'']\!] & = & \mathbf{repack}_{\mathbf{S}[\![ct'']\!]} \\
\mathbf{S}[\![\vdash t_1 \otimes t_2 \uparrow_{ct} t_1' \otimes t_2']\!] & = & \mathbf{S}[\![\vdash t_1 \uparrow_{ct} t_1']\!] \times \mathbf{S}[\![\vdash t_2 \uparrow_{ct} t_2']\!] \\
\mathbf{S}[\![\vdash t_1 \multimap t_2 \uparrow_{ct} t_1' \multimap t_2']\!] & = & \lambda f.(\mathbf{S}[\![\vdash t_2 \uparrow_{ct} t_2']\!] \circ f \circ \mathbf{S}[\![\vdash t_1' \downarrow_{ct} t_1]\!]) \\
\mathbf{S}[\![\vdash t \uparrow_{ct \,\mathbf{on}\, ct'} t']\!] & = & \mathbf{S}[\![\vdash t'' \uparrow_{ct} t']\!] \circ \mathbf{S}[\![\vdash t \uparrow_{ct'} t'']\!] \\
\mathbf{S}[\![\vdash t \uparrow_{ct} t']\!] & = & \mathbf{S}[\![\vdash t \downarrow_{ct'} t']\!]
\end{array}$$

$$\begin{array}{lcl}
\mathbf{S}[\![\vdash t \downarrow_{ct} t']\!] & \in & \mathbf{S}[\![t]\!] \Rightarrow_c \mathbf{S}[\![t']\!] \\
\mathbf{S}[\![\vdash dt :: ct' \downarrow_{ct} dt :: ct'']\!] & = & \mathbf{repack}_{\mathbf{S}[\![ct']\!]} \\
\mathbf{S}[\![\vdash t_1 \otimes t_2 \downarrow_{ct} t_1' \otimes t_2']\!] & = & \mathbf{S}[\![\vdash t_1 \downarrow_{ct} t_1']\!] \times \mathbf{S}[\![\vdash t_2 \downarrow_{ct} t_2']\!] \\
\mathbf{S}[\![\vdash t_1 \multimap t_2 \downarrow_{ct} t_1' \multimap t_2']\!] & = & \lambda f.(\mathbf{S}[\![\vdash t_2 \downarrow_{ct} t_2']\!] \circ f \circ \mathbf{S}[\![\vdash t_1' \uparrow_{ct} t_1]\!]) \\
\mathbf{S}[\![\vdash t \downarrow_{ct \,\mathbf{on}\, ct'} t']\!] & = & \mathbf{S}[\![\vdash t'' \downarrow_{ct'} t']\!] \circ \mathbf{S}[\![\vdash t \downarrow_{ct} t'']\!] \\
\mathbf{S}[\![\vdash t \downarrow_{ct} t']\!] & = & \mathbf{S}[\![\vdash t \uparrow_{ct'} t']\!]
\end{array}$$

$$\begin{array}{lcl}
\mathbf{S}[\![\vdash \Gamma \downarrow_{ct} \Gamma']\!] & \in & \mathbf{S}[\![\Gamma]\!] \Rightarrow_c \mathbf{S}[\![\Gamma']\!] \\
\mathbf{S}[\![\vdash \square \downarrow_{ct} \square]\!] & = & \lambda_.\bot \\
\mathbf{S}[\![\vdash \Gamma, x : t \downarrow_{ct} \Gamma', x : t']\!] & = & \mathbf{S}[\![\vdash \Gamma \downarrow_{ct} \Gamma']\!] \times \mathbf{S}[\![\vdash t \downarrow_{ct} t']\!]
\end{array}$$

Figure 3.22: Typed semantics - gathering/scattering judgment

1. Rules UPSTREAM and DOWNSTREAM handling stream types rely on the $\mathbf{repack}_w$ function, as explained in Section 2.7: this function models both ordinary buffers and the communication between a local time scale and the external world.

2. The rules for products (UPPROD and DOWNPROD) correspond to pairings, as always.

3. The interpretations of UPON and DOWNON shows that the composition of clocks give rise to the composition of time scales and hence gather/scattering, as discussed in the previous section.

4. For interpreting rule UPARROW, one makes the input enter the local time scale through scattering, traverse the function, and finally go back to external time through gathering.

5. The interpretation of DOWNARROW is symmetric to the one of UPARROW.

Remark 14. We feel that this section justifies the (arguably high) level of formalism involved in the definition of our type system. Once the interpretation of types has been defined, there is very little freedom in the interpretation of each rule: for most cases in Figure 3.19 and Figure 3.22, there is only one well-typed possibility. This structured approach to program semantics will also be convenient for stating and proving properties of the system in the next section.

$$
\begin{array}{ll}
 & \mathbf{S}[\![\Gamma \vdash e : t]\!] \\
: & \mathbf{S}[\![\Gamma \vdash t]\!]
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{S}[\![\textsc{Var}(\_) ::: (\Gamma, x : t \vdash x : t)]\!] \\
= & \pi_r
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{S}[\![\textsc{Weaken}(d, \_) ::: (\Gamma, x : t' \vdash e : t)]\!] \\
= & \mathbf{S}[\![d ::: (\Gamma \vdash e : t)]\!] \circ \pi_l
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{S}[\![\textsc{Fun}(d) ::: (\Gamma \vdash \mathtt{fun}\, x.\, e : t \multimap t')]\!] \\
= & \lambda\gamma.\lambda v.(\mathbf{S}[\![d ::: (\Gamma, x : t \vdash e : t')]\!](\gamma, v))
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{S}[\![\textsc{App}(d_1, d_2, d_3) ::: (\Gamma \vdash e\ e' : t')]\!] \\
= & \lambda\gamma.(\mathbf{S}[\![d_2 ::: (\Gamma_1 \vdash e : t \multimap t')]\!]\gamma_1(\mathbf{S}[\![d_3 ::: (\Gamma_2 \vdash e : t)\ \gamma_2)]\!] \text{ where } (\gamma_1, \gamma_2) = \mathbf{S}[\![d_1 ::: (\Gamma \vdash \Gamma_1 \otimes \Gamma_2)]\!]\gamma)
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{S}[\![\textsc{Pair}(d_1, d_2, d_3) ::: (\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2)]\!] \\
= & (\mathbf{S}[\![d_2 ::: (\Gamma_1 \vdash e_1 : t_1)]\!] \times \mathbf{S}[\![d_3 ::: (\Gamma_2 \vdash e_2 : t_2)]\!]) \circ \mathbf{S}[\![d_1 ::: (\Gamma \vdash \Gamma_1 \otimes \Gamma_2)]\!]
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{S}[\![\textsc{LetPair}(d_1, d_2, d_3) ::: (\Gamma \vdash \mathtt{let}\ (x, y) = e\ \mathtt{in}\ e' : t)]\!] \\
= & \lambda\gamma.(\mathbf{S}[\![d_3 ::: (\Gamma_2, x : t_1, y : t_2 \vdash e' : t)]\!]((\gamma_2, v_1), v_2) \text{ where } \begin{array}{l}(\gamma_1, \gamma_2) = \mathbf{S}[\![d_1 ::: (\Gamma \vdash \Gamma_1 \otimes \Gamma_2)]\!]\gamma \\ (v_1, v_2) = \mathbf{S}[\![d_2 ::: (\Gamma_1 \vdash e : t_1 \otimes t_2)]\!]\gamma_1)\end{array}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{S}[\![\textsc{Fix}(d_1, d_2, \_) ::: (\Gamma \vdash \mathtt{fix}\, e : t')]\!] \\
= & \lambda\gamma.\mathbf{fix}\,(\mathbf{S}[\![d_1 ::: (\Gamma \vdash e : t \multimap t')]\!]\gamma \circ \mathbf{S}[\![d_2 ::: (\vdash t' <:_1 t)]\!])
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{S}[\![\textsc{Const}() ::: (\square \vdash s : dtof(s) :: ct)]\!] \\
= & sync_{\square \vdash dtof(s) :: ct}\mathbf{K}[\![s]\!]
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{S}[\![\textsc{Op}() ::: (\square \vdash op : (\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct) \multimap (\mathbf{int} :: ct))]\!] \\
= & sync_{\square \vdash (\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct) \multimap (\mathbf{int} :: ct)}\mathbf{K}[\![op]\!]
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{S}[\![\textsc{Merge}() ::: (\square \vdash \mathtt{merge}\ p : (dt :: ct\ \mathbf{on}\ p) \otimes (dt :: ct\ \mathbf{on}\ \overline{p}) \multimap (dt :: ct))]\!] \\
= & sync_{\square \vdash (dt :: ct\ \mathbf{on}\ p) \otimes (dt :: ct\ \mathbf{on}\ \overline{p}) \multimap (dt :: ct)}\mathbf{K}[\![\mathtt{merge}\ p]\!]
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{S}[\![\textsc{When}() ::: (\square \vdash \mathtt{when}\ p : (dt :: ct) \multimap (dt :: ct\ \mathbf{on}\ p))]\!] \\
= & sync_{\square \vdash (dt :: ct) \multimap (dt :: ct\ \mathbf{on}\ p)}\mathbf{K}[\![\mathtt{when}\ p]\!]
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{S}[\![\textsc{Sub}(d_1, d_2) ::: (\Gamma \vdash e : t')]\!] \\
= & \mathbf{S}[\![d_1 ::: (\vdash t <:_k t')]\!] \circ \mathbf{S}[\![d_2 ::: (\Gamma \vdash e : t)]\!]
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{S}[\![\textsc{Rescale}(d_1, d_2, d_3) ::: (\Gamma \vdash e : t)]\!] \\
= & \mathbf{S}[\![d_3 ::: (\vdash t' \uparrow_{ct} t)]\!] \circ \mathbf{S}[\![d_2 ::: (\Gamma' \vdash e : t')]\!] \circ \mathbf{S}[\![d_1 ::: (\vdash \Gamma \downarrow_{ct} \Gamma')]\!]
\end{array}
$$

Figure 3.23: Interpretation of the typing judgment with explicit Church-style derivations

Remark 15. The notation used in Figures 3.19, 3.21, and 3.22 is abusive since there we identify derivation trees with their conclusions. A more rigorous, but harder to read, formulation of these interpretations would manipulate derivations as trees whose nodes are labeled with rule names and judgments and whose sub-trees are other derivations. For example, the derivation $d = \textsc{Var}(d')$ corresponds to a derivation whose conclusion is an application to the VAR rule. We write $d ::: (\Gamma, x : t \vdash x : t)$ to indicate that the conclusion of the derivation d is $\Gamma, x : t \vdash x : t$. Because the VAR rule has a premise $\vdash \Gamma$ value, the derivation d has a sub-derivation d', and furthermore we have $d' ::: (\vdash \Gamma \text{ value})$. A typing rule reflecting this information could be given as follows.

$$\frac{d' ::: (\vdash \Gamma \text{ value})}{\textsc{Var}(d) ::: (\Gamma, x : t \vdash x : t)}\ \textsc{VarExplicit}$$

The interpretation of typing derivations adapted to this syntax is given in Figure 3.23. We will generally adopt the more lightweight style used in this section when no confusion arises.

3.4 Metatheoretical Properties

In this section we study two important properties of the typed semantics, and derive a number of useful corollaries. The first property is totality: typed programs only denote total streams. In other words, programs do not deadlock. The second one relates the typed and untyped semantics. We have seen that the interpretation of the stream-processing operators were the *synchronized* versions of their untyped interpretation. This is in fact the case for the interpretation of *all* typing rules, as we will show.

3.4.1 Totality

The rule FIX for fixpoints forces the outputs of the function to be 1-adaptable to its inputs; in other words, one may only compute fixpoints of functions whose outputs never *instantaneously* depend on their inputs. If the type system is sound, this should ensure that all streams computed in a well-typed program are total. In particular, we should have the following.

Theorem (Stream Totality)**.** *Any closed well-typed expression of stream type denotes a total clocked stream. More formally, for any e, data type dt and clock type ct, one has*

$$\mathbf{S}[\![\Box \vdash e : dt :: ct]\!] \Downarrow_\infty$$

We will prove this theorem by induction over typing derivations. The induction hypothesis has to be generalized to non-stream types and non-empty contexts.

First try Let us try to define the proper induction hypothesis. We have to decide what it means to be a total pair or a total function, in addition to total streams. A total pair is simply a pair of total elements. On the other hand, a function is total if it maps total elements to total

$$\frac{xs \Downarrow_\infty}{xs \Downarrow_\infty^{dt :: ct}} \qquad \frac{x_1 \Downarrow_\infty^{t_1} \qquad x_2 \Downarrow_\infty^{t_2}}{(x_1, x_2) \Downarrow_\infty^{t_1 \otimes t_2}} \qquad \frac{\forall x \in \mathbf{S}[\![t_1]\!], x \Downarrow_\infty^{t_1} \Rightarrow f(x) \Downarrow_\infty^{t_2}}{f \Downarrow_\infty^{t_1 \multimap t_2}}$$

$$\frac{}{\bot \Downarrow_\infty^{\square}} \qquad \frac{\gamma \Downarrow_\infty^{\Gamma} \qquad x \Downarrow_\infty^{t}}{(\gamma, x) \Downarrow_\infty^{\Gamma, \mathbf{x}:t}}$$

$$\frac{\forall \gamma \in \mathbf{S}[\![\Gamma]\!], \gamma \Downarrow_\infty^{\Gamma} \Rightarrow f(\gamma) \Downarrow_\infty^{t}}{f \Downarrow_\infty^{\Gamma \vdash t}}$$

Figure 3.24: Tentative totality predicate

$$\frac{xs \Downarrow_n}{xs \Downarrow_n^{dt :: ct}} \qquad \frac{x_1 \Downarrow_n^{t_1} \qquad x_2 \Downarrow_n^{t_2}}{(x_1, x_2) \Downarrow_n^{t_1 \otimes t_2}} \qquad \frac{\forall m \leq n, \forall x \in \mathbf{S}[\![t_1]\!], x \Downarrow_m^{t_1} \Rightarrow f(x) \Downarrow_m^{t_2}}{f \Downarrow_n^{t_1 \multimap t_2}}$$

$$\frac{}{\bot \Downarrow_n^{\square}} \qquad \frac{\gamma \Downarrow_n^{\Gamma} \qquad x \Downarrow_n^{t}}{(\gamma, x) \Downarrow_n^{\Gamma, \mathbf{x}:t}}$$

$$\frac{\forall \gamma \in \mathbf{S}[\![\Gamma]\!], \gamma \Downarrow_n^{\Gamma} \Rightarrow f(\gamma) \Downarrow_n^{t}}{f \Downarrow_n^{\Gamma \vdash t}}$$

Figure 3.25: Step-indexed totality predicate

elements. Formally, we will write $x \Downarrow_\infty^t$ to express that $x \in \mathbf{S}[\![t]\!]$ is total, and define this predicate by induction on t. This notion extends to contexts Γ and typings $\Gamma \vdash t$, treating contexts as nested pairs and typings as functions. The corresponding predicate is presented as inference rules in Figure 3.24.

Now, to obtain the totality for streams we should prove the more general result, stated using the definitions above to package induction hypotheses in a type- and context-directed fashion.

Lemma (Totality)**.** *The interpretation of a typing derivation proving that an expression e has type t in the context* Γ *satisfies the totality predicate. In other words, one has*

$$\mathbf{S}[\![\Gamma \vdash e : t]\!] \Downarrow_\infty^{\Gamma \vdash t}$$

This lemma is equivalent to Theorem 3.4.1 when e is a closed expression of stream type. To see why, unfold the definition of the totality predicate $\Downarrow_\infty^{\vdash dt :: ct}$.

Step-indexing Unfortunately, the previous totality predicate is not strong enough for the proof to go through. The problem arises when trying to prove the totality lemma of the FIX

rule. To understand why, let us go back to Example 16. Consider the expressions *consincr*, of type $0(1) \multimap (1)$, and *nat*, which is the fixpoint of *consincr*. Since this program is accepted, the fixpoint in *nat* defines a total stream. But why?

The untyped semantics of *consincr* could be written as follows, following the conventions used in Chapter 2.

$$\begin{array}{rcl} \mathit{consincr}\ \mathit{xs} & = & 0.(\mathit{incr}\ \mathit{xs}) \\ \mathit{incr}\ (x.\mathit{xs}) & = & (x+1).(\mathit{incr}\ \mathit{xs}) \end{array}$$

This function is non-strict: it maps $\bot$ to the partial stream $0.\bot$, and more generally a stream *xs* converging up to n to $0.(\mathit{incr}\ \mathit{xs})$ which converges up to $n+1$. Its clock type reflects this fact, expressing that *consincr* maps an integer stream converging up to n whose clock is a *prefix* of $0(1)^\omega$ to an integer stream converging up to n whose clock is a *prefix* of $(1)^\omega$. Since $(1)^\omega <:_1 0(1)^\omega$, the amount of data in the output stream increases strictly with each approximation of the fixpoint, which is therefore total.

This information is not reflected in the totality predicate in Figure 3.24, which only expresses properties of functions whose arguments are *total* streams. To solve this problem, we have to reason explicitly about partial objects in the predicate. For streams, we already have the partial, "up to n" predicate $\_ \Downarrow_n$ which approximates the total convergence predicate $\_ \Downarrow_\infty$, given in Chapter 2. We lift it to functions and arrows in Figure 3.25. A pair converges up to n if both of its components do. The case of functions is more subtle than before, forcing the function to preserve convergence for all m smaller than the current n. This makes sure that a function which converges up to $n+1$ converges up to n. Contexts and typings also have their predicates, which are similar to that of products and functions. Finally, imitating the case of streams, we *define* the total convergence predicate in terms of the partial one.

Definition 16 (Total Convergence)**.** *An inhabitant f of $\mathbf{S}[\![\Gamma \vdash t]\!]$* converges totally *for the typing $\Gamma \vdash t$ when it converges partially up to all n for $\Gamma \vdash t$.*

$$f \Downarrow_\infty^{\Gamma \vdash t} \quad \stackrel{\text{def}}{=} \quad \forall n \in \mathbb{N},\ f \Downarrow_n^{\Gamma \vdash t}$$

The proof We first establish useful properties on the convergence predicate. They are all used for the proof of the totality lemma, and in particular for the fixpoint case.

Property 22. *All inhabitants of a type t converge up to zero.*

$$\forall t, \forall x \in \mathbf{S}[\![t]\!], x \Downarrow_0^t$$

Proof. We proceed by induction on t.

- Case $t = dt :: ct$: immediate from Definition 2.
- Case $t = t_1 \otimes t_2$: by induction hypotheses on t_1 and t_2.
- Case $t = t_1 \multimap t_2$: take $f \in \mathbf{S}[\![t_1 \multimap t_2]\!]$ and $x \in \mathbf{S}[\![t_1]\!]$. We have $f\,x \Downarrow_0^{t_2}$ by induction hypothesis, and thus $f \Downarrow_0^{t_1 \multimap t_2}$. □

Property 23. *Convergence up to $n+1$ implies convergence up to n, and that for both types and contexts.*

$$\forall n \in \mathbb{N},\ \forall x \in \mathbf{S}[\![t]\!],\ x \Downarrow^t_{n+1} \Rightarrow\ x \Downarrow^t_n$$
$$\forall n \in \mathbb{N},\ \forall \gamma \in \mathbf{S}[\![\Gamma]\!],\ \gamma \Downarrow^\Gamma_{n+1} \Rightarrow\ \gamma \Downarrow^\Gamma_n$$

Thus, for any x, n and m such that $n \leq m$ and $x \Downarrow^t_m$, we have $x \Downarrow^t_n$.

Proof. Let us prove this lemma for types, the case of contexts being similar to that of products. We proceed by induction on t.

- Case $t = dt :: ct$: immediate from Definition 2.
- Case $t = t_1 \otimes t_2$: by induction hypotheses on t_1 and t_2.
- Case $t = t_1 \multimap t_2$: take $f \in \mathbf{S}[\![t_1 \multimap t_2]\!]$ such that $f \Downarrow^{t_1 \multimap t_2}_{n+1}$. Let x be such that $x \Downarrow^{t_1}_n$. Because $n \leq n+1$, the fact that f converges up to $n+1$ implies $f\,x \Downarrow^{t_2}_n$. Thus $f \Downarrow^{t_1 \multimap t_2}_n$.

Once this has been established the last part of the property is readily proved by induction on the natural m. □

Property 24. *The convergence predicate for a type t is compatible with the order of the domain $\mathbf{S}[\![t]\!]$ in the following sense.*

$$\forall t,\ \forall n,\ \forall x, y \in \mathbf{S}[\![t]\!],\ x \sqsubseteq y \wedge x \Downarrow^t_n \Rightarrow\ y \Downarrow^t_n$$

Proof. By induction on t.

- Case $t = dt :: ct$: immediate.
- Case $t = t_1 \otimes t_2$: by induction hypotheses on t_1 and t_2.
- Case $t = t_1 \multimap t_2$: take $f, g \in \mathbf{S}[\![t_1 \multimap t_2]\!]$ such that $f \Downarrow^{t_1 \multimap t_2}_n$ and $f \sqsubseteq g$. Let x be such that $x \Downarrow^{t_1}_n$. We have $f\,x \Downarrow^{t_2}_n$ and, since $f \sqsubseteq g$, $f\,x \sqsubseteq g\,x$. The induction hypothesis on t_2 gives $g\,x \Downarrow^{t_2}_n$. □

We also need a characterization of the convergence of *map₂*, *when* and *merge*. It will be used in the corresponding cases of the proof.

Property 25. *Given a scalar $s \neq \bot \in \mathbb{S}$, we have*

$$\mathbf{const}\ s \Downarrow_\infty$$

Given a total function op and two streams xs and ys such that $xs \Downarrow_n$ and $ys \Downarrow_n$, we have

$$\mathbf{map_2}\ op\ xs\ ys \Downarrow_n$$

Given streams $w \leq (1)^\omega$, xs and ys such that $w \Downarrow_n$, $xs \Downarrow_{\mathcal{O}_w(n)}$ and $ys \Downarrow_{n-\mathcal{O}_w(n)}$, we have

$$\mathbf{merge}\ w\ (xs, ys) \Downarrow_n$$

Given streams $w \leq (1)^\omega$ and xs such that $w \Downarrow_n$ and $xs \Downarrow_n$, we have

$$\mathbf{when}\ w\ x \Downarrow_{\mathcal{O}_w(n)}$$

Proof. Each proof is done by induction on n. The base case is always immediate since any stream converges up to zero. The induction case for $\mathbf{map}_2$ is immediate. For **when** and **merge**, reason by case on the head of w which is either 0 or 1 since $w \leq (1)^\omega$. □

We have seen in the previous section that the separation, adaptability, gathering/scattering judgments denote continuous functions. Such functions obey convergence principles that are used for the totality lemma. We detail the adaptability and gathering/scattering cases.

Lemma 5 (Totality, Separation Judgment)**.** *Let* Γ, Γ_1 *and* Γ_2 *be contexts and let* γ *be in* $\mathbf{S}[\![\Gamma]\!]$. *Take* $(\gamma_1, \gamma_2) = \mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!]\gamma$. *For any* $n \in \mathbb{N}$ *such that* $\gamma \Downarrow_n^\Gamma$, *one has* $\gamma_1 \Downarrow_n^{\Gamma_1}$ *and* $\gamma_2 \Downarrow_n^{\Gamma_2}$.

Proof. Immediate by induction over typing derivations. □

An adaptability derivation proving $\vdash t <:_k t'$ acts as a coercion between t and t' in the sense that it describes a way to transform an inhabitant of t into one of t'. The delay k describes the amount of slack available between the producer and the consumer. Thus, the interpretation of a k-adaptability constraint sends an inhabitant of t converging up to some n into one of t' converging up to $n+k$.

Lemma 6 (Totality, Adaptability Judgment)**.** *Let* t *and* t' *be types and* x *an inhabitant of* t. *For any* $n \in \mathbb{N}$ *such that* $x \Downarrow_n^t$, *one has*

$$\mathbf{S}[\![\vdash t <:_k t']\!]x \Downarrow_{n+k}^{t'}$$

Proof. By induction on typing derivations proving $\vdash t <:_k t'$.

- Case ADAPTSTREAM: the premise $nf(ct) <:_k nf(ct')$ implies $\mathbf{S}[\![ct]\!] <:_k \mathbf{S}[\![ct]\!]$. This means in particular that $\mathcal{O}_{\mathbf{S}[\![ct]\!]}(i) \geq \mathcal{O}_{\mathbf{S}[\![ct']\!]}(i+k)$ for any i. Now, let x be such that $x \Downarrow_n^t$ for some n. Property 8 and $\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n) \geq \mathcal{O}_{\mathbf{S}[\![ct']\!]}(n+k)$ give $\mathbf{S}[\![\vdash dt :: ct <:_k dt :: ct]\!]x \Downarrow_{n+k}^{t'}$.
- Case ADAPTPROD: immediate use of the induction hypotheses on t_1 and t_2 given the definition of $\Downarrow_n^{t_1 \otimes t_2}$.
- Case ADAPTARROW: given $f \Downarrow_n^{t_1 \multimap t_2}$, we want to show $\mathbf{S}[\![\vdash t_1 \multimap t_2 <:_k t_1' \multimap t_2']\!]f \Downarrow_{n+k}^{t_1' \multimap t_2'}$. Assume $x \Downarrow_m^{t_1'}$ with $m \leq n+k$. We have

$$\begin{array}{rll}
x \in & \Downarrow_m^{t_1'} & \\
\mathbf{S}[\![\vdash t_1' <:_0 t_1]\!]\ x \in & \Downarrow_m^{t_1} & \text{(ind. hyp. on } \vdash t_1' <:_0 t_1) \\
\mathbf{S}[\![\vdash t_1' <:_0 t_1]\!]\ x \in & \Downarrow_{min(m,n)}^{t_1} & \text{(Property 23)} \\
(f \circ \mathbf{S}[\![\vdash t_1' <:_k t_1]\!])x \in & \Downarrow_{min(m,n)}^{t_2} & (f \Downarrow_n^{t_1 \multimap t_2}) \\
(\mathbf{S}[\![\vdash t_2 <:_k t_2']\!] \circ f \circ \mathbf{S}[\![\vdash t_1' <:_k t_1]\!])x \in & \Downarrow_{min(m,n)+k}^{t_2} & \text{(ind. hyp. on } \vdash t_2 <:_k t_2) \\
(\mathbf{S}[\![\vdash t_2 <:_k t_2']\!] \circ f \circ \mathbf{S}[\![\vdash t_1' <:_k t_1]\!])x \in & \Downarrow_{m+k}^{t_2} \cap \Downarrow_{n+k}^{t_2} & \\
(\mathbf{S}[\![\vdash t_2 <:_k t_2']\!] \circ f \circ \mathbf{S}[\![\vdash t_1' <:_k t_1]\!])x \in & \Downarrow_m^{t_2} & (m \leq m+k \text{ and } m \leq n+k)
\end{array}$$

  Hence $f \Downarrow_{n+k}^{t_1' \multimap t_2'}$. □

We now have to state an invariant for the gathering and scattering judgments. Chapter 2 gives the intuition in the case of streams with Property 10: the relation between the convergence of inputs and outputs can be described using the cumulative function of the clock driving the local time scale. Consider a time scale driven by a clock w.

- An element converging *inside* the time scale for $\mathcal{O}_w(n)$ local steps may leave the time scale via gathering, and the result converges for n global steps outside.
- Conversely, an element converging *outside* the time scale for n global steps may enter the time scale via scattering, and the result converges for $\mathcal{O}_w(n)$ time steps inside.

This is what the lemma below expresses formally.

Lemma 7 (Totality, Gathering and Scattering Judgments)**.** *Let t and t' be types, ct a clock type, and x an inhabitant of t. For any $n \in \mathbb{N}$, one has*

$$\begin{array}{rrcl} & x \Downarrow_n^t & \Rightarrow & \mathbf{S}[\![\vdash t \downarrow_{ct} t']\!] x \Downarrow^{t'}_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)} \\ \textit{and} & x \Downarrow^t_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)} & \Rightarrow & \mathbf{S}[\![\vdash t \uparrow_{ct} t']\!] x \Downarrow_n^{t'} \end{array}$$

Proof. By mutual induction over derivations proving $\vdash t \uparrow_{ct} t'$ and $\vdash t \downarrow_{ct} t'$. We prove rules in the order they have been explained in the previous sections.

- Case UpStream: taking $xs \Downarrow^{dt :: ct'}_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)}$, by definition we have $\mathbf{clock}\, xs =_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)} \mathbf{S}[\![ct']\!]$. The premise gives $\mathbf{S}[\![ct]\!]\ on\ \mathbf{S}[\![ct']\!] = \mathbf{S}[\![ct'']\!]$. We conclude by the second half of Property 10 that $\mathbf{repack}_{\mathbf{S}[\![ct'']\!]}\, xs \Downarrow_n^{dt :: ct''}$.
- Case DownStream: the reasoning is similar to the previous case, this time using the first half of Property 10.
- Case UpProd and DownProd: immediate use of the induction hypotheses on t_1 and t_2 given the definition of $\Downarrow_n^{t_1 \otimes t_2}$.
- Case UpArrow: take $f \Downarrow^{t_1 \multimap t_2}_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)}$ and $x \Downarrow_m^{t_1'}$ with $m \leq n$. Then

$$\begin{array}{rll} x \in & \Downarrow_m^{t_1'} & \\ \mathbf{S}[\![\vdash t_1' \downarrow_{ct} t_1]\!]\ x \in & \Downarrow^{t_1}_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(m)} & \text{(ind. hyp. on } \vdash t_1' \downarrow_{ct} t_1) \\ (f \circ \mathbf{S}[\![\vdash t_1' \downarrow_{ct} t_1]\!]) x \in & \Downarrow^{t_2}_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(m)} & (f \Downarrow^{t_1 \multimap t_2}_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)} \text{ and } \mathcal{O}_{\mathbf{S}[\![ct]\!]}(m) \leq \mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)) \\ (\mathbf{S}[\![\vdash t_2 \uparrow_{ct} t_2']\!] \circ f \circ \mathbf{S}[\![\vdash t_1' \downarrow_{ct} t_1]\!]) x \in & \Downarrow_m^{t_2'} & \text{(ind. hyp. on } \vdash t_2 \uparrow_{ct} t_2') \end{array}$$

  Hence $f \Downarrow_n^{t_1' \multimap t_2'}$.

- Case DOWNARROW: first, the premise $ct \le (1)$ implies $\mathbf{S}[\![ct]\!] \le (1)^\omega$, and thus that we have $\mathcal{O}_{\mathbf{S}[\![ct]\!]}(i) \le i$ for any i. Take $f \Downarrow_n^{t_1 \multimap t_2}$ and $x \Downarrow_m^{t_1'}$ with $m \le \mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)$. Since $\mathbf{S}[\![ct]\!]$ is binary, there exists $n' \le n$ such that $m = \mathcal{O}_{\mathbf{S}[\![ct]\!]}(n')$. Then

$$\begin{array}{rll}
 & x \in \Downarrow_m^{t_1'} & \\
 & x \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n')}^{t_1'} & (m = \mathcal{O}_{\mathbf{S}[\![ct]\!]}(n')) \\
\mathbf{S}[\![\vdash t_1' \uparrow_{ct} t_1]\!] & x \in \Downarrow_{n'}^{t_1} & \text{(ind. hyp. on } \vdash t_1' \uparrow_{ct} t_1) \\
(f \circ \mathbf{S}[\![\vdash t_1' \uparrow_{ct} t_1]\!]) & x \in \Downarrow_{n'}^{t_2} & (f \Downarrow_n^{t_1 \multimap t_2} \text{ and } n' \le n) \\
(\mathbf{S}[\![\vdash t_2 \downarrow_{ct} t_2']\!] \circ f \circ \mathbf{S}[\![\vdash t_1' \uparrow_{ct} t_1]\!]) & x \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n')}^{t_2} & \text{(ind. hyp. on } \vdash t_2 \downarrow_{ct} t_2') \\
(\mathbf{S}[\![\vdash t_2 \downarrow_{ct} t_2']\!] \circ f \circ \mathbf{S}[\![\vdash t_1' \uparrow_{ct} t_1]\!]) & x \in \Downarrow_m^{t_2} & (m = \mathcal{O}_{\mathbf{S}[\![ct]\!]}(n'))
\end{array}$$

  Hence $f \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)}^{t_1' \multimap t_2'}$.

- Case UPON: take $x \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct \textbf{ on } ct']\!]}(n)}^{t}$. Then

$$\begin{array}{rll}
 & x \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct \textbf{ on } ct']\!]}(n)}^{t} & \\
 & x \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct']\!]}(\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n))}^{t} & \text{(Property 9)} \\
\mathbf{S}[\![\vdash t \uparrow_{ct'} t'']\!] & x \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)}^{t} & \text{(ind. hyp. on } \vdash t \uparrow_{ct'} t'') \\
(\mathbf{S}[\![\vdash t'' \uparrow_{ct} t]\!] \circ \mathbf{S}[\![\vdash t \uparrow_{ct'} t'']\!]) & x \in \Downarrow_n^{t} & \text{(ind. hyp. on } \vdash t'' \uparrow_{ct} t')
\end{array}$$

- Case DOWNON: take $x \Downarrow_n^t$. Then

$$\begin{array}{rll}
 & x \in \Downarrow_n^{t} & \\
\mathbf{S}[\![\vdash t \downarrow_{ct} t'']\!] & x \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)}^{t} & \text{(ind. hyp. on } \vdash t \downarrow_{ct} t'') \\
(\mathbf{S}[\![\vdash t'' \downarrow_{ct'} t']\!] \circ \mathbf{S}[\![\vdash t \downarrow_{ct} t'']\!]) & x \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct']\!]}(\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n))}^{t} & \text{(ind. hyp. on } \vdash t'' \downarrow_{ct'} t') \\
(\mathbf{S}[\![\vdash t'' \downarrow_{ct'} t']\!] \circ \mathbf{S}[\![\vdash t \downarrow_{ct} t'']\!]) & x \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct \textbf{ on } ct]\!]}(n)}^{t} & \text{(Property 9)}
\end{array}$$

- Case UPINV: take $x \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)}^{t}$. Then

$$\begin{array}{rll}
 & x \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)}^{t} & \\
\mathbf{S}[\![\vdash t \downarrow_{ct'} t']\!] x & \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct']\!]}(\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n))}^{t'} & \text{(ind. hyp. on } \vdash t \downarrow_{ct} t') \\
\mathbf{S}[\![\vdash t \downarrow_{ct'} t']\!] x & \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct \textbf{ on } ct']\!]}(n)}^{t'} & \text{(Property 9)} \\
\mathbf{S}[\![\vdash t \downarrow_{ct'} t']\!] x & \in \Downarrow_n^{t'} & (ct \textbf{ on } ct' \equiv (1))
\end{array}$$

- Case DOWNINV: take $x \Downarrow_n^t$. Then

$$\begin{array}{rll}
 & x \in \Downarrow_n^{t} & \\
 & x \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct \textbf{ on } ct']\!]}(n)}^{t} & (ct \textbf{ on } ct' \equiv (1)) \\
 & x \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct']\!]}(\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n))}^{t} & \text{(Property 9)} \\
\mathbf{S}[\![\vdash t \uparrow_{ct'} t']\!] x & \in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)}^{t'} & \text{(ind. hyp. on } \vdash t \uparrow_{ct'} t')
\end{array}$$

which concludes the proof. □

We can finally prove the main result, which we had already stated.

Lemma 8 (Totality, Typing Judgment)**.** *Let Γ be a context, t a type and e an expression. Then, one has*

$$\mathbf{S}[\![\Gamma \vdash e : t]\!] \Downarrow_{\infty}^{\Gamma \vdash t}$$

Proof. By induction over typing derivations. The VAR, WEAKEN, LAMBDA, APP, PAIR and LET-PAIR rules are straightforward, applying induction hypotheses and Lemma 5. Let us detail the remaining cases, beginning with rules involving auxiliary judgments.

- Case FIX: assume $\gamma \Downarrow_n^{\Gamma}$. Let us abbreviate $\mathbf{S}[\![\Gamma \vdash e : t \multimap t']\!]\gamma$ as f. The induction hypothesis gives $f \Downarrow_n^{t \multimap t'}$. The following intermediate lemma is the key argument for convergence up to n.

  Lemma.
  $$\forall m \leq n, (f \circ \mathbf{S}[\![\vdash t' <:_1 t]\!])^m \bot \Downarrow_m^{t'}$$

  We proceed by induction on m. The base case is trivial by Property 22. For the induction case $m = m' + 1$, we have

  $$\begin{array}{rl}
  (f \circ \mathbf{S}[\![\vdash t' <:_1 t]\!])^{m'} \ \bot \in \Downarrow_{m'}^{t'} & \text{(ind. hyp.)} \\
  (\mathbf{S}[\![\vdash t' <:_1 t]\!] \circ (f \circ \mathbf{S}[\![\vdash t' <:_1 t]\!])^{m'})\bot \in \Downarrow_{m'+1}^{t} & \text{(Lemma 6)} \\
  (f \circ \mathbf{S}[\![\vdash t' <:_1 t]\!] \circ (f \circ \mathbf{S}[\![\vdash t' <:_1 t]\!])^{m'})\bot \in \Downarrow_{m'+1}^{t'} & (f \Downarrow_n^{t \multimap t'} \text{ and } m'+1 \leq n) \\
  (f \circ \mathbf{S}[\![\vdash t' <:_1 t]\!] \circ (f \circ \mathbf{S}[\![\vdash t' <:_1 t]\!])^{m'})\bot \in \Downarrow_n^{t'} & \\
  (f \circ \mathbf{S}[\![\vdash t' <:_1 t]\!])^{m'+1}\bot \in \Downarrow_{m'+1}^{t'} & \\
  (f \circ \mathbf{S}[\![\vdash t' <:_1 t]\!])^{m}\bot \in \Downarrow_{m}^{t'} & (m = m'+1)
  \end{array}$$

  We deduce from this lemma that $(f \circ \mathbf{S}[\![\vdash t' <:_1 t]\!])^n \bot \Downarrow_n^{t'}$. Now, remember that we have

  $$\mathbf{S}[\![\Gamma \vdash \mathtt{fix}\, e : t']\!] = \mathbf{fix}\,(f \circ \mathbf{S}[\![\vdash t' <:_1 t]\!]) = \bigsqcup_{m \in \mathbb{N}} (f \circ \mathbf{S}[\![\vdash t' <:_1 t]\!])^m \bot$$

  which is by definition greater than $(f \circ \mathbf{S}[\![\vdash t' <:_1 t]\!])^n \bot$ for the order of the domain $\mathbf{S}[\![t']\!]$. We obtain $\mathbf{S}[\![\Gamma \vdash \mathtt{fix}\, e : t']\!] \Downarrow_n^{t'}$ by Property 24.

- Case SUB: take $\gamma \Downarrow_n^{\Gamma}$. Then

  $$\begin{array}{rl}
  \gamma \in \Downarrow_n^{\Gamma} & \\
  \mathbf{S}[\![\Gamma \vdash e : t]\!] \ \gamma \in \Downarrow_n^{t} & \text{(ind. hyp. on } \Gamma \vdash e : t) \\
  (\mathbf{S}[\![\vdash t <:_k t']\!] \circ \mathbf{S}[\![\Gamma \vdash e : t]\!])\gamma \in \Downarrow_{n+k}^{t'} & \text{(Lemma 6)} \\
  (\mathbf{S}[\![\vdash t <:_k t']\!] \circ \mathbf{S}[\![\Gamma \vdash e : t]\!])\gamma \in \Downarrow_n^{t'} & \text{(Property 24)}
  \end{array}$$

- Case RESCALE: take $\gamma \Downarrow^{\Gamma}_{n}$. Then

$$\begin{array}{rll} \gamma \in \Downarrow^{\Gamma}_{n} & & \\ \mathbf{S}[\![\vdash \Gamma \downarrow_{ct} \Gamma']\!]\ \gamma \in \Downarrow^{\gamma'}_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)} & & \text{(Lemma 7)} \\ (\mathbf{S}[\![\Gamma' \vdash e : t']\!] \circ \mathbf{S}[\![\vdash \Gamma \downarrow_{ct} \Gamma']\!])\gamma \in \Downarrow^{t'}_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)} & & \text{(ind. hyp. on } \Gamma' \vdash e : t') \\ (\mathbf{S}[\![\vdash t' \uparrow_{ct} t]\!] \circ \mathbf{S}[\![\Gamma' \vdash e : t']\!] \circ \mathbf{S}[\![\vdash \Gamma \downarrow_{ct} \Gamma']\!])\gamma \in \Downarrow^{t}_{n} & & \text{(Lemma 7)} \end{array}$$

The remaining four cases correspond to the stream processing operators whose interpretations were defined using the terms of the untyped semantics through the family $sync_{\Gamma \vdash t}$ of projections. Yet, remember that their untyped semantics was defined in terms of embeddings into $\mathbb{K}$. We will first show that projections from $sync_{\Gamma \vdash t}$ annihilate these embeddings—they compose to the identity. The resulting functions converge as described by Property 25, and use Property 4 and Property 5 to move between streams and clocked streams. We now detail each case.

- Case CONST: by unfolding the definitions in the typed interpretation of this rule, we find

$$\begin{aligned} \mathbf{S}[\![\square \vdash s : dtof(s) :: ct]\!] &= sync_{\square \vdash dtof(s) :: ct} \mathbf{K}[\![op]\!] \\ &= \lambda\_.sync_{dtof(dt) :: ct}(stream(\mathbf{const}\ s)) \\ &= \lambda\_.\mathbf{pack}_{\mathbf{S}[\![ct]\!]}\ (\mathbf{const}\ s) \end{aligned}$$

  Now, for any i we know that $\mathbf{const}\ s \Downarrow_i$ by Property 25. In particular, $\mathbf{const}\ s \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)}$ for any n. Hence, $\mathbf{pack}_{\mathbf{S}[\![ct]\!]}\ (\mathbf{const}\ s) \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)}$ by Property 4, and thus

$$(\lambda\_.\mathbf{pack}_{\mathbf{S}[\![ct]\!]}\ (\mathbf{const}\ s)) \Downarrow^{\square \vdash dtof(s) :: ct}_{n}$$

  which is equivalent to the totality of rule CONST.

- Case OP: unfolding the e-p pairs again, we obtain

$$\begin{aligned} & \mathbf{S}[\![\square \vdash op : (\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct) \multimap (\mathbf{int} :: ct)]\!] \\ = \ & sync_{\square \vdash (\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct) \multimap (\mathbf{int} :: ct)} \mathbf{K}[\![op]\!] \\ = \ & \lambda\_.(sync_{(\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct) \multimap (\mathbf{int} :: ct)}\ f \\ & \qquad \text{where } f = fun(stream \circ \mathbf{map_2}\ [\![op]\!] \circ (unstream \times unstream) \circ unpair) \\ = \ & \lambda\_.\mathbf{pack}_{\mathbf{S}[\![ct]\!]} \circ \mathbf{map_2}\ [\![op]\!] \circ (\mathbf{unpack} \times \mathbf{unpack}) \end{aligned}$$

  Now, suppose that we are given $(xs, ys) \Downarrow^{(\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct)}_{n}$. Then

$$\begin{array}{rcl} (xs, ys) & \in & \Downarrow^{(\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct)}_{n} \\ (\mathbf{unpack} \times \mathbf{unpack})\ (xs, ys) & \in & \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)} \times \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)} \\ (\mathbf{map_2}\ [\![op]\!] \circ (\mathbf{unpack} \times \mathbf{unpack}))(xs, ys) & \in & \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)} \\ (\mathbf{pack}_{\mathbf{S}[\![ct]\!]} \circ \mathbf{map_2}\ [\![op]\!] \circ (\mathbf{unpack} \times \mathbf{unpack}))(xs, ys) & \in & \Downarrow^{\mathbf{int} :: ct}_{n} \end{array}$$

  Thus $\mathbf{pack}_{\mathbf{S}[\![ct]\!]} \circ \mathbf{map_2}\ [\![op]\!] \circ (\mathbf{unpack} \times \mathbf{unpack}) \Downarrow^{(\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct) \multimap (\mathbf{int} :: ct)}_{n}$ for any natural number n, and the totality lemma is verified for this rule.

- Case MERGE: as before, we unfold the definitions and find

$$\begin{aligned}&\mathbf{S}[\![\square \vdash \texttt{merge}\ p : (dt :: ct\ \mathbf{on}\ p) \otimes (dt :: ct\ \mathbf{on}\ \overline{p}) \multimap (dt :: ct)]\!]\\ =\ & sync_{\square \vdash (dt :: ct\ \mathbf{on}\ p)\otimes(dt :: ct\ \mathbf{on}\ \overline{p}) \multimap (dt :: ct)}\mathbf{K}[\![\texttt{merge}\ p]\!]\\ =\ & \lambda\_.\mathbf{pack}_{\mathbf{S}[\![ct]\!]} \circ \mathbf{merge}\ [\![p]\!] \circ (\mathbf{unpack} \times \mathbf{unpack})\end{aligned}$$

  Now, suppose that we are given $(xs, ys) \Downarrow_n^{(dt :: ct\ \mathbf{on}\ p)\otimes(dt :: ct\ \mathbf{on}\ \overline{p})}$. The premise $p \le (1)$ implies $[\![p]\!] \le (1)^\omega$, which we will use below. Then

$$\begin{aligned}(xs, ys) &\in \Downarrow_n^{(dt :: ct\ \mathbf{on}\ p)\otimes(dt :: ct\ \mathbf{on}\ \overline{p})}\\ (\mathbf{unpack} \times \mathbf{unpack})\ (xs, ys) &\in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct\ \mathbf{on}\ p]\!]}(n)} \times \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct\ \mathbf{on}\ \overline{p}]\!]}(n)}\\ (\mathbf{unpack} \times \mathbf{unpack})\ (xs, ys) &\in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct\ \mathbf{on}\ p]\!]}(n)} \times \Downarrow_{n-\mathcal{O}_{\mathbf{S}[\![ct\ \mathbf{on}\ p]\!]}(n)}\\ (\mathbf{merge}\ [\![p]\!] \circ (\mathbf{unpack} \times \mathbf{unpack}))(xs, ys) &\in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)}\\ (\mathbf{pack}_{\mathbf{S}[\![ct]\!]} \circ \mathbf{merge}\ [\![p]\!] \circ (\mathbf{unpack} \times \mathbf{unpack}))(xs, ys) &\in \Downarrow_n^{dt :: ct}\end{aligned}$$

  which concludes the proof of the totality lemma for rule MERGE.

- Case WHEN: we have

$$\begin{aligned}\mathbf{S}[\![\square \vdash \texttt{when}\ p : (dt :: ct) \multimap (dt :: ct\ \mathbf{on}\ p)]\!] &= sync_{\square \vdash (dt :: ct) \multimap (dt :: ct\ \mathbf{on}\ p)}\mathbf{K}[\![\texttt{when}\ p]\!]\\ &= \lambda\_.\mathbf{pack}_{\mathbf{S}[\![ct\ \mathbf{on}\ p]\!]} \circ \mathbf{when}\ [\![p]\!] \circ \mathbf{unpack}\end{aligned}$$

  Now, suppose that we are given $xs \Downarrow_n^{dt :: ct}$. As above, we have $[\![p]\!] \le (1)^\omega$. Then

$$\begin{aligned}xs &\in \Downarrow_n^{dt :: ct}\\ \mathbf{unpack}\ xs &\in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]}(n)}\\ (\mathbf{when}\ [\![p]\!] \circ \mathbf{unpack})xs &\in \Downarrow_{\mathcal{O}_{\mathbf{S}[\![ct]\!]\ on\ [\![p]\!]}(n)}\\ (\mathbf{pack}_{\mathbf{S}[\![ct\ \mathbf{on}\ p]\!]} \circ \mathbf{when}\ [\![p]\!] \circ \mathbf{unpack})xs &\in \Downarrow_n^{dt :: ct\ \mathbf{on}\ p}\end{aligned}$$

  which concludes the proof of the totality lemma for rule WHEN. □

This lemma is a key result on the type system and has a host of immediate consequences. In particular, it implies that all the streams in a program are total, since they converge up to n for any n. Coupled with the definition of the typed semantics, this proves that prefixes of clocks play no role in the semantics of closed programs.

Theorem 4 (Weak Causality)**.** *For any closed expression e, data type dt and clock type ct,*

$$\mathbf{S}[\![\square \vdash e : dt :: ct]\!] \Downarrow_\infty$$

Proof. Immediate by Lemma 8 and the definition of totality. □

Theorem 5 (Soundness)**.** *For any closed expression e, data type dt and clock type ct,*

$$\mathbf{clock}\,\mathbf{S}[\![\square \vdash e : dt :: ct]\!] = \mathbf{S}[\![ct]\!]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \square \text{ value}}}{\mathtt{x} : (0\,2) \vdash \mathtt{x} : (0\,2)}}{\vdash \mathtt{fun}\,\mathtt{x}.\mathtt{x} : (0\,2) \multimap (0\,2)} \qquad \dfrac{\dfrac{}{\vdash (1) <:_0 (0\,2)} \qquad \dfrac{}{\vdash (0\,2) <:_0 0(1)}}{\vdash (0\,2) \multimap (0\,2) <:_0 (1) \multimap 0(1)}}{\vdash \mathtt{fun}\,\mathtt{x}.\mathtt{x} : (1) \multimap 0(1)}$$

(a)

$$\dfrac{\dfrac{\dfrac{}{\vdash \square \text{ value}}}{\mathtt{x} : (0) \multimap (1) \vdash \mathtt{x} : (0) \multimap (1)} \qquad \dfrac{\dfrac{}{\vdash 1(0) <:_0 (0)} \qquad \dfrac{}{\vdash (1) <:_0 (1)}}{\vdash (0) \multimap (1) <:_0 1(0) \multimap (1)}}{\mathtt{x} : (0) \multimap (1) \vdash \mathtt{x} : 1(0) \multimap (1)}$$

(b)

Figure 3.26: Typing derivations where $\mathbf{S}[\![\Gamma \vdash e : t]\!] \circ sync_\Gamma \sqsubset sync_t \circ \mathbf{K}[\![e]\!]$

Proof. The definition of the typed semantics ensures that $\mathbf{clock}\,\mathbf{S}[\![\square \vdash e : dt :: ct]\!] \sqsubseteq \mathbf{S}[\![ct]\!]$ while weak causality implies $\mathbf{clock}\,\mathbf{S}[\![\square \vdash e : dt :: ct]\!] \Downarrow_\infty$. Because the left hand side is maximal in the domain **Ck**, we obtain the equation above. □

These results prove that clock types, which are syntactic objects, provide a precise description of the runtime behavior of clocked streams. In the next section we go one step further however, and relate the typed semantics to the untyped one.

3.4.2 Correspondence Between Typed and Untyped Semantics

We have seen that the typed semantics for the dedicated stream-processing operators was defined in a systematic manner through the use of e-p pairs $(desync_t, sync_t)$. We will now show that this is no accident, and is in fact the case for all expressions. More precisely, we prove the following result, which we call the Lax Coherence Lemma.

Lemma (Lax Coherence)**.** *For any expression e, context* Γ *and type t, one has*

$$\mathbf{S}[\![\Gamma \vdash e : t]\!] \circ sync_\Gamma \;\sqsubseteq\; sync_t \circ \mathbf{K}[\![e]\!] \tag{3.1}$$

Imprecise types Let us discuss briefly cases where inequality 3.1 is strict. First, in the case of a closed expression whose type is a value, Lemma 8 ensures that the left-hand side is maximal in the domain $\mathbf{S}[\![dt :: ct]\!]$, and thus that the two sides are equal. The inequality can only be strict when the expression contains free variables or has a non-value type. We give two examples where this actually happens.

Example 18. The typing derivation for the first example is given in Figure 3.4.2 (a). It assigns to the identity function the type $(0\,2) \multimap (0\,2)$, and then adapts it to the type $(1) \multimap 0(1)$.

Here, the left-hand side of the inequality gives rise to a function which, applied to the partial stream $[42].\bot$, computes the stream $[].\bot$.

$$\begin{array}{ll} & \mathbf{S}[\![\vdash \mathtt{fun\,x.x} : (1) \multimap 0(1)]\!](\mathit{sync}_{\square}\bot)([42].\bot) \\ = & (\mathit{sync}_{0(1)} \circ \mathit{desync}_{(0\,2)} \circ \mathit{sync}_{(0\,2)} \circ \mathit{desync}_{(1)})[42].\bot \\ = & (\mathit{sync}_{0(1)} \circ \mathit{desync}_{(0\,2)} \circ \mathit{sync}_{(0\,2)})42.\bot \\ = & (\mathit{sync}_{0(1)} \circ \mathit{desync}_{(0\,2)})[].\bot \\ = & [].\bot \end{array}$$

In contrast, the right-hand side maps the same stream to the strictly larger result $[].[42].\bot$.

$$\begin{array}{lll} (\mathit{sync}_{(1)\multimap 0(1)} \circ \mathbf{K}[\![\mathtt{fun\,x.x}]\!])\ \bot\ ([42].\bot) & = & (\mathit{sync}_{0(1)} \circ \mathit{desync}_{(1)})[42].\bot \\ & = & \mathit{sync}_{0(1)}42.\bot \\ & = & [].[42].\bot \end{array}$$

Thus the typed semantics, applied to synchronized inputs, produces less outputs than the synchronization of the typed semantics.

Example 19. The typing derivation for the second example is given in Figure 3.4.2 (b). It uses the fact that $(0) \multimap (0)$ is adaptable to $1(0) \multimap 1(0)$. Now, consider the following function *hd*:

$$\begin{array}{lll} \mathit{hd} & \in & \mathit{Stream}(\mathbb{V}) \Rightarrow_c \mathit{Stream}(\mathbb{V}) \\ \mathit{hd}\,(x.xs) & = & x.\bot \end{array}$$

which maps a stream to a non-$\bot$ value if and only if this stream converges for at least one step. Consider the environment $\sigma = \bot[x \mapsto \mathit{fun}(\mathit{stream} \circ \mathit{hd} \circ \mathit{unstream})]$. Then, the left hand side denotes the function $\lambda xs.\bot$, as we show below.

$$\begin{array}{ll} & \mathbf{S}[\![\mathtt{x} : (0) \multimap 1(0) \vdash \mathtt{x} : 1(0) \multimap 1(0)]\!]\ (\mathit{desync}_{\mathtt{x}:(0)\multimap 1(0)}\ \sigma) \\ = & (\mathit{sync}_{1(0)\multimap 1(0)} \circ \mathit{desync}_{(0)\multimap 1(0)})\ \mathit{hd} \\ = & \lambda xs.(\mathit{sync}_{1(0)} \circ \mathit{desync}_{1(0)} \circ \mathit{hd} \circ \mathit{sync}_{(0)} \circ \mathit{desync}_{1(0)})\ xs \\ = & \lambda xs.(\mathit{hd} \circ \mathit{sync}_{(0)} \circ \mathit{desync}_{1(0)})\ xs \\ = & \lambda xs.\mathit{hd}\,[]^{\omega} \\ = & \lambda xs.\bot \end{array}$$

In contrast, the right-hand side denotes the function given below.

$$\begin{array}{lll} \mathit{sync}_{1(0)\multimap 1(0)}(\mathbf{K}[\![\mathtt{x}]\!]\sigma) & = & \mathit{sync}_{1(0)\multimap 1(0)}\ \mathit{hd} \\ & = & \lambda xs.\mathit{sync}_{1(0)}\ (\mathit{hd}\,(\mathit{desync}_{1(0)}\ xs)) \\ & = & \lambda([x].\_).([x].\bot) \end{array}$$

This function differs from $\lambda xs.\bot$ since, for instance, it maps $[42].\bot$ to $[42].\bot$.

Observe that, in both examples, inequality 3.1 is strict in derivations that assign *imprecise* types to the expressions involved. For instance, assigning the type $(1) \multimap 0(1)$ to the identity function in the first example is clearly sub-optimal—this type makes the first element of the output stream falsely depend on the second element of the input stream. The same is true in the second derivation.

$$\boxed{\gamma \equiv_{\Gamma;S} \gamma'}$$

$$\textsc{CtxEqEmpty}\ \frac{}{\bot \equiv_{\square;S} \bot} \qquad \textsc{CtxEqSame}\ \frac{\gamma \equiv_{\Gamma;S} \gamma'}{(\gamma, v) \equiv_{\Gamma,x:t;S\cup\{x\}} (\gamma', v)} \qquad \textsc{CtxEqDiff}\ \frac{\gamma \equiv_{\Gamma;S} \gamma'}{(\gamma, v) \equiv_{\Gamma,x:t;S\setminus\{x\}} (\gamma', v')}$$

Figure 3.27: Context equivalence

Remark 16. The fact that the equality does not always hold comes from a defect in our typed semantics. For any types t and t' such that $\vdash t <:_0 t'$ holds, we would like the following to hold.

$$(\mathit{sync}_{t'} \circ \mathit{desync}_t,\ \mathit{sync}_t \circ \mathit{desync}_{t'}) \quad : \quad \mathbf{S}[\![t]\!] \triangleleft \mathbf{S}[\![t']\!]$$

This would confirm the intuition that "small" types characterize untyped computations in a more precise way than "large" ones. Unfortunately, this does not hold, as Example 18 gives the counter-example $\vdash (0\ 2) \multimap (0\ 2) <:_0 (1) \multimap 0(1)$. This comes from the fact that our semantics is *not strict enough*. We discuss this defect more precisely in Section 6.2.

Scoping issues We shall now prove the Lax Coherence Lemma. Part of the result expresses that the notion of scoping between the two languages coincide. The untyped semantics uses environments, which enforce lexical scoping by construction since $\sigma[x \mapsto v][x \mapsto v']$ is equal to $\sigma[x \mapsto v']$. On the other hand, the typed semantics manage the values of free variables as tuples, with a derivation of $x : t, x : t \vdash x : t$ interpreted as a binary function. If the typed semantics is to respect lexical scope, the result of its interpretation should not depend on its left argument since it corresponds to a binding of x that has been shadowed. More generally, we shall prove that the result of $\mathbf{S}[\![\Gamma \vdash e : t]\!]$ depends only on the rightmost binding of each variable in Γ.

This scoping issue comes from the fact that $(\mathit{desync}_\Gamma, \mathit{sync}_\Gamma)$ is not an e-p pair. While $\mathit{sync}_\Gamma \circ \mathit{desync}_\Gamma$ holds, we do not have $\mathit{sync}_\Gamma \circ \mathit{desync}_\Gamma$ in general. However, we prove that $\mathbf{S}[\![\Gamma \vdash e : t]\!] = \mathbf{S}[\![\Gamma \vdash e : t]\!] \circ \mathit{sync}_\Gamma \circ \mathit{desync}_\Gamma$ holds. This equation expresses formally that the typed semantics respects lexical scope.

To establish this result, we define an equivalence relation expressing that two inhabitants of a given context are *lexically identical*, and prove that the typed semantics respect this result.

Definition 17 (Lexical equivalence)**.** *Given a context Γ and a finite set of variables S, the relation $\gamma \equiv_{\Gamma;S} \gamma'$ between $\gamma, \gamma' \in \mathbf{S}[\![\Gamma]\!]$ expresses that, for each variable $x \in S$, the values corresponding to the rightmost binding in γ and in γ' are the same. It is defined as an inductive system of rules in Figure 3.27.*

We now prove a series of technical properties on the relation that will serve as building blocks for subsequent proofs.

Property 26. *Given context Γ and finite set $S \subseteq \mathit{Var}$, the relation $\equiv_{\Gamma;S}$ is an equivalence relation.*

Property 27. *Given context* Γ*, finite set* $S \subseteq Var$ *and inhabitants* $\gamma, \gamma' \in \mathbf{S}[\![\Gamma]\!]$*, one has*

1. $(\gamma, v) \equiv_{\Gamma, x:t; S} (\gamma', v')$ *and* $x \notin S$ *implies* $\gamma \equiv_{\Gamma; S} \gamma'$ *for any* $v, v' \in \mathbf{S}[\![t]\!]$*;*
2. $(\gamma, v) \equiv_{\Gamma, x:t; S} (\gamma', v')$ *and* $x \in S$ *implies* $v = v'$ *for any* $v, v' \in \mathbf{S}[\![t]\!]$*;*
3. $(\gamma, v) \equiv_{\Gamma, x:t; S} (\gamma', v')$ *and* $x \in S$ *implies* $\gamma \equiv_{\Gamma; S \setminus \{x\}} \gamma'$*;*
4. $\gamma \equiv_{\Gamma; S} \gamma'$ *implies* $\gamma \equiv_{\Gamma; S'} \gamma'$ *for any* $S' \subseteq S$*.*

Proof. Each statement can be proved by induction over Γ. The first three are immediate, but we detail the fourth one.

- Case $\Box$: immediate.
- Case $\Gamma, x : t$: this case consists in proving $(\gamma, v) \equiv_{\Gamma, x:t; S'} (\gamma', v')$, assuming $(\gamma, v) \equiv_{\Gamma, x:t; S} (\gamma', v')$. We reason by case on whether x belongs to S and S'.
  - Case $x \notin S$: then $x \notin S'$ since $S' \subseteq S$. In this case the first statement gives $\gamma \equiv_{\Gamma; S} \gamma'$, and thus by induction hypothesis we have $\gamma \equiv_{\Gamma; S'} \gamma'$. Since $S' \setminus \{x\} = S'$, we conclude by rule CTXEQDIFF.
  - Case $x \in S$: the second statement gives $v = v'$ and $\gamma \equiv_{\Gamma; S''} \gamma'$ for some finite set S'' such that $S = S'' \cup \{x\}$. Now, if $x \in S'$, then there is S''' such that $S' = S''' \cup \{x\}$ and $S''' \subseteq S'$. Thus the induction hypothesis gives $\gamma \equiv_{\Gamma; S'''} \gamma'$ and by rule CTXEQSAME and the fact that $v = v'$ we get $(\gamma, v) \equiv_{\Gamma; S'' \cup \{x\}} (\gamma', v')$ and thus finish with $(\gamma, v) \equiv_{\Gamma; S'} (\gamma', v')$. If $x \notin S'$, we have $S' \subseteq S''$ and thus $\gamma \equiv_{\Gamma; S'} \gamma'$. Since $S' \setminus \{x\} = S'$, we conclude by rule CTXEQDIFF. $\Box$

Property 28. *For any* $\Gamma, \gamma, \gamma' \in \mathbf{S}[\![\Gamma]\!]$*,* S *and* $x \notin dom(\Gamma)$*, if* $\gamma \equiv_{\Gamma; S} \gamma'$ *then* $\gamma \equiv_{\Gamma; S \cup \{x\}} \gamma'$*.*

Proof. By induction over Γ.

- Case $\Box$: immediate.
- Case $\Gamma, y : t$: we have $(\gamma, v) \equiv_{\Gamma, y:t; S} (\gamma', v')$ and $x \neq y$. Reason by case on whether $y \in S$.
  - Case $y \notin S$: by Property 27 (1) we have $\gamma \equiv_{\Gamma; S} \gamma'$ and thus by induction $\gamma \equiv_{\Gamma; S \cup \{x\}} \gamma'$ since $dom(\Gamma) \subseteq dom(\Gamma, y : t)$. Rule CTXEQDIFF gives $(\gamma, v) \equiv_{\Gamma, y:t; S \cup \{x\} \setminus \{y\}} (\gamma', v')$ which concludes the proof since $y \notin S$ and $x \neq y$ implies $S \cup \{x\} \setminus \{y\} = S \cup \{x\}$.
  - Case $y \in S$: by Property 27 (2) and (3) we have $v = v'$ and $\gamma \equiv_{\Gamma; S \setminus \{y\}} \gamma'$. By induction we have $\gamma \equiv_{\Gamma; S \setminus \{y\} \cup \{x\}} \gamma'$ and thus we can conclude by rule CTXEQSAME. $\Box$

One important point is that while $(desync_\Gamma, sync_\Gamma)$ is not an e-p pair because $sync_\Gamma \circ desync_\Gamma$ is not the identity in general, it is *up to lexical equivalence*. We establish this result proving an intermediate lemma first.

Property 29. *For any* Γ*,* $\gamma \in \mathbf{S}[\![\Gamma]\!]$*,* $\sigma \in Env(\mathbb{K})$*,* $v \in \mathbb{K}$*,* S *and* $x \notin S$ *such that* $\gamma \equiv_{\Gamma; S} sync_\Gamma \sigma$*, we have* $\gamma \equiv_{\Gamma; S} sync_\Gamma \sigma[x \mapsto v]$*.*

Proof. By induction over Γ.

- Case $\square$: immediate since $sync_{\square}\sigma[x \mapsto v] = \bot \equiv_{\square;S} \bot$ for any S.
- Case $\Gamma, y : t$: we shall prove $(\gamma, v') \equiv_{\Gamma,y:t;S} (sync_{\Gamma}\sigma[x \mapsto v], sync_t\sigma[x \mapsto v](y))$ assuming $(\gamma, v') \equiv_{\Gamma,y:t;S} (sync_{\Gamma}\sigma, sync_t\sigma(y))$. We will reason by case on whether y belongs to S; notice that when it does not we have $x \neq y$ since $x \in S$.
  - Case $y \notin S$: then $y \notin S$ and by Property 27 (1) we have $\gamma \equiv_{\Gamma;S} sync_{\Gamma}\sigma$ and thus by induction $\gamma \equiv_{\Gamma;S} sync_{\Gamma}\sigma[x \mapsto v]$. Applying rule CTXEQDIFF, we get $(\gamma, v') \equiv_{\Gamma,y:t;S\setminus\{y\}} (sync_{\Gamma}\sigma[x \mapsto v], sync_t\sigma[x \mapsto v](y))$ which concludes this case since $S \setminus \{x\} = S$.
  - Case $y \in S$: by Property 27 (2) and $x \neq y$ we have $v' = sync_t\sigma(y) = sync_t\sigma[x \mapsto v](y)$. Define S' as $S \setminus \{y\}$; we have $\gamma \equiv_{\Gamma;S'} sync_t\sigma$ by Property 27 (3), and thus by induction $\gamma \equiv_{\Gamma;S'} sync_t\sigma[y \mapsto v]$. We conclude this case using rule CTXEQSAME. □

Property 30. *For any Γ, $\gamma \in \mathbf{S}[\![\Gamma]\!]$ and S, we have $\gamma \equiv_{\Gamma;S} sync_{\Gamma}(desync_{\Gamma}\gamma)$.*

Proof. By induction over Γ.

- Case $\square$: immediate since $sync_{\square}\sigma[x \mapsto v] = \bot \equiv_{\square;S} \bot$ for any S.
- Case $\Gamma, x : t$: we have $sync_{\Gamma,x:t}(desync_{\Gamma,x:t}(\gamma, v)) = (sync_{\Gamma}(desync_{\Gamma}\gamma)[x \mapsto desync_t v], v)$ and hence we shall prove $(\gamma, v) \equiv_{\Gamma,x:t;S} (sync_{\Gamma}(desync_{\Gamma}\gamma)[x \mapsto desync_t v], v)$. Now, define S' as $S \setminus \{x\}$; by induction, we have $\gamma \equiv_{\Gamma;S'} sync_{\Gamma}(desync_{\Gamma}\sigma)$. Thus, and since by definition $x \notin S'$, from Property 3.4.2 we deduce $\gamma \equiv_{\Gamma;S'} sync_{\Gamma}(desync_{\Gamma})[x \mapsto v]$. Applying rule CTXEQSAME we obtain $(\gamma, v) \equiv_{\Gamma;S'\cup\{x\}} (sync_{\Gamma}(desync_{\Gamma})[x \mapsto v], v)$ which concludes the proof since $S' \cup \{x\} = S$. □

The next lemma shows that the typed semantics respects lexical equivalence and will be instrumental in proving the Lax Coherence Lemma. As usual, we need to prove a series of properties on auxiliary judgments.

Property 31. *The interpretation of a separation judgment preserves lexical-relatedness. In other words, given contexts Γ, Γ_1, Γ_2 and inhabitants $\gamma, \gamma' \in \mathbf{S}[\![\Gamma]\!]$, for any $d ::: (\Gamma \vdash \Gamma_1 \otimes \Gamma_2)$, taking $(\gamma_1, \gamma_2) = \mathbf{S}[\![d ::: (\Gamma \vdash \Gamma_1 \otimes \Gamma_2)]\!]\gamma$ and $(\gamma'_1, \gamma'_2) = \mathbf{S}[\![d ::: (\Gamma \vdash \Gamma_1 \otimes \Gamma_2)]\!]\gamma'$, then for any finite set S, $\gamma \equiv_{\Gamma;S} \gamma'$ implies*

$$\gamma_1 \equiv_{\Gamma_1;S} \gamma'_1 \quad \textit{and} \quad \gamma_2 \equiv_{\Gamma_1;S} \gamma'_2$$

Proof. By induction over typing derivations.

- Case SEPEMPTY: immediate, inhabitants of empty contexts are always lexically related.
- Case SEPCONTRACT: we prove $(\gamma_i, v) \equiv_{\Gamma_i,x:t;S} (\gamma'_i, v')$ assuming $(\gamma, v) \equiv_{\Gamma,x:t;S} (\gamma', v')$. We reason by case on whether $x \in S$.
  - Case $x \notin S$: we apply the induction hypothesis to obtain $\gamma_i \equiv_{\Gamma_i;S} \gamma'_i$. Through rule CTXEQDIFF we have $(\gamma_i, v) \equiv_{\Gamma_i;S\setminus\{x\}} (\gamma'_i, v')$ which concludes since $S \setminus \{x\} = S$.

  - Case $x \in S$: we apply the induction hypothesis to obtain $\gamma_i \equiv_{\Gamma_i; S \setminus \{x\}} \gamma'_i$. By Property 27 (2) we have $v = v'$, hence by CTXEQSAME we have $(\gamma_i, v) \equiv_{\Gamma_i; S \setminus \{x\} \cup \{x\}} (\gamma'_i, v)$ concluding the proof.

- Case SEPLEFT: we shall prove $(\gamma_1, v) \equiv_{\Gamma_1, x:t; S} (\gamma'_1, v')$ and $\gamma_2 \equiv_{\Gamma_2; S} \gamma'_2$ assuming as before $(\gamma, v) \equiv_{\Gamma, x:t; S} (\gamma', v')$ and knowing $x \notin dom(\Gamma_2)$. We reason by case on whether $x \in S$.
  - Case $x \notin S$: we apply the induction hypothesis to obtain $\gamma_i \equiv_{\Gamma_i; S} \gamma'_i$. As in the proof of case SEPCONTRACT, we conclude $(\gamma_1, v) \equiv_{\Gamma_1, x:t; S} (\gamma'_1, v')$ using rule CTXEQDIFF (since $S = S \setminus \{x\}$).
  - Case $x \in S$: we apply the induction hypothesis to obtain $\gamma_i \equiv_{\Gamma_i; S \setminus \{x\}} \gamma'_i$. As before, by Property 27 (2) and rule CTXEQSAME we prove $(\gamma_1, v) \equiv_{\Gamma_1, x:t; S} (\gamma'_1, v')$. We obtain $\gamma_2 \equiv_{\Gamma_2; S} \gamma'_2$ using $x \notin dom(\Gamma_2)$ and Property 28.

- Case SEPRIGHT: symmetric to SEPLEFT. □

Property 32. *For any* $\Gamma, \Gamma', ct, d ::: (\vdash \Gamma \downarrow_{ct} \Gamma')$, $\gamma, \gamma' \in \mathbf{S}[\![\Gamma]\!]$ *and* S *such that* $\gamma \equiv_{\Gamma; S} \gamma'$, *one has*

$$\mathbf{S}[\![d ::: (\vdash \Gamma \downarrow_{ct} \Gamma')]\!]\gamma \equiv_{\Gamma'; S} \mathbf{S}[\![d ::: (\vdash \Gamma \downarrow_{ct} \Gamma')]\!]\gamma'$$

Proof. The proof is immediate by induction over Γ and follows the usual pattern of reasoning by case on whether $x \in S$ for the case $\Gamma, x : t$. □

Lemma 9. *For any* $\Gamma, e, t, d ::: (\Gamma \vdash e : t)$ *and* $\gamma, \gamma' \in \mathbf{S}[\![\Gamma]\!]$ *such that* $\gamma \equiv_{\Gamma; FV(e)} \gamma'$, *one has*

$$\mathbf{S}[\![d ::: (\Gamma \vdash e : t)]\!]\gamma = \mathbf{S}[\![d ::: (\Gamma \vdash e : t)]\!]\gamma'$$

Proof. We proceed by induction on the typing derivation p.

- Case VAR: we have $(\gamma, v) \equiv_{\Gamma, x:t; FV(x)} (\gamma, v')$ and thus, since $x \in FV(x)$, $v = v'$ by Property 27 (2). This proves $\mathbf{S}[\![\Gamma, x : t \vdash x : t]\!](\gamma, v) = \mathbf{S}[\![\Gamma, x : t \vdash x : t]\!](\gamma', v')$.

- Case WEAKEN: we have $(\gamma, v) \equiv_{\Gamma, x:t; FV(e)} (\gamma, v')$ and thus, since $x \notin FV(e)$, $\gamma \equiv_{\Gamma; FV(e)} \gamma'$ by Property 27 (1). We conclude applying the induction hypothesis.

- Case LAMBDA: we shall prove that for any v, one has

$$\begin{array}{rrcl} & \mathbf{S}[\![\Gamma \vdash \mathtt{fun}\, x.e : t \multimap t']\!]\ \gamma\ v & = & \mathbf{S}[\![\Gamma \vdash \mathtt{fun}\, x.e : t \multimap t']\!]\ \gamma'\ v \\ \Leftrightarrow & \mathbf{S}[\![\Gamma, x : t \vdash e : t']\!]\ (\gamma, v) & = & \mathbf{S}[\![\Gamma, x : t \vdash e : t']\!]\ (\gamma', v) \end{array}$$

  We have $\gamma \equiv_{\Gamma; FV(e) \setminus \{x\}} \gamma'$, hence by EQCTXSAME we obtain $(\gamma, v) \equiv_{\Gamma, x:t; FV(e)} (\gamma', v)$ which is enough to conclude using the induction hypothesis.

- Case APP, PAIR and LETPAIR: all three cases are similar, we first apply Property 31 and then induction hypotheses.

- Case FIX and SUB: we directly apply the induction hypothesis since γ and γ' are passed to the rest of the derivation without change.

- Case CONST, OP, MERGE and WHEN: immediate since the context is empty.
- Case RESCALE: we first apply Property 32 and then the induction hypothesis. □

We can now combine Property 30 and Lemma 9 to prove that $(desync_\Gamma, sync_\Gamma)$ acts as an e-p pair from the point of view of the typed semantics.

Lemma 10. *For any* Γ, e *and* t *and* $d ::: (\Gamma \vdash e : t)$, *one has*

$$\mathbf{S}[\![d ::: (\Gamma \vdash e : t)]\!] = \mathbf{S}[\![d ::: (\Gamma \vdash e : t)]\!] \circ sync_\Gamma \circ desync_\Gamma$$

This shows that the typed semantics follows lexical scoping rules, a result we will now use to prove the Lax Coherence Lemma.

Lax Coherence We proceed as usual in order to establish the inequation explained before, showing intermediate results on auxiliary judgments.

Property 33. *For any* Γ, Γ_1 *and* Γ_2, *one has*

$$\mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!] \circ sync_\Gamma = \langle sync_{\Gamma_1}, sync_{\Gamma_2} \rangle$$

Proof. Routine induction over derivations. Unfolding definitions and applying induction hypotheses is enough to prove all four cases. For instance, case of SEPCONTRACT goes as

$$\begin{array}{ll} & \mathbf{S}[\![\Gamma, x : t \vdash \Gamma_1, x : t \otimes \Gamma_2, x : t]\!] \circ sync_{\Gamma, x:t} \\ = & \lambda\sigma.((\gamma_1, sync_t\sigma(x)), (\gamma_2, sync_t\sigma(x))) \text{ where } (\gamma_1, \gamma_2) = \mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!]\ (sync_\Gamma\sigma) \\ = & \lambda\sigma.((sync_{\Gamma_1}\sigma, sync_t\sigma(x)), (sync_{\Gamma_2}\sigma, sync_t\sigma(x))) \\ = & \langle sync_{\Gamma_1, x:t}, sync_{\Gamma_2, x:t} \rangle \end{array}$$

□

Property 34. *For any* t, t' *and* k, *one has*

$$\begin{array}{lll} \mathbf{S}[\![\vdash t <:_k t']\!] \circ sync_t & \sqsubseteq & sync_{t'} \\ desync_{t'} \circ \mathbf{S}[\![\vdash t <:_k t']\!] & \sqsubseteq & desync_t \end{array}$$

Proof. First, remark that the first inequation actually implies the second since

$$\begin{array}{lll} & \mathbf{S}[\![\vdash t <:_k t']\!] \circ sync_t \sqsubseteq sync_{t'} & \\ \Rightarrow & desync_{t'} \circ \mathbf{S}[\![\vdash t <:_k t']\!] \circ sync_t \sqsubseteq desync_{t'} \circ sync_{t'} & (desync_{t'} \circ \_ \text{ continuous}) \\ \Rightarrow & desync_{t'} \circ \mathbf{S}[\![\vdash t <:_k t']\!] \circ sync_t \sqsubseteq id & (desync_{t'} \circ sync_{t'} \sqsubseteq id) \\ \Rightarrow & desync_{t'} \circ \mathbf{S}[\![\vdash t <:_k t']\!] \sqsubseteq desync_t & (sync_t \circ desync_t = id) \end{array}$$

Thus we only prove the first by induction over derivations of the adaptability judgment.

- Case ADAPTSTREAM: one has

$$\begin{array}{lll} & desync_{dt :: ct} \circ sync_{dt :: ct} \sqsubseteq id & ((desync_t, sync_t) \text{ e-p pair}) \\ \Rightarrow & sync_{dt :: ct'} \circ desync_{dt :: ct} \circ sync_{dt :: ct} \sqsubseteq sync_{dt :: ct'} & (sync_{dt :: ct'} \circ \_ \text{ continuous}) \\ \Leftrightarrow & \mathbf{S}[\![\vdash dt :: ct <:_k dt :: ct']\!] \circ sync_{dt :: ct} \sqsubseteq sync_{dt :: ct'} & (\text{def. } \mathbf{S}[\![\vdash dt :: ct <:_k dt :: ct']\!]) \end{array}$$

- Case ADAPTPAIR: one has

$$\begin{array}{lll} & \mathbf{S}[\![\vdash t_1 \otimes t_2 <:_k t_1' \otimes t_2']\!] \circ sync_{t_1 \otimes t_2} & \\ = & (\mathbf{S}[\![\vdash t_1 <:_k t_1']\!] \times \mathbf{S}[\![\vdash t_2 <:_k t_2']\!]) \circ sync_{t_1 \otimes t_2} & (\text{def. } \mathbf{S}[\![\vdash t_1 \otimes t_2 <:_k t_1' \otimes t_2']\!]) \\ = & (\mathbf{S}[\![\vdash t_1 <:_k t_1']\!] \circ sync_{t_1}) \times (\mathbf{S}[\![\vdash t_2 <:_k t_2']\!] \circ sync_{t_2}) & (\text{def. } sync_{t_1' \otimes t_2'}) \\ \sqsubseteq & sync_{t_1'} \times sync_{t_2'} & (\text{ind. hyp.}) \\ = & sync_{t_1' \otimes t_2'} & (\text{def. } sync_{t_1' \otimes t_2'}) \end{array}$$

- Case ADAPTFUN: for any $f \in \mathbf{S}[\![t_1 \multimap t_2]\!]$, one has

$$\begin{array}{lll} & (\mathbf{S}[\![\vdash t_1 \multimap t_2 <:_k t_1' \multimap t_2']\!] \circ sync_{t_1 \multimap t_2}) f & \\ = & \mathbf{S}[\![\vdash t_1 \multimap t_2 <:_k t_1' \multimap t_2']\!](sync_{t_2} \circ f \circ desync_{t_1}) & (\text{def. } sync_{t_1 \multimap t_2}) \\ = & \mathbf{S}[\![\vdash t_2 <:_k t_2']\!] \circ sync_{t_2} \circ f \circ desync_{t_1} \circ \mathbf{S}[\![\vdash t_1' <:_k t_1]\!] & (\text{def. } \mathbf{S}[\![\vdash t_1 \multimap t_2 <:_k t_1' \multimap t_2']\!]) \\ \sqsubseteq & sync_{t_2'} \circ f \circ desync_{t_1'} & (\text{ind. hyp.}) \\ = & sync_{t_1' \multimap t_2'} f & (\text{def. } sync_{t_1' \multimap t_2'}) \end{array}$$

  which uses the statement $desync_{t_1} \circ \mathbf{S}[\![\vdash t_1' <:_k t_1]\!] \sqsubseteq desync_{t_1'}$ proved above. □

Property 35. *For any t, t' and ct, one has*

$$\begin{array}{lll} \mathbf{S}[\![\vdash t \uparrow_{ct} t']\!] \circ sync_t & \sqsubseteq & sync_{t'} \\ \mathbf{S}[\![\vdash t \downarrow_{ct} t']\!] \circ sync_t & \sqsubseteq & sync_{t'} \\ \mathbf{S}[\![\vdash \Gamma \downarrow_{ct} \Gamma']\!] \circ sync_\Gamma & \sqsubseteq & sync_{\Gamma'} \end{array}$$

Proof. By mutual induction on derivations of $\vdash t \uparrow_{ct} t'$ and $\vdash t \downarrow_{ct} t'$. The proof is routine. Cases handling type formers, including UPSTREAM and DOWNSTREAM, can be proved as in the previous property. We only detail some of the other cases.

- Case UPON: one has

$$\begin{array}{lll} \mathbf{S}[\![\vdash t \uparrow_{ct \,\mathbf{on}\, ct'} t']\!] \circ sync_t = \mathbf{S}[\![\vdash t'' \uparrow_{ct} t']\!] \circ \mathbf{S}[\![\vdash t \uparrow_{ct'} t'']\!] \circ sync_t & & (\text{def. } \mathbf{S}[\![\vdash t \uparrow_{ct \,\mathbf{on}\, ct'} t']\!]) \\ \sqsubseteq \mathbf{S}[\![\vdash t'' \uparrow_{ct} t']\!] \circ sync_{t''} & & (\text{ind. hyp.}) \\ \sqsubseteq sync_{t'} & & (\text{ind. hyp.}) \end{array}$$

All remaining cases are similar to the ones discussed above. □

Property 36. *For any type t, function $f : \mathbb{K} \Rightarrow_c \mathbb{K}$ and $n \geq 0$, one has*

$$(sync_{t \multimap t} f)^n \bot \quad \sqsubseteq \quad sync_t(f^n \bot)$$

Finally, as a last step before proving the Lax Coherence Lemma itself, we need a dedicated property used in the case of fixpoints.

Property 37. *For any type t and function $f : \mathbb{K} \Rightarrow_c \mathbb{K}$, one has*

$$\mathbf{fix}\,(sync_{t \multimap t} f) \quad \sqsubseteq \quad sync_t(\mathbf{fix}\, f)$$

Proof. Consider the predicate $P(x)$ defined by $x \sqsubseteq sync_t(\mathbf{fix}\, f)$. It is admissible, and thus we can use Scott induction. Now, for any $x \in \mathbb{K}$, assuming $P(x)$ one has

$$\begin{array}{rcll}
(sync_{t \multimap t} f)x & = & (sync_t \circ f \circ desync_t)x & (\text{def. } sync_{t \multimap t}) \\
& \sqsubseteq & (sync_t \circ f \circ desync_t \circ sync_t)(\mathbf{fix}\, f) & (P(x) \Leftrightarrow x \sqsubseteq sync_t(\mathbf{fix}\, f)) \\
& \sqsubseteq & (sync_t \circ f)(\mathbf{fix}\, f) & (desync_t \circ sync_t \sqsubseteq id) \\
& = & sync_t(\mathbf{fix}\, f) & (f(\mathbf{fix}\, f) = \mathbf{fix}\, f)
\end{array}$$

and thus $P((sync_{t \multimap t} f)x)$. □

We now reach the final lemma stated in the beginning of this subsection. The proof will use all the properties stated up to now.

Lemma 11 (Lax Coherence)**.** *For any Γ, e, t, one has*

$$\mathbf{S}[\![\Gamma \vdash e : t]\!] \circ sync_\Gamma \quad \sqsubseteq \quad sync_t \circ \mathbf{K}[\![e]\!]$$

Proof. By induction on typing derivations of $\Gamma \vdash e : t$.

- Case VAR: one has

$$\begin{array}{rcll}
 & & \mathbf{S}[\![\Gamma, x : t \vdash x : t]\!] \circ sync_{\Gamma, x:t} & \\
& = & \lambda\sigma.\mathbf{S}[\![\Gamma, x : t \vdash x : t]\!](sync_\Gamma \sigma, sync_t \sigma(x)) & (\text{def. } sync_{\Gamma,x:t}) \\
& = & \lambda\sigma.sync_t \sigma(x) & (\text{def. } \mathbf{S}[\![\Gamma, x : t \vdash x : t]\!]) \\
& = & sync_t \circ \lambda\sigma.\sigma(x) & \\
& = & sync_t \circ \mathbf{K}[\![x]\!] & (\text{def. } \mathbf{K}[\![x]\!])
\end{array}$$

- Case WEAKEN: one has

$$\begin{array}{rcll}
\mathbf{S}[\![\Gamma, x : t \vdash e : t']\!] \circ sync_{\Gamma,x:t} & = & \mathbf{S}[\![\Gamma \vdash e : t']\!] \circ \pi_l \circ sync_{\Gamma,x:t} & (\text{def. } \mathbf{S}[\![\Gamma, x : t \vdash x : t]\!]) \\
& = & \mathbf{S}[\![\Gamma \vdash e : t']\!] \circ sync_\Gamma & (\text{def. } sync_{\Gamma,x:t}) \\
& \sqsubseteq & sync_t \circ \mathbf{K}[\![e]\!] & (\text{ind. hyp.})
\end{array}$$

- Case LAMBDA: one has

$$\begin{array}{rcll}
 & & \mathbf{S}[\![\Gamma \vdash \mathtt{fun}\, x.e : t \multimap t']\!] \circ sync_\Gamma & \\
& = & \lambda\sigma.\lambda v.\mathbf{S}[\![\Gamma, x : t \vdash e : t']\!](sync_\Gamma \sigma, v) & (\text{def.}) \\
& = & \lambda\sigma.\lambda v.(\mathbf{S}[\![\Gamma, x : t \vdash e : t']\!] \circ sync_{\Gamma,x:t} \circ desync_{\Gamma,x:t})(sync_\Gamma \sigma, v) & (\text{Lemma 10}) \\
& \sqsubseteq & \lambda\sigma.\lambda v.(sync_{t'} \circ \mathbf{K}[\![e]\!])(desync_\Gamma(sync_\Gamma \sigma))[x \mapsto desync_t v] & (\text{ind. hyp., def.}) \\
& = & (\lambda\sigma.\lambda v.(sync_{t'} \circ \mathbf{K}[\![e]\!])\sigma[x \mapsto desync_t v])) \circ desync_\Gamma \circ sync_\Gamma & \\
& \sqsubseteq & \lambda\sigma.\lambda v.(sync_{t'} \circ \mathbf{K}[\![e]\!])\sigma[x \mapsto desync_t v] & (\text{Property 20}) \\
& = & sync_{t \multimap t'} \circ \lambda\sigma.\lambda v.\mathbf{K}[\![e]\!]\sigma[x \mapsto v] & (\text{def.}) \\
& = & sync_{t \multimap t'} \circ \mathbf{K}[\![\mathtt{fun}\, x.e]\!] & (\text{def.})
\end{array}$$

- Case APP: one has

$$\begin{array}{lll}
& \mathbf{S}[\![\Gamma \vdash e\ e' : t']\!] \circ sync_\Gamma & \\
= & \lambda\sigma.\mathbf{S}[\![\Gamma_1 \vdash e : t \multimap t']\!]\gamma_1(\mathbf{S}[\![\Gamma_2 \vdash e' : t]\!]\gamma_2) & \text{(def. } \mathbf{S}[\![\Gamma \vdash e\ e' : t']\!]\text{)} \\
& \quad \text{where } (\gamma_1, \gamma_2) = \mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!](sync_\Gamma\sigma) & \\
= & \lambda\sigma.\mathbf{S}[\![\Gamma_1 \vdash e : t \multimap t']\!](sync_{\Gamma_1}\sigma)(\mathbf{S}[\![\Gamma_2 \vdash e' : t]\!](sync_{\Gamma_2}\sigma)) & \text{(Property 35)} \\
\sqsubseteq & \lambda\sigma.(sync_{t\multimap t'} \circ \mathbf{K}[\![e]\!])\sigma((sync_t \circ \mathbf{K}[\![e']\!])\sigma) & \text{(ind. hyp.)} \\
= & \lambda\sigma.sync_{t'}(\mathbf{K}[\![e]\!]\sigma((desync_t \circ sync_t)(\mathbf{K}[\![e']\!]\sigma))) & \text{(def. } sync_{t\multimap t'}\text{)} \\
\sqsubseteq & \lambda\sigma.sync_{t'}(\mathbf{K}[\![e]\!]\sigma(\mathbf{K}[\![e']\!]\sigma)) & (desync_t \circ sync_t \sqsubseteq id) \\
= & sync_{t'} \circ \mathbf{K}[\![e\ e']\!] & \text{(def. } \mathbf{K}[\![e\ e']\!]\text{)}
\end{array}$$

- Case PAIR: one has

$$\begin{array}{lll}
& \mathbf{S}[\![\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2]\!] \circ sync_\Gamma & \\
= & (\mathbf{S}[\![\Gamma_1 \vdash e_1 : t_1]\!] \times \mathbf{S}[\![\Gamma_2 \vdash e_2 : t_2]\!]) \circ \mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!] \circ sync_\Gamma & \text{(def. } \mathbf{S}[\![\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2]\!]\text{)} \\
= & (\mathbf{S}[\![\Gamma_1 \vdash e_1 : t_1]\!] \times \mathbf{S}[\![\Gamma_2 \vdash e_2 : t_2]\!]) \circ \langle sync_{\Gamma_1}, sync_{\Gamma_2}\rangle & \text{(Property 33)} \\
= & \langle \mathbf{S}[\![\Gamma_1 \vdash e_1 : t_1]\!] \circ sync_{\Gamma_1}, \mathbf{S}[\![\Gamma_2 \vdash e_2 : t_2]\!] \circ sync_{\Gamma_2}\rangle & \\
\sqsubseteq & \langle sync_{t_1} \circ \mathbf{K}[\![e_1]\!], sync_{t_2} \circ \mathbf{K}[\![e_2]\!]\rangle & \text{(ind. hyp.)} \\
= & sync_{t_1\otimes t_2} \circ \langle \mathbf{K}[\![e_1]\!], \mathbf{K}[\![e_2]\!]\rangle & \text{(def. } sync_{t_1\otimes t_2}\text{)} \\
= & sync_{t_1\otimes t_2} \circ \mathbf{K}[\![(e_1, e_2)]\!] & \text{(def. } \mathbf{K}[\![(e_1, e_2)]\!]\text{)}
\end{array}$$

- Case LETPAIR: the proof is a straightforward if tedious variation on the case of LAMBDA and PAIR, combining Property 33 and Lemma 10 with the induction hypothesis. We leave it to the brave reader.

- Case FIX: one has

$$\begin{array}{lll}
& \mathbf{S}[\![\Gamma \vdash \mathtt{fix}\ e : t]\!] \circ sync_\Gamma & \\
= & \lambda\sigma.\mathbf{fix}\,((\mathbf{S}[\![\Gamma \vdash e : t \multimap t']\!](sync_\Gamma\sigma)) \circ \mathbf{S}[\![\vdash t' <:_1 t]\!]) & \text{(def. } \mathbf{S}[\![\Gamma \vdash \mathtt{fix}\ e : t]\!]\text{)} \\
\sqsubseteq & \lambda\sigma.\mathbf{fix}\,((sync_{t\multimap t'}(\mathbf{K}[\![e]\!]\sigma)) \circ \mathbf{S}[\![\vdash t' <:_1 t]\!]) & \text{(ind. hyp.)} \\
\sqsubseteq & \lambda\sigma.\mathbf{fix}\,(sync_{t'} \circ (\mathbf{K}[\![e]\!]\sigma) \circ desync_t \circ \mathbf{S}[\![\vdash t' <:_1 t]\!]) & \text{(def. } sync_{t\multimap t'}\text{)} \\
\sqsubseteq & \lambda\sigma.\mathbf{fix}\,(sync_{t'} \circ (\mathbf{K}[\![e]\!]\sigma) \circ desync_{t'}) & \text{(Property 34)} \\
= & \lambda\sigma.\mathbf{fix}\,(sync_{t'\multimap t'}(\mathbf{K}[\![e]\!]\sigma)) & \text{(def. } sync_{t'}\,t'\text{)} \\
\sqsubseteq & \lambda\sigma.\mathbf{fix}\,(sync_{t'\multimap t'}(\mathbf{K}[\![e]\!]\sigma)) & \text{(def. } sync_{t'}\,t'\text{)} \\
\sqsubseteq & \lambda\sigma.sync_t(\mathbf{fix}\,(\mathbf{K}[\![e]\!]\sigma)) & \text{(Property 37)} \\
= & sync_t \circ \mathbf{K}[\![\mathtt{fix}\ e]\!] & \text{(def. } \mathbf{K}[\![\mathtt{fix}\ e]\!]\text{)}
\end{array}$$

- Case CONST, OP, MERGE and WHEN: theses cases are immediate because their interpretation is of the form $\mathbf{S}[\![\Box \vdash e : t]\!] = sync_{\Box\vdash t}\mathbf{K}[\![e]\!]$ by definition.

- Case SUB: one has

$$\begin{array}{lll}
\mathbf{S}[\![\Gamma \vdash e : t']\!] \circ sync_\Gamma & = & \mathbf{S}[\![\vdash t <:_k t']\!] \circ \mathbf{S}[\![\Gamma \vdash e : t]\!] \circ sync_\Gamma \quad \text{(def. } \mathbf{S}[\![\Gamma \vdash e : t']\!]\text{)} \\
& \sqsubseteq & \mathbf{S}[\![\vdash t <:_k t']\!] \circ sync_t \circ \mathbf{K}[\![e]\!] \quad \text{(ind. hyp.)} \\
& \sqsubseteq & sync_{t'} \circ \mathbf{K}[\![e]\!] \quad \text{(Property 34)}
\end{array}$$

- Case RESCALE: one has

$$\begin{array}{rcll} \mathbf{S}[\![\Gamma \vdash e : t]\!] \circ sync_\Gamma & = & \mathbf{S}[\![\vdash t \uparrow_{ct} t']\!] \circ \mathbf{S}[\![\Gamma' \vdash e : t']\!] \circ \mathbf{S}[\![\vdash \Gamma \downarrow_{ct} \Gamma']\!] \circ sync_\Gamma & \text{(def. } \mathbf{S}[\![\Gamma \vdash e : t']\!]) \\ & \sqsubseteq & \mathbf{S}[\![\vdash t \uparrow_{ct} t']\!] \circ \mathbf{S}[\![\Gamma' \vdash e : t']\!] \circ sync_{\Gamma'} & \text{(Property 35)} \\ & \sqsubseteq & \mathbf{S}[\![\vdash t \uparrow_{ct} t']\!] \circ sync_{t'} \circ \mathbf{K}[\![e]\!] & \text{(ind. hyp.)} \\ & \sqsubseteq & sync_t \circ \mathbf{K}[\![e]\!] & \text{(Property 35)} \end{array}$$

  which concludes the proof. □

We conclude this section with some remarkable high-level corollaries of the Lax Coherence and Totality lemmas.

Theorem 6 (Synchronization)**.** *For any closed expression e of stream type, one has*

$$\mathbf{S}[\![\square \vdash e : dt :: ct]\!]\bot \quad = \quad sync_{dt::ct}(\mathbf{K}[\![e]\!]\bot)$$

Proof. From Lemma 11 we know that the left-hand side is smaller than the right-hand one, and from Theorem 4 that it is total. Since in the domain $\mathbf{S}[\![dt :: ct]\!]$ total elements are maximal, the left-hand side is maximal and hence equal to the right-hand side. □

Theorem 7 (Strong Causality)**.** *Take a closed expression e such that there exists a clock type ct with* $rate(\mathbf{S}[\![ct]\!]) > 0$ *and a derivation of* $\square \vdash e : dt :: ct$*. Then, stream* $(\mathbf{K}[\![e]\!]\bot)$ *is total.*

Proof. Let us write k for $\mathbf{K}[\![e]\!]\bot$ and s for $\mathbf{S}[\![\square \vdash e : dt :: ct]\!]\bot$. From Theorem 6 and the fact that ct has a strictly positive rate, we have $k = desync_{dt::ct} s$ and thus

$$\begin{array}{rcl} stream\ k & = & stream\ (desync_{dt::ct} s) \\ & = & (stream \circ unstream \circ unpack)\, s \\ & = & unpack\ s \end{array}$$

which is total since s is and $rate(\mathbf{S}[\![ct]\!]) > 0$. □

3.5 Discussion

We finish the chapter with a discussion of alternative presentations of the semantics, and a comparison with existing synchronous languages.

3.5.1 Improving the Typed Semantics

Domain theory provides a fixpoint operator for any function of a given domain into itself and thus was a natural way to define the semantics of untyped programs, where we need to make sense of arbitrary recursive definitions. In contrast, using domain theory for the typed semantics is debatable. On the one hand, it makes it easy to state results such as Lemma 11 since the untyped and typed theorems refer to objects living inside the same category. On the other hand, typed programs are better behaved than untyped one, yet the typed semantics

does not reflect this fact natively. For instance, we have had to prove explicitly that well-typed programs correspond to total elements of a domain (Lemma 8). We would like this fact to be true *by construction* in a better semantic setting.

An alternative would be to have a typed semantics where partial streams do not exist. It should be possible to provide such a semantics in complete ultrametric spaces (see, for example, Krishnaswami and Benton [2011]), or in the topos of trees $\hat{\omega}$, using recent work of Birkedal et al. [2012]. All streams would be infinite/total by definition and recursive definitions would be explained using Banach's fixed point theorem or the existence of fixpoints in $\hat{\omega}$ rather than Kleene's fixpoint theorem.

3.5.2 Causality

The synchronous setting involves two distinct time scales. The first one is the ambient global time scale, which is the successive time steps between which buffering is needed. Then there is the less apparent but nevertheless important *internal* time scale, in which the actual computations occur, during a time step.

In traditional synchronous programming languages, each time scale is controlled by a separate static analysis. The global time scale is governed by clock types, which check the consistency of buffering between time steps. The internal time scale is governed by the causality analysis, which checks that the computations performed during a time step terminate, and thus that time actually passes. We argue that this separation, while simple to implement, actually limits our understanding and hampers the design of better languages and compilers.

Consider the Lucy-n program given in Figure 3.28 (a). It defines two recursive binary stream variables `x` and `y`. The first variable denotes the periodic stream $(0\ 1)^\omega$ while the second one denotes its negation $(1\ 0)^\omega$. What matters here is that the computation of the elements of odd rank of x depends on the elements of odd rank of y, but that this dependence is reversed for elements of even rank. The `buffer` operator marks an explicit application of the SUB rule.

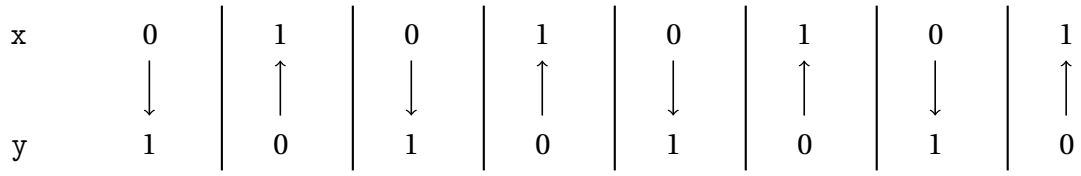
The pair $(\mathtt{x},\mathtt{y})$ admit several valid clock types in Lucy-n. A first possibility is $(1)\otimes(1)$, which corresponds to the chronogram given in Figure 3.28 (b). Vertical bars correspond to separate time steps and arrows to data dependencies. The chronogram shows that dependencies during a time step alternate: at even time steps `x` depends on `y` and conversely at odd time steps. Following the tradition set by Lucid Synchrone, Lucy-n uses a causality analysis based on the absence of cycles in the *syntactic* instantaneous dependence graph, which does not take clocks into account. Thus, here the graph is cyclic, and this program is rejected. Another valid type is $(1\ 0\ 0\ 1)\otimes(0\ 1\ 1\ 0)$, which corresponds to the chronogram given in Figure 3.28 (c). In contrast with the first one, it is accepted by the causality analysis of Lucy-n since dependencies alternate but always cross a time step, making the syntactic instantaneous dependence graph acyclic. Nevertheless, this type has a lower throughput than $(1)\otimes(1)$.

This example readily translates to μAS. It takes the form of a function *twohalvesbody*. Its fixpoint corresponds to the output of *twohalves*. The corresponding code is given in Figure 3.28 (e). While the type

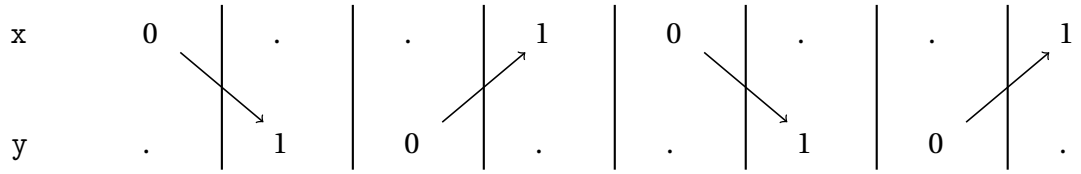
$$(1)\otimes(1)\multimap(1)\otimes(1)$$

```
let node twohalves () = (x, y) where
  rec x = merge (10) false (not buffer (y when (01)))
  and y = merge (10) (not buffer (x when (10))) false
```

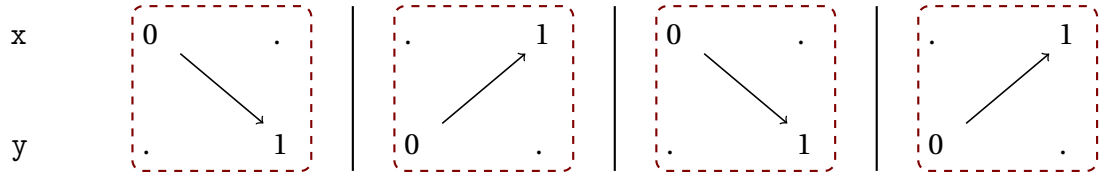
(a) - Example code



(b) - Chronogram and dependencies for $\mathtt{x} :: (1)$ and $\mathtt{y} :: (1)$



(c) - Chronogram and dependencies for $\mathtt{x} :: (1\ 0\ 0\ 1)$ and $\mathtt{y} :: (0\ 1\ 1\ 0)$



(d) - Chronogram and dependencies for $\mathtt{x} :: (1)$ and $\mathtt{y} :: (1)$ rescaled by $(2)^{\omega}$

$$
\begin{aligned}
\textit{twohalvesbody} &= \mathtt{fun\ z.let\ (x,y) = z\ in\ (merge\ (1\ 0)\ 0\ (not\ (when\ (0\ 1)\ y)},\\
&\qquad \mathtt{merge\ (1\ 0)\ (not\ (when\ (1\ 0)\ x)\ 0))}\\
\textit{twohalves} &= \mathtt{fix}\ \textit{twohalvesbody}
\end{aligned}
$$

(e) - Translation to μAS

Figure 3.28: Lucy-n program - clock-dependent causality

is valid for *twohalvesbody*, it makes *twohalves* ill-typed because the adaptability constraint of rule FIX cannot be satisfied. As for Lucy-n, assigning the type

$$0(1\,0\,0\,1)\otimes(0\,0\,1\,1)\multimap(1\,0\,0\,1)\otimes(0\,1\,1\,0)$$

to *twohalvesbody* makes it possible to accept *twohalves* with the type $(1\,0\,0\,1)\otimes(0\,1\,1\,0)$ since the 1-adaptability relation

$$\vdash(1\,0\,0\,1)\otimes(0\,1\,1\,0)<:_1 0(1\,0\,0\,1)\otimes(0\,0\,1\,1)$$

then holds. As explained before, this type has sub-optimal throughput; yet, in μAS, we can add a local time scale driven by the clock type (2) to recover the first type. This corresponds to the chronogram of Figure 3.28 (d). Dashed boxes show how local time steps are now aggregated to give the same observable behavior as in Figure 3.28 (b). We will see in the next chapter that such a program does not necessitate any particular treatment during code generation.

This example shows that in Lucy-n clock types influence the causality analysis, which is problematic when the two analyses are completely separate. Here the type inference engine makes a choice that leads to rejection later in the compilation process. To accept the first type requires enriching both the clock type inference engine and causality analysis of Lucy-n, as well as intra-step scheduling and code generation. In contrast, using local time scales and integer clocks one may, at least in theory, avoid complex intra-step causality conditions and rely on clock typing instead.

3.6 Bibliographic notes

We now briefly discuss related work and models that were not touched upon in Section 2.11.

n-Synchrony The language presented in this chapter is an extension of Lucy-n, proposed by Mandel, Plateau, and Pouzet [2010], which takes place in the n-synchronous framework of Cohen et al. [2006].[2] The most complete description of Lucy-n to date can be found in the PhD dissertation of Plateau [2010]. Up to now this line of work has focused on solving difficult type inference questions and comparatively little attention has been paid to the semantics and properties of the type system. This thesis intends to remedy this gap. In particular, to our knowledge this is the first time a result relating the typed and untyped semantics for an n-synchronous language has been stated or proven.

Another salient feature of the language is the integration of several static analyses that have traditionally been disjoint in synchronous functional languages: clock typing, initialization analysis and causality analysis. We will discuss this point in detail in Chapter 6; let us briefly mention that we consider the integration of causality analysis and clocking is feasible only because local time scales make the distinction between global time steps and the unfolding of dependencies inside a time step.

[2] Strictly speaking, the language is not a superset of Lucy-n since clock polymorphism and node reuse are missing. These points will be discussed in Chapter 5.

Local time scales In the binary case, a restricted form of local time scales is already present in other synchronous functional languages. Local time scales driven by binary clocks correspond to wrapping a block of code inside a conditional statement. Such a construction has been used as an optimization during the compilation process, as discussed in the PhD thesis of Raymond [1991] and, later, in the paper of Biernacki et al. [2008]. They can also be seen as *activation conditions* [Halbwachs, 2005] checked by the clock type system. The *multiclock* extension of Esterel v7 [Esterel Technologies, 2005, Chapters 13 and 14] offers constructions roughly similar to activation conditions, but integrated at the module level.

It is, to our knowledge, the first time a general local time scale construction has been proposed for a functional synchronous language. There is, however, an existing concept which is very close and in fact partly inspired our work: the notion of *reactive domain* proposed by Pasteur [Pasteur, 2013; Mandel et al., 2010] in the setting of ReactiveML.

ReactiveML [Mandel and Pouzet, 2005] is a general-purpose functional language which offers primitives originating from the synchronous model. It is more expressive than languages à la Lustre, at the expense of relying on unbounded data-structures and dynamic memory management. Reactive domains provided the inspiration for our local time scales: they fuse several reactions of a block of code into one macro-reaction, making this macro-reaction appear atomic to the outside world. Pasteur also introduces an abstract notion of *clock*, quite different from ours: in his setting, a clock is an abstract name identifying a domain. Such clocks serve dual roles: first, the programmer explicitly manipulates them to control the speed at which a domain executes relatively to the external world; second, they provide the basis for a custom clock type system preventing domain-local signals to escape their domain.

The differences between local time scales and reactive domains reflect the philosophical differences between functional synchronous languages such as Lucid Synchrone or Lucy-n on the one hand and ReactiveML on the other. Functional synchronous languages offer strong guarantees, including static memory bounds; local time scales preserve this fact thanks to the static information contained in clock types. The price to pay is the restricted expressiveness of the language; for instance, none of these are Turing-complete. ReactiveML in general and reactive domains in particular do not provide such guarantees, but are much less restrictive; for instance, the amount of internal steps a domain performs is not fixed outright but depends on a dynamic condition which is determined on the fly.

Semantics of programming languages Our approach to the semantics of the language can be traced back to several sources. We used elementary domain theory as explained in Chapter 2. The other main ingredient is the definition of the typed semantics as a dependently typed map from typing derivations to some semantic setting. This idea comes from several sources, including the already mentioned paper of Reynolds [2000] on subtyping, or the older realizability literature; let us also mention more practically-minded work on the representation of typed terms in dependent type theory [Benton et al., 2012].

The total and partial convergence predicates we used to prove the absence of deadlocks are example of a *unary logical relation*, also called a *realizability predicate*. Logical relations and realizability are classic tools in the metatheoretical study of programming language semantics

and computational logic [Van Oosten, 2008; Girard, 1987]. In the unary case, a realizability predicate defines the set of *realizers* for a type, which are in a sense *good* inhabitants of the type, for whatever notion of goodness one is interested in—in our case, realizers are *total* elements of the domain $\mathbf{S}[\![t]\!]$. One says that a program *realizes* a type t if it is a realizer of t. Then, one typically proves a so-called *adequacy lemma* stating that *well-typed programs realize their types*. Lemma 8 is the adequacy lemma for our logical relation.

Chapter 4

Compilation

In this chapter, we explain how the typed programs of Chapter 3 can be implemented as digital synchronous circuits. This takes the form of a new interpretation of typing derivations. An important difference between this interpretation and the synchronous semantics is the finite nature of the underlying objects: programs are no longer explained in terms of mathematical functions acting upon infinite streams of data, but as finite state machines processing finite chunks of input and outputs in an incremental fashion. Therefore, this interpretation can be seen as a compilation scheme.

The type system provides two main ingredients in order to reduce finite-state description. First, clock types makes it possible to statically bound the amount of data processed in each reaction of a machine in a way that still guarantees that the global behavior of the program is preserved. Second, linear higher-order functions can be implemented as first-order ones in a modular way by reducing them to additional inputs and outputs. This latter part relies crucially on the careful handling of scoping and linearity present in the type system.

We propose a first-order language to describe finite state machines. The language is more structured and syntactic than the graph-based formalism of automata theory. This makes it a convenient target for the compilation process. It is equipped with a simple type system for ensuring that machines have finite state. The translation is thus structured as a type-preserving interpretation.

The rest of this chapter is organized as follows. Section 4.1 presents an extended overview of the translation. Section 4.2 defines and explains the target language for describing state machines, including its type system, operational semantics, and graphical representation. We also prove useful properties, such as a subject reduction result, and show how well-typed machines describe finite-state transducers. Section 4.3 builds a higher-order layer on top of the first-order language. This layer is used as the target of the type-preserving translation, which is described in Section 4.4. Each typing judgment gives rise to a state machine for use in the compilation of programs. In particular, linear aspects of the type system are translated to machines handling data movement. We prove its correctness using a logical relation between machines and the synchronous semantics. Finally, Section 4.5 discusses the implementation of the machine language to actual digital circuits, including practical details about hardware description languages.

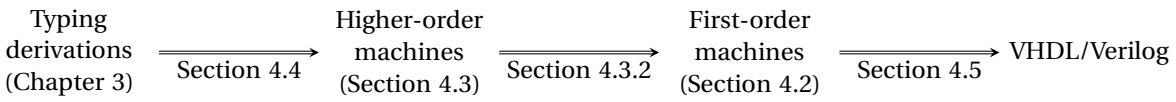


Figure 4.1: Compilation - overview

4.1 Overview

The typing derivations of Chapter 3 are nothing more than terms belonging to a linear λ-calculus enriched with special constants for stream processing. These special constants include operators directly present in the source language—that is, stream sampling, merging, pointwise operators, and ultimately periodic words—but also the operations needed by the adaptability, gathering, and scattering judgments. In this chapter, we traduce this linear λ-calculus, including its constants, to finite state digital circuits.

Figure 4.1 describes the flow of the translation from typing derivation to hardware description languages. However, this chapter is not organized in this top-down order but rather proceed from the bottom up. Let us describe each intermediate step.

1. First, we define a combinator language for describing first-order state machines. The combinators include sequential and parallel composition, as well as feedback loops. Section 4.2 details the following points.

   - The semantics of the language is operational. It is expressed by the predicate

     $$m/x \to m'/y$$

     which states that the machine m evolves into m' while consuming input x and producing output y. This semantics is non-deterministic, as there may be several (m', y) pairs for a given (m, x) pair.
   - We equip the language of machines with a simple type system. We show that well-typed machines have finite state, and thus can be implemented as finite state transducers.
   - The language of machines includes the *replication* combinator $\texttt{mrepl}_n(m)$. One reaction of this machine performs at most n reactions of the wrapped machine m; the precise amount of reactions performed is specified by an additional input.
   - Nearly all constants present in typing derivations have matching primitive combinators in the language of machines. For those which do not, we build "macro" machines by combining several machines of the base language.

2. In Section 4.3, we build a language of linear higher-order machines on top of the base one described in the previous step. The goal is to obtain new machine and combinators implementing currying, evaluation. This is done by reducing inputs of higher types

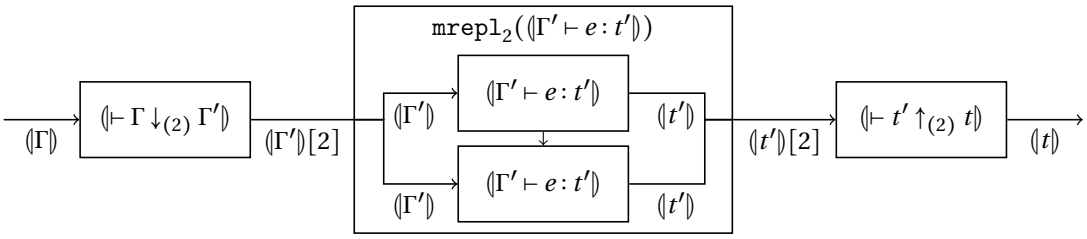


Figure 4.2: Compilation - informal compilation scheme for a local time scale driven by (2)

to additional, first-order inputs and outputs, as alluded to in Chapter 3. We lift all the constants from the language of first-order machines to this higher-order setting, as well as the replication combinator.

3. In Section 4.4 we finally interpret the typing derivations of Chapter 3 into the higher-order machine language defined previously. The compiled machine $\llparenthesis\Gamma \vdash e : t\rrparenthesis$ is defined by induction on a derivation of $\Gamma \vdash e : t$. The compilation is mostly transparent: rule APP in typing derivations is interpreted as the application in linear higher-order machines, pairing by pairing, and so on. The only interesting case is the one of local time scales.

   According to rule RESCALE, a local time scale is made of four parts: the driving clock type ct, a scattering judgment $\Gamma \vdash ct : \Gamma'$, the code running inside the time scale $\Gamma' \vdash e : t'$, and the gathering judgment $\vdash t' \uparrow_{ct} t$. Imagine the case of a time scale driven by (2). Then, the compilation of $\Gamma \vdash e : t$ proceeds as in the following informal and slightly simplified explanation.

   a) The inputs of e present in Γ are scattered by (2). Concretely, this is done by the scattering machine $\llparenthesis\vdash \Gamma \downarrow_{(2)} \Gamma'\rrparenthesis$, whose inputs are in $\llparenthesis\Gamma\rrparenthesis$ and whose outputs are in $\llparenthesis\Gamma'\rrparenthesis[2]$. Informally, the notation $\llparenthesis\Gamma'\rrparenthesis[2]$ describes lists of size *at most* two of values in $\llparenthesis\Gamma'\rrparenthesis$.

   b) The machine $\llparenthesis\Gamma'\rrparenthesis e t'$ has inputs in $\llparenthesis\Gamma'\rrparenthesis$ and outputs in $\llparenthesis t\rrparenthesis$. By replicating it twice, one obtains a machine whose input is in $\llparenthesis\Gamma'\rrparenthesis[2]$ and output is in $\llparenthesis t'\rrparenthesis 2$. This machine is compatible with the scattering one obtained in step a).

   c) Finally, the outputs obtained after step b) are gathered by (2). This is done by the gathering machine $\llparenthesis\vdash t' \uparrow_{(2)} t\rrparenthesis$, which takes an input of type $\llparenthesis t'\rrparenthesis[2]$ and producess a final output in $\llparenthesis t\rrparenthesis$.

   This code generation scheme is represented in a slightly stylized manner in Figure 4.2. The machine $\llparenthesis\Gamma \vdash e : t\rrparenthesis$ inside the local time scale is replicated twice. We have added the thin vertical arrow between the copies to highlight the fact that they are ordered: the state of the first copy is passed to the second one during a global reaction.

4. In Section 4.5 we explain informally how first-order machines can be implemented in hardware description languages such as VHDL or Verilog. This explanation is informal

v	::=	`()`	Unit value
	|	$n \in \mathbb{N}$	Natural number
	|	(v, v)	Pair
	|	v^*	List
mt	::=	`unit`	Unit type
	|	$\{n\}$	Natural number bounded by n
	|	$mt \times mt$	Product type
	|	$mt[n]$	List type
mtm	::=	$mt \multimap mt$	

Figure 4.3: Machine language - syntax of values and types

as we did not want to give a formal semantics for such languages. Finally, by composing this implementation with all the previous steps, one may translate typing derivations to finite-state digital synchronous circuits.

4.2 A Machine Construction Kit

This section presents a language dedicated to the description of state machines. It is a language of name-free, first-order combinators endowed with an operational semantics explaining how a machine, given input data, transitions from its current state to the next, producing output data along the way. It is equipped with a simple type system that helps structure the translation from the source language, and ensures that our machines are finite state. We first explain the syntax, type system and operational semantics of the language in one go. Then, we show that the type system captures the finiteness of machines. Finally, we define some composite "macro-machines" which will prove convenient later on.

4.2.1 Syntax, Typing, and Reactions

Figure 4.3 gives the syntax of types for the target language. It comprises four syntactic categories: values v, value types mt, and machine types mtm.

Values and their types A value v is a finite piece of data computed by a machine during a reaction. It can be either a simple scalar value, such as a natural n, boolean b or unit value `()`, or a compound value. A compound value is either a pair of values (v_1, v_2), or a list of values v_l. We refer to the i-th element of v_l, if it exists, as $v_l[i]$, write $x; xl$ for list cons, and write $[x_1, \ldots, x_n]$ for list literals.

A value type mt describes which values are allowed as inputs or outputs of a machine during a reaction. Unit and product types are as expected. A *sized* type $mt[n]$ describes lists of values of type mt; the natural number n is the maximum size of the list. Finally, the value type dt denotes a scalar with the same grammar as in Chapter 3.

$$\boxed{\vdash v : mt} \text{ and } \boxed{\vdash^i v : mt}$$

$$\frac{}{\vdash * : \mathtt{unit}} \qquad \frac{n \leq m}{\vdash n : \{m\}} \qquad \frac{\vdash v_1 : mt_1 \qquad \vdash v_2 : mt_2}{\vdash (v_1, v_2) : mt_1 \times mt_2} \qquad \frac{\vdash^n xl : mt}{\vdash xl : mt[n]}$$

$$\frac{}{\vdash^n \epsilon : mt} \qquad \frac{\vdash x : mt \qquad \vdash^n xl : mt}{\vdash^{1+n} x.xl : mt}$$

Figure 4.4: Machines - value typing judgment

Value typing The value typing judgment is given in Figure 4.4. The judgment $\vdash v : mt$ expresses that the value v has type mt. Its rules are given in Figure 4.4. It relies on the mutually-defined auxiliary judgment $\vdash^n xl : mt$ expressing that the list xl is of size n and that each of its element is of type mt; its rules are defined in the lower row of Figure 4.4. The first three rules describe purely scalar values. Products are once again typed componentwise, at the same index. A value vl of type $mt[n]$ is a list whose size is *at most* n and whose elements inhabit mt, as expressed by judgment $\vdash^n vl : mt$.

Machine types While value types describe data, machine types mtm describe the interface of a machine $mt_1 \multimap mt_2$ with inputs in mt_1 and outputs in mt_2. The grammar of machine types is not recursive since the language is first-order. Given a machine type $mtm = mt_1 \multimap mt_2$, we write mtm^- for its input type mt_1, and mtm^+ for its output type mt_2. We also write $(mt_1 \multimap mt_2)[n]$ for the machine type $mt_1[n] \multimap mt_2[n]$.

Machines We now turn to the description of machines m, which describe possibly stateful circuits with inputs and outputs. Their grammar is given in Figure 4.5. Each machine comes together with a typing rule and a description of its operational semantics.

- The typing judgment expresses that a machine m has (machine) type mtm, which is written $\vdash m : mtm$. Since machines do not contain variables, the judgment involves no context handling. Typing rules are given in Figure 4.6

- The operational semantics is built on a *reaction* judgment expressing that a machine m reacts to some input value x by producing an output value y and becoming a new machine m', which is written $m/x \to m'/y$. In this case we also say that m *evolves* into m'. If m holds explicit type annotations, so will m'; annotations are preserved by reactions. The reaction judgment will thus frequently involve the judgments governing the evolution of types described in the previous paragraphs. Reaction rules are given in Figure 4.7.

m	$::=$	$\mathtt{mid}_{mt}$	Identity
	$\mid$	$\mathtt{mswap}_{mtm}$	Exchange
	$\mid$	$\mathtt{mdup}_{mt}$	Duplication
	$\mid$	$\mathtt{mforg}_{mt}$	Erasure (*forget*)
	$\mid$	$\mathtt{mneutr}_{mt}$	Neutralize (*absorb*)
	$\mid$	$\mathtt{mneutrinv}_{mt}$	Inverted neutralize (*generate*)
	$\mid$	$\mathtt{mconst}_{v}$	Constant
	$\mid$	$\mathtt{mmux}_{mt}$	Multiplexer
	$\mid$	$\mathtt{mpw}_{p}(n)$	Ultimately periodic integer word
	$\mid$	$\mathtt{msum}^{n}_{n}$	Summation
	$\mid$	$\mathtt{mbuff}^{mt}_{n,n,n}(v)$	Bounded buffer
	$\mid$	$\mathtt{mgath}^{mt}_{n,n}$	Scattering
	$\mid$	$\mathtt{mscatt}^{mt}_{n,n}$	Gathering
	$\mid$	$\mathtt{mstutt}^{n}_{mt}$	Stuttering
	$\mid$	$\mathtt{munwrap}_{mt}$	Unwrapping
	$\mid$	$\mathtt{mzip}^{n}_{mt,mt}$	Zipping
	$\mid$	$\mathtt{munzip}^{n}_{mt,mt}$	Unzipping
	$\mid$	$\mathtt{mncoer}_{n,n}$	Bounded integer coercion
	$\mid$	$\mathtt{mlcoer}^{mt}_{n,n}$	List size coercion
	$\mid$	$m \bullet m$	Horizontal composition
	$\mid$	$m \Vert m$	Vertical composition
	$\mid$	$\mathtt{mfb}^{mt}_{mt,mt}(m)$	Feedback loop
	$\mid$	$\mathtt{mrepl}_{n}(m)$	Replication

Figure 4.5: Machine language - syntax of machines

Since the language is relatively simple, we describe simultaneously machines, their typing rules, and their operational semantics.

Machines fall into two broad groups: they are either atomic machines implementing fixed functions, or machine combinators that build complex machines from simpler ones. Let us describe each group in turn.

1. The first group of atomic machines provide low-level facilities for data movement. They implement operations that would be implicit in a language with names, such as weakening and contraction. They are:

   - the identity machine $\boxed{\mathtt{mid}_{mt}}$, which transmit its input as is;

$$\textsc{MID}\quad \frac{}{\vdash \mathtt{mid}_{mt} : mt \multimap mt} \qquad \frac{}{\mathtt{mid}_{mt}/x \to \mathtt{mid}_{mt}/x}$$

$$\boxed{\vdash m : mtm}$$

$$\textsc{MId}\ \frac{}{\vdash \mathtt{mid}_{mt} : mt \multimap mt} \qquad \textsc{MSwap}\ \frac{}{\vdash \mathtt{mswap}_{mtm} : mtm^{+} \times mtm^{-} \multimap mtm^{-} \times mtm^{+}} \qquad \textsc{MDup}\ \frac{}{\vdash \mathtt{mdup}_{mt} : mt \multimap mt \times mt}$$

$$\textsc{MForget}\ \frac{}{\vdash \mathtt{mforg}_{mt} : mt \multimap \mathtt{unit}} \qquad \textsc{MNeutralize}\ \frac{}{\vdash \mathtt{mneutr}_{mt} : \mathtt{unit} \times mt \multimap mt} \qquad \textsc{MNeutralizeInv}\ \frac{}{\vdash \mathtt{mneutrinv}_{mt} : mt \multimap \mathtt{unit} \times mt}$$

$$\textsc{MConst}\ \frac{\vdash v : mt}{\vdash \mathtt{mconst}_{v} : \mathtt{unit} \multimap mt} \qquad \textsc{MMux}\ \frac{}{\vdash \mathtt{mmux}_{mt} : \{1\} \times mt \times mt \multimap mt} \qquad \textsc{MPword}\ \frac{\vdash n : \{|u|+|v|-1\}}{\vdash \mathtt{mpw}_{u(v)}(n) : \mathtt{unit} \multimap \{\lceil u(v) \rceil\}}$$

$$\textsc{MSum}\ \frac{}{\vdash \mathtt{msum}^{n_2}_{n_1} : \{n_1\}[n_2] \multimap \{n_1 * n_2\}} \qquad \textsc{MBuff}\ \frac{\vdash c : mt[n_3]}{\vdash \mathtt{mbuff}^{mt}_{n_1,n_2,n_3}(c) : \{n_2\} \times mt[n_1] \multimap mt[n_2]}$$

$$\textsc{MGather}\ \frac{}{\vdash \mathtt{mgath}^{mt}_{n_1,n_2} : mt[n_1][n_2] \multimap mt[n_1 * n_2]} \qquad \textsc{MScatter}\ \frac{}{\vdash \mathtt{mscatt}^{mt}_{n_1,n_2} : \{n_1\}[n_2] \times mt[n_1 * n_2] \multimap mt[n_1][n_2]}$$

$$\textsc{MStutter}\ \frac{}{\vdash \mathtt{mstutt}^{n}_{mt} : \{n\} \times mt \multimap mt[n]} \qquad \textsc{MUnwrap}\ \frac{}{\vdash \mathtt{munwrap}_{mt} : mt[1] \multimap mt}$$

$$\textsc{MZip}\ \frac{}{\vdash \mathtt{mzip}^{n}_{mt_1,mt_2} : mt_1[n] \times mt_2[n] \multimap (mt_1 \times mt_2)[n]}$$

$$\textsc{MUnzip}\ \frac{}{\vdash \mathtt{munzip}^{n}_{mt_1,mt_2} : (mt_1 \times mt_2)[n] \multimap mt_1[n] \times mt_2[n]} \qquad \textsc{MNCoer}\ \frac{}{\vdash \mathtt{mncoer}_{n_1,n_2} : \{n_1\} \multimap \{n_2\}}$$

$$\textsc{MLCoer}\ \frac{}{\vdash \mathtt{mlcoer}^{mt}_{n_1,n_2} : mt[n_1] \multimap mt[n_2]} \qquad \textsc{MComp}\ \frac{\vdash m_1 : mt_2 \multimap mt_3 \qquad \vdash m_2 : mt_1 \multimap mt_2}{\vdash m_1 \bullet m_2 : mt_1 \multimap mt_3}$$

$$\textsc{MPar}\ \frac{\vdash m_1 : mt_1 \multimap mt_2 \qquad \vdash m_2 : mt_3 \multimap mt_4}{\vdash m_1 \parallel m_2 : mt_1 \times mt_3 \multimap mt_2 \times mt_4} \qquad \textsc{MFeedback}\ \frac{\vdash m : mt_1 \times mt_3 \multimap mt_2 \times mt_3}{\vdash \mathtt{mfb}^{mt_3}_{mt_1,mt_2}(m) : mt_1 \multimap mt_2}$$

$$\textsc{MRepl}\ \frac{\vdash m : mt_1 \multimap mt_2}{\vdash \mathtt{mrepl}_{n}(m) : \{n\} \times mt_1[n] \multimap mt_2[n]}$$

Figure 4.6: Machines - main typing judgment

$$\boxed{m/x \to m'/y} \text{ and } \boxed{m/xl \to_n m'/yl}$$

$$\frac{}{\mathtt{mid}_{mt}/x \to \mathtt{mid}_{mt}/x} \qquad \frac{}{\mathtt{mswap}_{mtm}/(x,y) \to \mathtt{mswap}_{mtm}/(y,x)} \qquad \frac{}{\mathtt{mdup}_{mt}/x \to \mathtt{mdup}_{mt}/(x,x)}$$

$$\frac{}{\mathtt{mforg}_{mt}/x \to \mathtt{mforg}_{mt}/()} \qquad \frac{}{\mathtt{mneutr}_{mt}/((),x) \to \mathtt{mneutr}_{mt}/x}$$

$$\frac{}{\mathtt{mneutrinv}_{mt}/x \to \mathtt{mneutrinv}_{mt}/((),x)} \qquad \frac{}{\mathtt{mconst}_{s}/() \to \mathtt{mconst}_{s}/s}$$

$$\frac{}{\mathtt{mmux}_{mt}/(1,x,y) \to \mathtt{mmux}_{mt}/x} \qquad \frac{}{\mathtt{mmux}_{mt}/(0,x,y) \to \mathtt{mmux}_{mt}/y}$$

$$\frac{i < |u|}{\mathtt{mpw}_{u(v)}(i)/() \to \mathtt{mpw}_{u(v)}(i+1)/|u|[i]} \qquad \frac{|u| \le i < |u|+|v|-1}{\mathtt{mpw}_{u(v)}(i)/() \to \mathtt{mpw}_{u(v)}(i+1)/v[i-|u|]}$$

$$\frac{}{\mathtt{mpw}_{u(v)}(|u|+|v|-1)/() \to \mathtt{mpw}_{u(v)}(|u|)/v[|v|-1]} \qquad \frac{}{\mathtt{msum}_{n_1}^{n_2}/xl \to \mathtt{msum}_{n_1}^{n_2}/\sum xl}$$

$$\frac{c.xl = yl.c' \qquad |yl| = a \qquad |c'| \le n_3}{\mathtt{mbuff}_{n_1,n_2,n_3}^{mt}(c)/(a,xl) \to \mathtt{mbuff}_{n_1,n_2,n_3}^{mt}(c')/yl} \qquad \frac{gather(xl,yl)}{\mathtt{mgath}_{n_1,n_2}^{mt}/xs \to \mathtt{mgath}_{n_1,n_2}^{mt}/ys}$$

$$\frac{scatter(xl,yl,zl)}{\mathtt{mscatt}_{n_1,n_2}^{mt}/(xl,yl) \to \mathtt{mscatt}_{n_1,n_2}^{mt}/zl} \qquad \frac{}{\mathtt{mstutt}_{mt}^{n}/(a,x) \to \mathtt{mstutt}_{mt}^{n}/x^a}$$

$$\frac{}{\mathtt{munwrap}_{mt}/[x] \to \mathtt{munwrap}_{mt}/x} \qquad \frac{zip(xl,yl,zl)}{\mathtt{mzip}_{mt_1,mt_2}^{n}/(xl,yl) \to \mathtt{mzip}_{mt_1,mt_2}^{n}/zl}$$

$$\frac{zip(yl,zl,xl)}{\mathtt{munzip}_{mt_1,mt_2}^{n}/xl \to \mathtt{munzip}_{mt_1,mt_2}^{n}/(yl,zl)} \qquad \frac{a \le n_2}{\mathtt{mncoer}_{n_1,n_2}/a \to \mathtt{mncoer}_{n_1,n_2}/a}$$

$$\frac{xl = yl.zl \qquad |yl| = min(|xl|,n_2)}{\mathtt{mlcoer}_{n_1,n_2}^{mt}/xl \to \mathtt{mlcoer}_{n_1,n_2}^{mt}/yl} \qquad \frac{m_1/x \to m_1'/y \qquad m_2/y \to m_2'/z}{m_2 \bullet m_1/x \to m_2' \bullet m_1'/z}$$

$$\frac{m_1/x_1 \to m_1'/y_1 \qquad m_2/x_2 \to m_2'/y_2}{m_1 \shortparallel m_2/(x_1,x_2) \to m_1' \shortparallel m_2'/(y_1,y_2)} \qquad \frac{\vdash z : mt_3 \qquad m/(x,z) \to m'/(y,z)}{\mathtt{mfb}_{mt_1,mt_2}^{mt_3}(m)/x \to \mathtt{mfb}_{mt_1,mt_2}^{mt_3}(m')/y}$$

$$\frac{m/xl \to_a m/yl}{\mathtt{mrepl}_n(m)/(a,xl) \to \mathtt{mrepl}_n(m)/yl} \qquad \frac{}{m/\epsilon \to_0 m/\epsilon} \qquad \frac{m/x \to m/y \qquad m/xl \to_n m/yl}{m/x.xl \to_{1+n} m/y.yl}$$

Figure 4.7: Machines - reaction judgment

- the exchange machine $\boxed{\texttt{mswap}_{mtm}}$, which exchanges its inputs;

  MSWAP
  $$\frac{}{\vdash \texttt{mswap}_{mtm} : mtm^+ \times mtm^- \multimap mtm^- \times mtm^+} \qquad \frac{}{\texttt{mswap}_{mtm}/(x,y) \to \texttt{mswap}_{mtm}/(y,x)}$$

- the duplication machine $\boxed{\texttt{mdup}_{mt}}$, which duplicates its input;

  MDUP
  $$\frac{}{\vdash \texttt{mdup}_{mt} : mt \multimap mt \times mt} \qquad \frac{}{\texttt{mdup}_{mt}/x \to \texttt{mdup}_{mt}/(x,x)}$$

- the erasure machine $\boxed{\texttt{mforg}_{mt}}$, which reads its input but outputs nothing;

  MFORGET
  $$\frac{}{\vdash \texttt{mforg}_{mt} : mt \multimap \texttt{unit}} \qquad \frac{}{\texttt{mforg}_{mt}/x \to \texttt{mforg}_{mt}/()}$$

- the neutralization machine $\boxed{\texttt{mneutr}_{mt}}$, which forgets its useless left input;

  MNEUTRALIZE
  $$\frac{}{\vdash \texttt{mneutr}_{mt} : \texttt{unit} \times mt \multimap mt} \qquad \frac{}{\texttt{mneutr}_{mt}/((),x) \to \texttt{mneutr}_{mt}/x}$$

- and the generation machine $\boxed{\texttt{mneutrinv}_{mt}}$, which generates a useless left output.

  MNEUTRALIZEINV
  $$\frac{}{\vdash \texttt{mneutrinv}_{mt} : mt \multimap \texttt{unit} \times mt} \qquad \frac{}{\texttt{mneutrinv}_{mt}/x \to \texttt{mneutrinv}_{mt}/((),x)}$$

Their typing rules reflect their behavior. Their reaction rules show that they are stateless machines: these machines always evolve into themselves.

2. The second group of atomic machines are lower-level versions of operators present in the source language: constants, the mux variant of the split and merge operator, ultimately periodic words, clock composition implemented as sums, and buffers.

   a) The constant machine $\boxed{\texttt{mconst}_v}$ is parametrized by a value v of some type dt. It has no input and one output of type dt on which at every reaction it produces the value v.

   MCONST
   $$\frac{\vdash v : mt}{\vdash \texttt{mconst}_v : \texttt{unit} \multimap mt} \qquad \frac{}{\texttt{mconst}_s/() \to \texttt{mconst}_s/s}$$

   b) The *multiplexing* machine $\boxed{\texttt{mmux}_{mt}}$ is the machine equivalent of an "if" statement. It receives a boolean `x` and two inputs `y` and `z` of type mt. The output is either `y` when `x` is equal to 1, or `z` when `x` is equal to 0.

   MMUX
   $$\frac{}{\vdash \texttt{mmux}_{mt} : \{1\} \times mt \times mt \multimap mt}$$
   $$\frac{}{\texttt{mmux}_{mt}/(1,x,y) \to \texttt{mmux}_{mt}/x} \qquad \frac{}{\texttt{mmux}_{mt}/(0,x,y) \to \texttt{mmux}_{mt}/y}$$

c) Next is the first stateful machine, $\boxed{\texttt{mpw}_p(n)}$. This machine implements an ultimately periodic integer word p. The natural number $n \leq |u| + |v| - 1$ acts as an index inside the list $|u|.|v|$ pointing to the next element to produce. Its typing rule expresses that the outputs are bounded by $\lceil p \rceil$, the maximum value of p. It is updated in the expected way.

MPWORD

$$\frac{\vdash n : \{|u|+|v|-1\}}{\vdash \texttt{mpw}_{u(v)}(n) : \texttt{unit} \twoheadrightarrow \{\lceil p \rceil\}} \qquad \frac{i < |u|}{\texttt{mpw}_{u(v)}(i)/() \to \texttt{mpw}_{u(v)}(i+1)/|u|[i]}$$

$$\frac{|u| \leq i < |u|+|v|-1}{\texttt{mpw}_{u(v)}(i)/() \to \texttt{mpw}_{u(v)}(i+1)/v[i-|u|]} \qquad \frac{}{\texttt{mpw}_{u(v)}(|u|+|v|-1)/() \to \texttt{mpw}_{u(v)}(|u|)/v[|v|-1]}$$

The following reactions show the evolution of the internal counter when computing the successive values of $0(4\ 1)^\omega$.

$$\begin{array}{rcl} \texttt{mpw}_{0(4\ 1)}(0)/() & \to & \texttt{mpw}_{0(4\ 1)}(1)/0 \\ \texttt{mpw}_{0(4\ 1)}(1)/() & \to & \texttt{mpw}_{0(4\ 1)}(2)/4 \\ \texttt{mpw}_{0(4\ 1)}(2)/() & \to & \texttt{mpw}_{0(4\ 1)}(1)/1 \end{array}$$

d) The $\boxed{\texttt{msum}_n^m}$ machine receives a list of at most n integers, each of which is smaller than n. It returns the sum of all these integers, whose value is at most $n * m$.

MSUM

$$\frac{}{\vdash \texttt{msum}_{n_1}^{n_2} : \{n_1\}[n_2] \twoheadrightarrow \{n_1 * n_2\}} \qquad \frac{}{\texttt{msum}_{n_1}^{n_2}/xl \to \texttt{msum}_{n_1}^{n_2}/\sum xl}$$

e) The stateful machine $\boxed{\texttt{mbuff}_{n_1,n_2,n_3}^{mt}(c)}$ implements a bounded buffer holding at most n_3 elements. The list c is the current content of the buffer. The naturals n_1 and n_2 bound respectively the amount of data written and read per reaction.

MBUFF

$$\frac{\vdash c : mt[n_3]}{\vdash \texttt{mbuff}_{n_1,n_2,n_3}^{mt}(c) : \{n_2\} \times mt[n_1] \twoheadrightarrow mt[n_2]} \qquad \frac{c.xl = yl.c' \qquad |yl| = a \qquad |c'| \leq n_3}{\texttt{mbuff}_{n_1,n_2,n_3}^{mt}(c)/(a,xl) \to \texttt{mbuff}_{n_1,n_2,n_3}^{mt}(c')/yl}$$

We give some examples of reactions below. The last reaction cannot proceed since the buffer overflows.

$$\begin{array}{rcl} \texttt{mbuff}_{1,2,2}^{\{128\}}([])/(0,[0]) & \to & \texttt{mbuff}_{1,2,2}^{\{128\}}([0])/[] \\ \texttt{mbuff}_{1,2,2}^{\{128\}}([0])/(0,[1]) & \to & \texttt{mbuff}_{1,2,2}^{\{128\}}([0,1])/[] \\ \texttt{mbuff}_{1,2,2}^{\{128\}}([0,1])/(2,[2]) & \to & \texttt{mbuff}_{1,2,2}^{\{128\}}([2])/[0,1] \\ \texttt{mbuff}_{1,2,2}^{\{128\}}([2])/(0,[4,5]) & \not\to & \end{array}$$

3. The remaining group of atomic machines implement operations related to lists—we call them the *list processing* machines. While all these machines are stateless, the scattering, gathering, stuttering and zipping machines perform relatively complex tasks which we describe by predicates. We define each predicate along with the description of the machine.

a) The gathering machine $\boxed{\mathtt{mgath}^{mt}_{n_1,n_2}}$ concatenates a list of lists. Formally, it transforms values of types $mt[n_1][n_2]$ into values of type $mt_1[n_1 * n_2]$, making it the operational counterpart to the **unpack** stream function. Its behavior is characterized by the following predicate, with $\sum$ denoting list concatenation here.

$$gather(xl,yl) \stackrel{\text{def}}{=} yl = \sum_{0\le i<|xl|} xl[i]$$

$$\frac{}{\vdash \mathtt{mgath}^{mt}_{n_1,n_2} : mt[n_1][n_2] \multimap mt[n_1 * n_2]}\ \text{MGATHER} \qquad \frac{gather(xl,yl)}{\mathtt{mgath}^{mt}_{n_1,n_2}/xs \to \mathtt{mgath}^{mt}_{n_1,n_2}/ys}$$

Here are some examples of reactions. Remember that in the type $mt[n]$, the number n describes not the size but the *maximal* size of the list.

$$\begin{aligned}
\mathtt{mgath}^{\{1\}}_{2,3}/[[0,1],[1,0],[0,0]] &\to \mathtt{mgath}^{\{1\}}_{2,3}/[0,1,1,0,0,0]\\
\mathtt{mgath}^{\{1\}}_{2,3}/[[1],[0],[]] &\to \mathtt{mgath}^{\{1\}}_{2,3}/[1,0]\\
\mathtt{mgath}^{\{1\}}_{2,3}/[[],[1,0]] &\to \mathtt{mgath}^{\{1\}}_{2,3}/[1,0]\\
\mathtt{mgath}^{\{1\}}_{2,3}/[[0],[]] &\to \mathtt{mgath}^{\{1\}}_{2,3}/[0]
\end{aligned}$$

b) Symmetrically, the scattering machine $\boxed{\mathtt{mscatt}^{mt}_{n_1,n_2}}$ cuts a list into a list of smaller lists, and is the operational version of **pack**. Similarly to **pack**, in addition to the input list yl to be cut it also expects a list of naturals xl giving the size of each sub-list. This is what the predicate $scatter(xl,yl,zl)$ defined below expresses formally.

$$scatter(xl,yl,zl) \stackrel{\text{def}}{=} gather(zl,yl) \text{ and } \forall 0 \le i < |xl|, |zl[i]| = xl[i]$$

$$\frac{}{\vdash \mathtt{mscatt}^{mt}_{n_1,n_2} : \{n_1\}[n_2] \times mt[n_1 * n_2] \multimap mt[n_1][n_2]}\ \text{MSCATTER} \qquad \frac{scatter(xl,yl,zl)}{\mathtt{mscatt}^{mt}_{n_1,n_2}/(xl,yl) \to \mathtt{mscatt}^{mt}_{n_1,n_2}/zl}$$

Some examples of reactions are given below.

$$\begin{aligned}
\mathtt{mscatt}^{\{1\}}_{2,3}/([0,1,1,0,0,0],[2,2,2]) &\to \mathtt{mscatt}^{\{1\}}_{2,3}/[[0,1],[1,0],[0,0]]\\
\mathtt{mscatt}^{\{1\}}_{2,3}/([1,0],[1,1,0]) &\to \mathtt{mscatt}^{\{1\}}_{2,3}/[[1],[0],[]]\\
\mathtt{mscatt}^{\{1\}}_{2,3}/([1,0],[0,2]) &\to \mathtt{mscatt}^{\{1\}}_{2,3}/[[],[1,0]]\\
\mathtt{mscatt}^{\{1\}}_{2,3}/([[0],[]],[1]) &\to \mathtt{mscatt}^{\{1\}}_{2,3}/[0]
\end{aligned}$$

c) The stuttering machine $\boxed{\mathtt{mstutt}^{mt}_{n}}$ consumes a pair (x,a) with x a value having type mt and $a \le n$ an integer and produces the list v^a, an abbreviation for the list xl that has length a and such that $xl[i] = x$ for any $0 \le i < a$. The output has type $mt[n]$, since $a \le n$.

$$\frac{}{\vdash \mathtt{mstutt}^{n}_{mt} : \{n\} \times mt \multimap mt[n]}\ \text{MSTUTTER} \qquad \frac{}{\mathtt{mstutt}^{n}_{mt}/(a,x) \to \mathtt{mstutt}^{n}_{mt}/x^a}$$

d) The unwrapping machine $\boxed{\texttt{munwrap}_{mt}}$ receives a value of type $mt[1]$. Here, this input should always be a list of size one, and thus of the form $[x]$. The unwrapping machine removes the indirection by returning x. This machine can invert the effect of the stuttering one in the degenerate case of "stuttering" every value once.

$$\textsc{MUnwrap}\quad \frac{}{\vdash \texttt{munwrap}_{mt} : mt[1] \multimap mt} \qquad \frac{}{\texttt{munwrap}_{mt}/[x] \to \texttt{munwrap}_{mt}/x}$$

e) The zipping machine $\boxed{\texttt{mzip}^n_{mt_1,mt_2}}$ transforms a pair of lists (xl, yl) into a list of pairs zl. The two components of the pair should have the same size at each reaction. The predicate $zip(xl, yl, zl)$ below expresses the behavior of the function formally.

$$zip(xl, yl, zl) \stackrel{\text{def}}{=} |xl| = |yl| = |zl| \text{ and } \forall 0 \le i < |xl|, zl[i] = (xl[i], yl[i])$$

$$\textsc{MZip}\quad \frac{}{\vdash \texttt{mzip}^n_{mt_1,mt_2} : mt_1[n] \times mt_2[n] \multimap (mt_1 \times mt_2)[n]} \qquad \frac{zip(xl, yl, zl)}{\texttt{mzip}^n_{mt_1,mt_2}/(xl, yl) \to \texttt{mzip}^n_{mt_1,mt_2}/zl}$$

f) The unzipping machine $\boxed{\texttt{munzip}^n_{mt_1,mt_2}}$ inverts the effect of the previous one. Its reaction rule uses the same predicate but applied symmetrically, the zipped list now being the input rather than the output.

$$\textsc{MUnzip}\quad \frac{}{\vdash \texttt{munzip}^n_{mt_1,mt_2} : (mt_1 \times mt_2)[n] \multimap mt_1[n] \times mt_2[n]}$$

$$\frac{zip(yl, zl, xl)}{\texttt{munzip}^n_{mt_1,mt_2}/xl \to \texttt{munzip}^n_{mt_1,mt_2}/(yl, zl)}$$

g) The bounded integer coercion machine $\boxed{\texttt{mncoer}_{n_1,n_2}}$ takes an integer in $\{n_1\}$ and transforms it into an element of $\{n_2\}$. It reacts only when the input is smaller than n_2.

$$\textsc{MNCoer}\quad \frac{}{\vdash \texttt{mncoer}_{n_1,n_2} : \{n_1\} \multimap \{n_2\}} \qquad \frac{a \le n_2}{\texttt{mncoer}_{n_1,n_2}/a \to \texttt{mncoer}_{n_1,n_2}/a}$$

h) The list size coercion machine $\boxed{\texttt{mlcoer}^{mt}_{n_1,n_2}}$ takes a list in $mt[n_1]$ and transforms it into an element of $mt[n_2]$. Its operational behavior depends on whether n_2 is (strictly) smaller than n_1. If $n_1 \le n_2$ no work is performed, as by definition any value in $mt[n_1]$ is also in $mt[n_2]$. If $n_2 < n_1$, the $n_1 - n_2$ last elements of the input list are dropped.

$$\textsc{MLCoer}\quad \frac{}{\vdash \texttt{mlcoer}^{mt}_{n_1,n_2} : mt[n_1] \multimap mt[n_2]} \qquad \frac{xl = yl.zl \qquad |yl| = min(|xl|, n_2)}{\texttt{mlcoer}^{mt}_{n_1,n_2}/xl \to \texttt{mlcoer}^{mt}_{n_1,n_2}/yl}$$

4. The machines of the last group are not atomic but *combinators* that correspond to various forms of composition, including feedback and replication. They make it possible to build complex machines from simple ones.

a) The (horizontal) composition machine $\boxed{m_1 \bullet m_2}$ consists in plugging the outputs of m_2 into the inputs of m_1. We follow the usual mathematical notation, where conceptually m_2 is computed first.

$$\textsc{MComp}\ \frac{\vdash m_1 : mt_2 \multimap mt_3 \qquad \vdash m_2 : mt_1 \multimap mt_2}{\vdash m_1 \bullet m_2 : mt_1 \multimap mt_3} \qquad \frac{m_1/x \to m_1'/y \qquad m_2/y \to m_2'/z}{m_2 \bullet m_1/x \to m_2' \bullet m_1'/z}$$

b) The (vertical) composition machine $\boxed{m_1 \parallel m_2}$ consists in putting m_1 and m_2 in parallel, with no communication between the two components. We picture this as m_1 above m_2, their inputs and outputs put side-by-side.

$$\textsc{MPar}\ \frac{\vdash m_1 : mt_1 \multimap mt_2 \qquad \vdash m_2 : mt_3 \multimap mt_4}{\vdash m_1 \parallel m_2 : mt_1 \times mt_3 \multimap mt_2 \times mt_4} \qquad \frac{m_1/x_1 \to m_1'/y_1 \qquad m_2/x_2 \to m_2'/y_2}{m_1 \parallel m_2/(x_1,x_2) \to m_1' \parallel m_2'/(y_1,y_2)}$$

c) The feedback machine $\boxed{\mathtt{mfb}^{mt_3}_{mt_1,mt_2}(m)}$ adds a feedback loop around m. This machine should have type $mt_1 \times mt_3 \multimap mt_2 \times mt_3$. The feedback loop plugs the second output back into the second input. One is left with the type $mt_1 \multimap mt_2$. The reaction rule of this machine relies on non-determinism: one may perform a reaction for any value z of type mt_3 that is accepted by m.

$$\textsc{MFeedback}\ \frac{\vdash m : mt_1 \times mt_3 \multimap mt_2 \times mt_3}{\vdash \mathtt{mfb}^{mt_3}_{mt_1,mt_2}(m) : mt_1 \multimap mt_2} \qquad \frac{\vdash z : mt_3 \qquad m/(x,z) \to m'/(y,z)}{\mathtt{mfb}^{mt_3}_{mt_1,mt_2}(m)/x \to \mathtt{mfb}^{mt_3}_{mt_1,mt_2}(m')/y}$$

d) The *replication* machine $\boxed{\mathtt{mrepl}_n(m)}$ executes a variable amount of steps of the machine m. Suppose that m is a machine of type $mt_1 \multimap mt_2$. The replication machine receives a natural $a \leq n$ a list xl which should be of size a. It then performs n reactions of m, each fed with the next element of xl. The resulting list yl of type $mt_2[n]$ is produced.

$$\textsc{MRepl}\ \frac{\vdash m : mt_1 \multimap mt_2}{\vdash \mathtt{mrepl}_n(m) : \{n\} \times mt_1[n] \multimap mt_2[n]} \qquad \frac{m/xl \to_a m/yl}{\mathtt{mrepl}_n(m)/(a,xl) \to \mathtt{mrepl}_n(m)/yl}$$

Remark 17. The operations provided by this machine language might appear very close to the ones present in the synchronous semantics. There are, however, some small but important differences due to the lower-level nature of the machine language.

- Some source-level operators do not have an immediate machine-level counterpart. For instance, the split and merge operators have to be implemented using the multiplexing machine.
- Consider the replication machine $\mathtt{mrepl}_n(m)$. In contrast with the local time scales of the source language, neither scattering nor gathering is involved. Instead, it produces lists of outputs and inputs of the wrapped machine m. When the inputs or outputs of m itself are lists, $\mathtt{mrepl}_n(m)$ reads or writes lists of lists.

These differences and others of the same kind will be dealt with during the translation process.

Graphical representation Figure 4.8 gives the graphical representation of base machines. The framed top left cell shows how a generic machine m is represented as a box whose inputs come from the left and outputs exit to the right. Product value types are represented as independent, parallel wires; wires of unit type are not represented. The rest of the figure displays seven base machines that have a particular graphical representations. We expect those graphical versions to be self-explanatory at this point, as they closely match the explanations of Section 4.2. Note that we do not represent the output of the erasure machine, since it is of unit type. The representation of the exchange machine as a pair of crossing wires is generalized to permutations in the obvious manner. Finally, we do not give dedicated representations to the remaining machines. Instead, they are depicted as opaque boxes with the name of the machine explicitly written.

4.2.2 Metatheoretical Properties

This subsection states and proves properties of the machine language and its type system. The main results are subject reduction and finiteness. The finiteness result shows that our machine language actually corresponds to finite state automata.

Decidability First, as a sanity check, it should be clear that typing is actually decidable. This is not surprising since machines are heavily annotated with their types.

Property 38 (Decidability of Typing)**.** *There is an algorithm that decides for any value v and value type mt whether $\vdash v : mt$. The same is true for machines and machine types.*

Proof. The algorithm can be deduced directly from the form of typing rules of Figure 4.4 and Figure 4.6, since they are syntax directed. □

Remark 18. We could have stated a stronger result: annotations make it possible to find a machine type mtm for a machine m such that $\vdash m : mtm$, if it exists. Note that it is not unique because of the $\mathtt{mconst}_v$ machine. The type of a value is not unique since any natural n belongs to an infinite number of value types $\{n\}$, $\{n+1\}$, and so on. Thus, the MCONST rule does not give a unique type to the $\mathtt{mconst}_v$ machine. In practice, one will probably be interested in the most precise type for a given v, which boils down to $\{n\}$ when v is some natural n.

Reactions, Totality, Determinism With the operational semantics given in Figure 4.7, some machines are neither *reactive*–they have no possible reaction for a given input–nor *deterministic*–they have several possible reactions for the same input. A machine which is not reactive is said to be *partial*. Let us give two examples.

Example 20. Consider the following four machines.

$$m_1 = \mathtt{mdup}_{\{1\}} \bullet \mathtt{mmux}_{\{1\}} \bullet \mathtt{mswap}_{\{1\}\to\{1\}\times\{1\}} \bullet ((\mathtt{mconst}_0 \parallel \mathtt{mconst}_1) \bullet \mathtt{mdup}_{\mathtt{unit}}) \parallel \mathtt{mid}_{\{1\}})$$

$$m_1' = \mathtt{mfb}^{\{1\}}_{\mathtt{unit},\{1\}}(m_1)$$

$$m_2 = \mathtt{mswap}_{\{1\}\to\{1\}} \bullet m_1$$

$$m_2' = \mathtt{mfb}^{\{1\}}_{\mathtt{unit},\{1\}}(m_2)$$

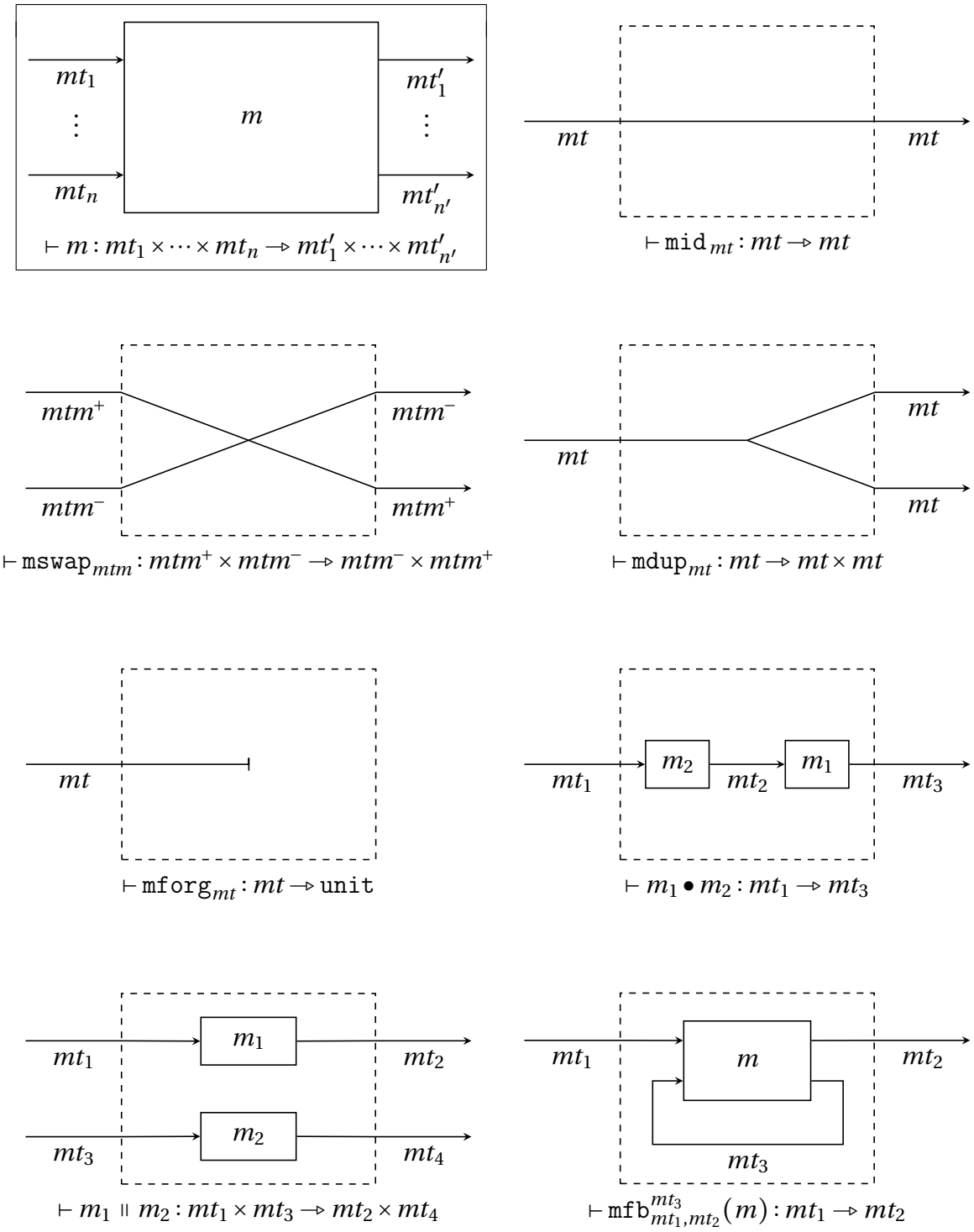


Figure 4.8: Machines - graphical representation

Their types are respectively $\mathtt{unit} \times \{1\} \multimap \{1\} \times \{1\}$ for m_1 and m_2 and $\mathtt{unit} \multimap \{1\}$ for m_1' and m_2'. The machines m_1 and m_2 are deterministic and reactive. By definition of the reaction judgment, m_1' may output a boolean x only if it satisfies the equation $x = \mathit{if}\ x\ \mathit{then}\ 1\ \mathit{else}\ 0$ while the output of m_2' is any y such that $y = \text{if}\ y\ \mathit{then}\ 0\ \mathit{else}\ 1$ holds. The first equation is satisfied by any boolean and the second by none of them. Thus m_1' is not deterministic and m_2' is not reactive. This shows that the feedback operator preserves neither property.

Example 21. Consider the reaction rule for buffers in Figure 4.7. It is perfectly possible that an input *xl* is large enough to overflow the buffer, even if both inputs *a* and *xl* are well-typed. In this case, the premise relating the buffer capacity n_3 and its occupancy $|c'|$ will forbid the reaction, making it partial. For example, in the reaction below, we try to add two values to a half-full buffer whose maximum capacity is 2.

$$\mathtt{mbuff}^{\{128\}}_{2,2,2}([0])/(0,[1,2]) \quad \not\to$$

Conversely, the first input, which indicates the desired amount of data to be consumed, can be too large with respect to the current content of the buffer. In this case, no reaction happen. For example, in the reaction below, we try to consume two values from a buffer holding only one.

$$\mathtt{mbuff}^{\{128\}}_{2,2,2}([0])/(2,[]) \quad \not\to$$

This shows that the type system of machines is not precise enough to capture the absence of buffer overflow or underflow. Similarly, several other machines are able to react only when their inputs have consistent sizes or values, such as scattering and gathering.

While the type system does not rule out partiality or non-determinism, it still serves the important role of enforcing the finiteness of machines. To express this fact we need to show that types are preserved along reactions.

Property 39 (Subject Reduction)**.** *Let m be a machine, mtm a machine type and $i \geq 0$ such that $\vdash m : mtm$. Then, for any machine m', we have both*

- *for any x, y such that $\vdash x : mtm^-$ and $m/x \to m'/y$, then $\vdash y : mtm^+$ and $\vdash m' : mtm$;*
- *for any xl, yl and $n \geq 0$ such that $\vdash^n xl : mtm^-$ and $m/xl \to_n m'/yl$, then $\vdash^n yl : mtm^+$ and $\vdash m' : mtm$. Moreover, $|xl| = |yl| = n$.*

Proof. The proof is a routine induction on derivation trees of the reaction and bulk reaction judgments. The two properties really have to be proved together since the judgments are mutually recursive. Let us discuss some cases.

- The buffer case is immediate because of the explicit premises in its reaction rule. From typing we know that $a \leq n_2$ and hence $|yl| \leq n_2$, and since $|c'| \leq n_3$ we have $\vdash^{n_3} c' : mt$ which proves that $m' = \mathtt{mbuff}^{mt}_{n_1,n_2,n_3}(c')$ is well-typed.
- The feedback case relies on the premise $\vdash z : mt_3$ to deduce $\vdash (x,z) : mt_1 \times mt_3$ and apply the induction hypothesis.

- Assume the replication machine processes inputs (a, xl) and produces yl for the current reaction, evolving into machine m'. We have $a \leq n$ and furthermore $|xl| = |yl| = a$ by definition of $m/xl \to_a m'/yl$, and thus $|yl| \leq n$ which means that yl is well-typed. □

Finiteness The operational semantics models state change by rewriting machines. A consequence is that when $m/x \to m'/y$, the machine m and m' are closely related: we would like to express formally that they differ only by their state. Such machines are said to have the same *skeleton*, written $m \equiv m'$. We define this relation to be the smallest congruence relating buffers and ultimately periodic words that differ only by their state, as these are the only primitive stateful machines. Formally, let $\equiv$ be the smallest relation such that

$$\mathtt{mbuff}^{mt}_{n_1,n_2,n_3}(c) \equiv \mathtt{mbuff}^{mt}_{n_1,n_2,n_3}(c') \qquad\qquad \mathtt{mpw}_p(i) \equiv \mathtt{mpw}_p(i')$$

and, abusing notation, write $\equiv$ again for the smallest congruence containing it. We say that the $\equiv$-equivalence class of a machine m is its skeleton.

Property 40. *Let m, m' be machines and x, y be values such that $m/x \to m'/y$. Then m and m' have the same skeleton.*

Proof. Immediate by induction on the reaction judgment. □

Property 41. *For any type mtm, the skeleton of machines of type mtm is a finite set.*

Proof. By induction on machines. The proof is immediate in all cases but the buffer and periodic word ones. Equivalent buffer machines $\mathtt{mbuff}^{mt}_{n_1,n_2,n_3}(c)$ differ in their content c but share the same annotations, in particular mt and n_3. Because c is of type $mt[n_3]$ in well-typed machines, and there is only a finite number of values inhabiting any value type, the number of buffer machines is bounded. The reasoning is similar for equivalent periodic word machines $\mathtt{mpw}_{u(v)}(i)$, which differ only in their counter i which is of type $\{|u| + |v| - 1\}$. □

We say that a machine m' is *reachable* from a machine $\vdash m : mtm$ if there exists a natural number n, a list $\vdash^n xl : mtm^-$ and a list yl such that $m/xl \to_n m'/yl$.

Corollary 1. *The set of machines reachable from any well-typed machine is finite.*

Proof. Combine Property 40 and Property 41. □

This result suggests a very crude manner to translate a machine $\vdash m : mtm$ into a transducer. Its skeleton becomes the set of states of the transducer. To build the transition relation, check for any m' in the skeleton of m and values $\vdash x : mtm^-$ and $\vdash y : mtm^+$ whether $m/x \to m'/y$, and add the transition if necessary. We know that checking for well-typed outputs is enough, thanks to Property 39. The resulting transducer is finite-state by construction, and can in theory be implemented as a digital synchronous circuit.

This construction is of dubious practical interest since it explicitly builds an automaton whose size is exponential in the size of the corresponding machine. We discuss a more realistic and enlightening way of implementing machines in traditional hardware description languages in Section 4.5. But let us first explain the last ingredient required by the translation.

4.2.3 Macro-Machines

Let us present some macro-machines that bridge the gap between source-level operators and our basic machines.

The simplest macro-machines bridge the gap between low-level machines and source-level constructions. Figure 4.9 gives the definition of each macro-machine as well as its typing rule, even if the latter can always be reconstructed from the definition of the machine and the rules of Figure 4.6.

- The first macro-machine implements a permutation of an arbitrary number of inputs. We write $\boxed{\langle mt_1, \ldots, mt_n \mapsto i_1, \ldots, i_n \rangle}$ for the machine with n inputs and outputs, and whose j-th output is equal to its i_j-th input. To model an actual permutation, all the i_n should be distinct and between 0 and $n-1$. Its typing rule follows.

$$\frac{}{\vdash \langle mt_1, \ldots, mt_n \mapsto i_1, \ldots, i_n \rangle : mt_1 \times \ldots \times mt_n \multimap mt_{i_1} \times \ldots \times mt_{i_n}}$$

  Any such permutation can be implemented by combining $\mathtt{mswap}_{mtm}$ machines in the appropriate way by decomposing the permutation into a product of transpositions. In practice we only use small permutations and thus neglect the performance issues that may arise from the precise choice of decomposition.

- The *left-plugging* macro-machine $\boxed{\mathtt{mapply}^{mt}_{mtm}(m_1, m_2)}$ provides a composition pattern that frequently occurs when defining other macro-machines. The machine m_1 has no input and one output, which is plugged by this macro-machine into the the first input m_2.

$$\frac{\vdash m_1 : \mathtt{unit} \multimap mt \qquad \vdash m_2 : mt \times mtm^- \multimap mtm^+}{\vdash \mathtt{mapply}^{mt}_{mtm}(m_1, m_2) : mtm}$$

  Its definition involves a certain amount of plumbing.

$$\mathtt{mapply}^{mt}_{mtm}(m_1, m_2) \quad = \quad m_2 \bullet (m_1 \parallel \mathtt{mid}_{mtm^-}) \bullet \mathtt{mneutrinv}_{mtm^-}$$

- Several base machines receive a first input which controls their behavior. This is the case for the replication, gathering, and scattering machines for instance. It is convenient to define macro counterparts in which this input is provided by a fixed machine.

  – The first input of the basic replication machine determines how many steps of the replicated machine must be performed. In $\boxed{\mathtt{mdrivect}^n_{mtm}(m_1, m_2)}$, this information is directly provided by the output of the machine m_1, which has no inputs.

$$\frac{\vdash m_1 : \mathtt{unit} \multimap \{n\} \qquad \vdash m_2 : mtm}{\vdash \mathtt{mdrivect}^n_{mtm}(m_1, m_2) : mtm[n]}$$

  Its definition uses the left-plugging machine.

$$\mathtt{mdrivenict}^n_{mt}(m_1, m_2) \quad = \quad \mathtt{mapply}^{\{n\}}_{mtm}(m_1, \mathtt{mrepl}_n(m_2))$$

$$\frac{}{\vdash \langle mt_1,\ldots,mt_n \mapsto i_1,\ldots,i_n \rangle : mt_1 \times \ldots \times mt_n \twoheadrightarrow mt_{i_1} \times \ldots \times mt_{i_n}}$$

$$\frac{\vdash m_1 : \mathtt{unit} \twoheadrightarrow mt \qquad \vdash m_2 : mt \times mtm^- \twoheadrightarrow mtm^+}{\vdash \mathtt{mapply}^{mt}_{mtm}(m_1, m_2) : mtm}$$

$$\frac{\vdash m_1 : \mathtt{unit} \twoheadrightarrow \{n\} \qquad \vdash m_2 : mtm}{\vdash \mathtt{mdrivect}^{n}_{mtm}(m_1, m_2) : mtm[n]} \qquad \frac{\vdash m_1 : \mathtt{unit} \twoheadrightarrow \{n\} \qquad \vdash m_2 : \mathtt{unit} \twoheadrightarrow mt}{\vdash \mathtt{mdrivenict}^{n}_{mt}(m_1, m_2) : \mathtt{unit} \twoheadrightarrow mt[n]}$$

$$\frac{\vdash m_1 : \mathtt{unit} \twoheadrightarrow \{n_1\} \qquad \vdash m_2 : \mathtt{unit} \twoheadrightarrow \{n_2\}}{\vdash \mathtt{mscattct}^{mt}_{n_1,n_2}(m_1, m_2) : mt[n_1 * n_2] \twoheadrightarrow mt[n_1][n_2]} \qquad \frac{\vdash m_1 : \mathtt{unit} \twoheadrightarrow \{n\}}{\vdash \mathtt{mstutterct}^{n}_{mt}(m) : mt \twoheadrightarrow mt[n]}$$

$$\frac{\vdash m : \mathtt{unit} \twoheadrightarrow \{n_2\}}{\vdash \mathtt{mbuffct}^{mt}_{n_1,n_2,n_3}(m) : mt[n_1] \twoheadrightarrow mt[n_2]} \qquad \frac{}{\vdash \mathtt{mwrap}_{mt} : mt \twoheadrightarrow mt[1]}$$

$$\frac{\vdash m : \mathtt{unit} \twoheadrightarrow \{1\}}{\vdash \mathtt{msrcmerge}_{mt}(m) : mt[1] \times mt[1] \twoheadrightarrow mt[1]} \qquad \frac{\vdash m : \mathtt{unit} \twoheadrightarrow \{1\}}{\vdash \mathtt{msrcwhen}_{mt}(m) : mt[1] \twoheadrightarrow mt[1]}$$

(a) Typing rules - can be deduced from the expansions

$$\begin{aligned}
\mathtt{mapply}^{mt}_{mtm}(m_1, m_2) &= m_2 \bullet (m_1 \parallel \mathtt{mid}_{mtm^-}) \bullet \mathtt{mneutrinv}_{mtm^-} \\
\mathtt{mdrivect}^{n}_{mtm}(m_1, m_2) &= \mathtt{mapply}^{\{n\}}_{mtm}(m_1, \mathtt{mrepl}_n(m_2)) \\
\mathtt{mdrivenict}^{n}_{mt}(m_1, m_2) &= \mathtt{mdrivect}^{n}_{\{n\} \twoheadrightarrow mt[n]}(m_1, m_2) \\
&\quad \bullet \mathtt{mapply}^{\{n\}}_{\mathtt{unit} \twoheadrightarrow \mathtt{unit}[n]}(m_1, \mathtt{mstutt}^{n_1}_{\mathtt{unit}}) \\
\mathtt{mstutterct}^{n}_{mt}(m) &= \mathtt{mapply}^{\{n\}}_{mt \twoheadrightarrow mt[n]}(m, \mathtt{mstutt}^{n}_{mt}) \\
\mathtt{mscattct}^{mt}_{n_1,n_2}(m_1, m_2) &= \mathtt{mapply}^{\{n_1\}[n_2]}_{mt[n_1 * n_2] \twoheadrightarrow mt[n_1][n_2]}(m, \mathtt{mscatt}^{mt}_{n_1,n_2}) \\
&\quad \text{where } m = \mathtt{mdrivenict}^{n_1}_{\{n_2\}}(m_1, m_2) \\
\mathtt{mbuffct}^{mt}_{n_1,n_2,n_3}(m) &= \mathtt{mapply}^{\{n\}}_{mt[n_1] \twoheadrightarrow mt[n_2]}(m, \mathtt{mbuff}^{mt}_{n_1,n_2,n_3}(\epsilon)) \\
\mathtt{mwrap}_{mt} &= \mathtt{mstutterct}^{1}_{mt}(\mathtt{mconst}_1) \\
\mathtt{msrcmerge}_{mt}(m) &= \mathtt{mapply}^{\{1\}}_{mt[1] \twoheadrightarrow mt[1] \times mt[1]}(m, m') \\
&\quad \text{where } m' = \mathtt{mmux}_{mt[1] \times mt[1]} \\
&\qquad \bullet \langle mt[1], mt[1], \{1\}, mt[1], mt[1] \mapsto 2,3,0,4,1 \rangle \\
&\qquad \bullet ((\mathtt{mdup}_{mt[1]} \bullet \mathtt{mconst}_{[]}) \parallel \mathtt{mid}_{\{1\}} \parallel \mathtt{mdup}_{mt[1]}) \\
&\qquad \bullet \mathtt{mneutrinv}_{mt} \\
\mathtt{msrcwhen}_{mt}(m) &= \mathtt{mapply}^{\{1\}}_{mt[1] \twoheadrightarrow mt[1]}(m, m') \\
&\quad \text{where } m' = \mathtt{mmux}_{mt[1]} \bullet (\mathtt{mid}_{\{1\}} \parallel \mathtt{mid}_{mt} \parallel \mathtt{mconst}_{[]}) \\
&\qquad \bullet \langle \mathtt{unit}, \{1\}, mt \mapsto 1,2,0 \rangle \bullet \mathtt{mneutrinv}_{mt}
\end{aligned}$$

(b) Expansions

Figure 4.9: First-order macro-machines - typing and expansions

– The *no-input* replication macro-machine $\boxed{\texttt{mdrivenict}^{n}_{mt}(m_1,m_2)}$ is a variant of the previous machine useful when m_2 has no inputs. In this case, the previous macro-machine gives has type $\texttt{unit}[n] \multimap\!\triangleright\, mt_2[n]$, which is not very convenient; we generally prefer $\texttt{unit} \multimap\!\triangleright\, mt_2[n]$. The stuttering machine bridges the gap.

$$\begin{array}{rcl}\texttt{mdrivenict}^{n}_{mt}(m_1,m_2) & = & \texttt{mdrivect}^{n}_{\{n\}\multimap\!\triangleright mt[n]}(m_1,m_2) \\ & & \bullet\, \texttt{mapply}^{\{n\}}_{\texttt{unit}\multimap\!\triangleright\texttt{unit}[n]}(m_1,\texttt{mstutt}^{n}_{\texttt{unit}})\end{array}$$

– The first input of the basic scattering machine determines the shape of the output list. This input is a list of integers of type $\{n_1\}[n_2]$. Its size determines the size of the output list of lists, bounded by n_2, and its elements determine the size of each sub-list, bounded by n_1. In the macro-machine $\boxed{\texttt{mscattct}^{mt}_{n_1,n_2}(m_1,m_2)}$, the machine m_1 provides the size of the list of lists, bounded by n_1, and the machine m_2 provides the size of each sub-list.

$$\frac{\vdash m_1 : \texttt{unit} \multimap\!\triangleright \{n_1\} \qquad \vdash m_2 : \texttt{unit} \multimap\!\triangleright \{n_2\}}{\vdash \texttt{mscattct}^{mt}_{n_1,n_2}(m_1,m_2) : mt[n_1 * n_2] \multimap\!\triangleright mt[n_1][n_2]}$$

This relies again on the left-plugging machine. We use the no-input replication macro-machine to combine m_1 and m_2 in order to obtain the list of sizes, which has type $\{n_1\}[n_2]$.

$$\begin{array}{rcl}\texttt{mscattct}^{mt}_{n_1,n_2}(m_1,m_2) & = & \texttt{mapply}^{\{n_1\}[n_2]}_{mt[n_1*n_2]\multimap\!\triangleright mt[n_1][n_2]}(m,\texttt{mscatt}^{mt}_{n_1,n_2}) \\ & & \text{where } m = \texttt{mdrivenict}^{n_2}_{\{n_1\}}(m_2,m_1)\end{array}$$

– The first input of the basic stuttering machine determines how many copies of its second input have to be produced. In the macro-machine $\boxed{\texttt{mstutterct}^{n}_{mt}(m)}$ this input is provided by the machine m.

$$\frac{\vdash m_1 : \texttt{unit} \multimap\!\triangleright \{n\}}{\vdash \texttt{mstutterct}^{n}_{mt}(m) : mt \multimap\!\triangleright mt[n]}$$

As before, we use the left-plugging macro machine.

$$\texttt{mstutterct}^{n}_{mt}(m) \;=\; \texttt{mapply}^{\{n\}}_{mt\multimap\!\triangleright mt[n]}(m,\texttt{mstutt}^{n}_{mt})$$

– The first input of the basic buffer machine determines the amount of data that has to be produced for the current reaction. In $\boxed{\texttt{mbuffct}^{mt}_{n_1,n_2,n_3}(m)}$ this input is provided by the machine m.

$$\frac{\vdash m : \texttt{unit} \multimap\!\triangleright \{n_2\}}{\vdash \texttt{mbuffct}^{mt}_{n_1,n_2,n_3}(m) : mt[n_1] \multimap\!\triangleright mt[n_2]}$$

Again, we use the left-plugging macro machine.

$$\texttt{mbuffct}^{mt}_{n_1,n_2,n_3}(m) \;=\; \texttt{mapply}^{\{n\}}_{mt[n_1]\multimap\!\triangleright mt[n_2]}(m,\texttt{mbuff}^{mt}_{n_1,n_2,n_3}([]))$$

- Our language includes the list unwrapping machine that transforms a list of size one $[x]$ into an element x. We sometimes need the symmetric macro-machine $\boxed{\mathtt{mwrap}_{mt}}$ that wraps a value x into a list of size one, returning $[x]$.

$$\frac{}{\vdash \mathtt{mwrap}_{mt} : mt \multimap mt[1]}$$

  This machine is a special case of stuttering.

$$\mathtt{mwrap}_{mt} \;=\; \mathtt{mstutterct}^{1}_{mt}(\mathtt{mconst}_{1})$$

- The last two macro-machines, $\boxed{\mathtt{msrcmerge}_{mt}(m)}$ and $\boxed{\mathtt{msrcwhen}_{mt}(m)}$, implement low-level variants of the stream merging and sampling operators present in the source language. In both cases the machine m provides the condition controling which elements should be merged or sampled.

$$\frac{\vdash m : \mathtt{unit} \multimap \{1\}}{\vdash \mathtt{msrcmerge}_{mt}(m) : mt[1] \times mt[1] \multimap mt[1]} \qquad \frac{\vdash m : \mathtt{unit} \multimap \{1\}}{\vdash \mathtt{msrcwhen}_{mt}(m) : mt[1] \multimap mt[1]}$$

  The merging machine, given an input $[x]$, selects $([x],[])$ when the current output of m is 1 and $([],[x])$ when the current output of m is 0.

$$\begin{aligned}\mathtt{msrcmerge}_{mt}(m) \;=\; & \mathtt{mapply}^{\{1\}}_{mt[1]\multimap mt[1]\times mt[1]}(m, m') \\ & \text{where } m' = \mathtt{mmux}_{mt[1]\times mt[1]} \\ & \qquad \bullet \,\langle mt[1], mt[1], \{1\}, mt[1], mt[1] \mapsto 2,3,0,4,1 \rangle \\ & \qquad \bullet \,((\mathtt{mdup}_{mt[1]} \bullet \mathtt{mconst}_{[]}) \parallel \mathtt{mid}_{\{1\}} \parallel \mathtt{mdup}_{mt[1]}) \\ & \qquad \bullet \,\mathtt{mneutrinv}_{mt}\end{aligned}$$

  The sampling machine, given an input $[x]$, selects $[x]$ when the current output of m is 1 and $[]$ when the current output of m is 0.

$$\begin{aligned}\mathtt{msrcwhen}_{mt}(m) \;=\; & \mathtt{mapply}^{\{1\}}_{mt[1]\multimap mt[1]}(m, m') \\ & \text{where } m' = \mathtt{mmux}_{mt[1]} \bullet (\mathtt{mid}_{\{1\}} \parallel \mathtt{mid}_{mt} \parallel \mathtt{mconst}_{[]}) \\ & \qquad \bullet \,\langle \mathtt{unit}, \{1\}, mt \mapsto 1,2,0 \rangle \bullet \mathtt{mneutrinv}_{mt}\end{aligned}$$

4.3 Linear Higher-Order Machines

Circuits, and thus machines, are intrinsically first-order. We have already explained informally in Section 3.2 how linearity provides a partial solution to the problem of implementing higher-order functions on top of first-order machines. In the rest of this section we implement this intuition by building higher-order "macro" types on top of machine types. We then build a series of macro-machines inhabiting the higher-order macro types. They provide the usual tools of functional programming, such as application, currying, and so on. We also wrap the ad-hoc operators of the base language—buffers, scattering, and wrapping for instance—into simple conversion code to make them compatible with the higher-order combinators.

mtm	+=	$\texttt{munit}$	Unit machine type
	|	$\texttt{mint}\{n\}$	Bounded machine type
	|	$mtm \boxtimes mtm$	Product machine type
	|	$mtm[n]$	List machine type
	|	mtm^*	Reversed machine type
	|	$mtm \boxminus mtm$	Arrow machine type

Figure 4.10: Higher-order macro-machines - syntax of types

4.3.1 Types

Programming with linear higher-order machines involves a relatively high number of new types of machines. Even if all of these are macros that ultimately are to be defined in terms of the base language, we find it easier to present them as a language extension. In addition, a large number of new machines actually lift the base ones to work in the higher-order setting, and it is convenient to express them as syntax. Figure 4.12 extends the grammars of Figure 4.3 with the new types. Let us explain the two syntactic categories in turn.

The base machine type presented in Figure 4.5 is very simple: a machine is simply a pair of value types, one for inputs and one for outputs. Figure 4.10 enriches it with the missing type formers found in traditional functional languages such as the simply-typed lambda-calculus. First, we have the product of machine types $mtm_1 \boxtimes mtm_2$. It is to be thought as the "vertical juxtaposition" of the inputs of mtm_1 and mtm_2. This new product justifies the addition of its neutral element, the unit machine type $\texttt{munit}$ which is the type of machine with no input nor input. It is also be useful to lift the list constructors from values to machines as $mtm[n]$. This corresponds to a machine consuming lists of type $mtm^-[n]$ and producing lists of type $mtm^+[n]$. The type mtm^* may feel unfamiliar; intuitively, if a machine of type mtm can be seen as *producing* an inhabitant of mtm, a machine of type mtm^* can be seen as *consuming* an inhabitant of mtm. We say that mtm and mtm^* are *complementary* types. In more concrete terms, mtm^* is the type mtm with its inputs and outputs reversed. Finally, the type $mtm_1 \boxminus mtm_2$ is the type of machines that conceptually "receive" a machine of type mtm_1, use it only once, and "return" a machine of type mtm_2. We assume that products bind tighter than arrows and thus $mtm_1 \boxminus mtm_1 \boxtimes mtm_2$ is short for $mtm_1 \boxminus (mtm_1 \boxtimes mtm_2)$. Finally, the type $\texttt{mint}\{n\}$ is the type for machines without inputs and returning an integer bounded by n.

Type expansion Figure 4.11 defines the expansion of the new type formers from the previous paragraph in terms of the base machine and value type. The first four cases closely match the preceding informal explanations, including for the reversed machine type mtm^*. To explain the expansion of the machine arrow type $mtm_1 \boxminus mtm_2$, remember the intuitions given in Section 3.2. In the simplest case, the machine is actually a first-order one: then mtm_1 is empty, or more precisely is equal to $\texttt{unit} \multimap \texttt{unit}$. Then, $mtm_1 \boxminus mtm_2$ should be morally the same as mtm_2, and thus include as inputs the inputs of mtm_2 and as outputs the outputs

$$
\begin{array}{lcl}
\mathtt{munit} & \stackrel{\text{def}}{=} & \mathtt{unit} \multimap \mathtt{unit} \\
\mathtt{mint}\{n\} & \stackrel{\text{def}}{=} & \mathtt{unit} \multimap \{n\} \\
mtm_1 \boxtimes mtm_2 & \stackrel{\text{def}}{=} & mtm_1^- \times mtm_2^- \multimap mtm_1^+ \times mtm_2^+ \\
mtm[n] & \stackrel{\text{def}}{=} & mtm^-[n] \multimap mtm^+[n] \\
mtm^* & \stackrel{\text{def}}{=} & mtm^+ \multimap mtm^- \\
mtm_1 \boxminus mtm_2 & \stackrel{\text{def}}{=} & mtm_1^* \boxtimes mtm_2
\end{array}
$$

Figure 4.11: Higher-order macro-machines - expansions of types

of mtm_2. Now, suppose that mtm_1 has no input: the only thing m can do is access its output, which should thus be added *as inputs* to $mtm_1 \boxminus mtm_2$. Now, if mtm_1 actually has inputs, these should be provided by m, and thus appear as additional outputs of m. This leads us to the idea that $mtm_1 \boxminus mtm_2$ is the juxtaposition of mtm_1, with *inputs and outputs reversed*, and mtm_2. This is exactly the definition given in Figure 4.11. Completely expanding the definition gives $mtm_1 \boxminus mtm_2 = mtm_1^+ \times mtm_2^- \multimap mtm_1^- \times mtm_2^+$.

4.3.2 Machines and their expansions

Figure 4.12 gives the syntax of the new macro machines, and Figure 4.13 their derived typing rules. As before, rules can always be deduced from definitions, but are recalled for the sake of clarity. They are of three kinds.

1. There are macro-machines which provide new operations related to higher-order functions, such as currying or application. They are implemented using permutations and the feedback operator.

2. There are macro-machines which wrap machines from the base language to give them higher-order types. For instance, a machine m of type $\{n_1\} \times \{n_2\} \multimap \{n_3\}$ would be lifted to $\mathtt{mint}\{n_1\} \boxtimes \mathtt{mint}\{n_2\} \boxminus \mathtt{mint}\{n_3\}$ using permutations.

3. There are machines that correspond to frequently occurring patterns in the translation, like the low-level machines described at the beginning of this section.

We describe each one of them, its type, and its expansion in terms of base machines. Since the expansions are not very readable, we draw some of them as formal circuits in Figure 4.14.

- The $\boxed{\mathtt{mhoid}_{mtm}}$ machine is the *higher-order identity machine*. It should simply pass its argument along. Unfolding the macro-type $mtm \boxminus mtm$, we obtain $mtm^+ \times mtm^- \multimap mtm^- \times mtm^+$, which is exactly that of the first-order exchange machine. This leads to the expansion

$$\mathtt{mhoid}_{mtm} \stackrel{\text{def}}{=} \mathtt{mswap}_{mtm}$$

m	+=	$\texttt{mhoid}_{mtm}$	Higher-order identity
	|	$\texttt{mhoneutr}_{mtm}$	Higher-order neutralization
	|	$\texttt{mhoneutrinv}_{mtm}$	Higher-order generation
	|	$\texttt{mhoconst}^{n}_{n}$	Lifted constant
	|	$\texttt{mhosum}^{n_2}_{n_1}$	Lifted sum
	|	$\texttt{mhomerge}_{mt}(m)$	Lifted merging
	|	$\texttt{mhowhen}_{mt}(m)$	Lifted sampling
	|	$\texttt{mhoev}_{mtm,mtm}$	Evaluation machine
	|	$\texttt{mplug}(m,m)$	Application combinator
	|	$\texttt{mhogath}^{mtm}_{n,n}(m,m)$	Lifted scattering
	|	$\texttt{mhoscatt}^{mtm}_{n,n}(m,m)$	Lifted gathering
	|	$\texttt{mhowrap}_{mtm}$	Lifted wrapping
	|	$\texttt{mhounwrap}_{mtm}$	Lifted unwrapping
	|	$\texttt{mhozip}^{n}_{mtm,mtm}$	Lifted zip
	|	$\texttt{mhounzip}^{n}_{mtm,mtm}$	Lifted unzip
	|	$\texttt{mhoncoer}_{n,n}$	Lifted bounded integer coercion
	|	$\texttt{mholcoer}^{mtm}_{n,n}$	Lifted list size coercion
	|	$m \boxdot m$	Higher-order horizontal composition
	|	$m \boxtimes m$	Higher-order vertical composition
	|	$\texttt{mhofb}^{mtm}_{mtm,mtm}(m)$	Higher-order feedback loop
	|	$\texttt{mhorepl}^{n}_{mtm,mtm}(m)$	Higher-order replication
	|	$\texttt{mcurry}^{mtm}_{mtm,mtm}$	Currying
	|	$\texttt{muncurry}^{mtm}_{mtm,mtm}$	Decurrying
	|	$\texttt{minterpose}(m,m)$	Interposition

Figure 4.12: Higher-order macro-machines - syntax

whose graphical representation was already depicted in Figure 4.8. This figure shows that the exchange machine performs no work of its own, simply acting as some kind of relay between the inputs/outputs on the upper row, which correspond to mtm^*, and the inputs/outputs of the lower row, which correspond to mtm. Requests of type mtm^- transmitted to $\texttt{mhoid}_{mtm}$ are passed to its argument, while the outputs received from its argument are passed back as answers.

- The *higher-order neutralization machine* $\boxed{\texttt{mhoneutr}_{mtm}}$ is the higher-order counterpart to the neutralization machine. It has the same type has the neutralization machine, except that `unit` has been replaced with `munit`, $\times$ with $\boxtimes$ and $\multimap$ with $\boxminus$. This gives $(\texttt{munit} \boxtimes mtm) \boxminus mtm$, which is in fact equal to $\texttt{munit} \boxtimes (mtm \boxminus mtm)$. This justifies the expansion below.

$$\texttt{mhoneutr}_{mtm} \stackrel{\text{def}}{=} \texttt{mid}_{\texttt{unit}} \parallel \texttt{mhoid}_{mtm}$$

$$\frac{}{\vdash \mathtt{mhoid}_{mtm} : mtm \boxminus mtm} \qquad \frac{}{\vdash \mathtt{mhoneutr}_{mtm} : \mathtt{munit} \boxtimes mtm \boxminus mtm}$$

$$\frac{}{\vdash \mathtt{mhoneutrinv}_{mtm} : mtm \boxminus \mathtt{munit} \boxtimes mtm} \qquad \frac{\vdash n : \{m\}}{\vdash \mathtt{mhoconst}^{n}_{m} : \mathtt{unit} \times \mathtt{unit} \multimap \mathtt{unit} \times \{m\}[1]}$$

$$\frac{}{\vdash \mathtt{mhosum}^{n_2}_{n_1} : \mathtt{mint}\{n_1\}[n_2] \boxminus \{n_1 \times n_2\}}$$

$$\frac{\vdash m : \mathtt{unit} \multimap \{1\}}{\vdash \mathtt{mhomerge}_{mt}(m) : \mathtt{unit} \times mt[1] \times mt[1] \times \mathtt{unit} \multimap \mathtt{unit} \times \mathtt{unit} \times \mathtt{unit} \times mt[1]}$$

$$\frac{\vdash m : \mathtt{unit} \multimap \{1\}}{\vdash \mathtt{mhowhen}_{mt}(m) : \mathtt{unit} \times mt[1] \times \mathtt{unit} \multimap \mathtt{unit} \times \mathtt{unit} \times mt[1]}$$

$$\frac{}{\vdash \mathtt{mhoev}_{mtm_1,mtm_2} : ((mtm_1 \boxminus mtm_2) \boxtimes mtm_1) \boxminus mtm_2} \qquad \frac{\vdash m_1 : mtm_1 \boxminus mtm_2 \qquad \vdash m_2 : mtm_1}{\vdash \mathtt{mplug}(m_1, m_2) : mtm_2}$$

$$\frac{\vdash m_1 : \mathtt{mint}\{n_1\} \qquad \vdash m_2 : \mathtt{mint}\{n_2\}}{\vdash \mathtt{mhogath}^{mtm}_{n_1,n_2}(m_1, m_2) : mtm[n_1][n_2] \boxminus mtm[n_1 * n_2]}$$

$$\frac{\vdash m_1 : \mathtt{mint}\{n_1\} \qquad \vdash m_2 : \mathtt{mint}\{n_2\}}{\vdash \mathtt{mhoscatt}^{mtm}_{n_1,n_2}(m_1, m_2) : mtm[n_1 * n_2] \boxminus mtm[n_1][n_2]} \qquad \frac{}{\vdash \mathtt{mhowrap}_{mtm} : mtm \boxminus mtm[1]}$$

$$\frac{}{\vdash \mathtt{mhounwrap}_{mtm} : mtm[1] \boxminus mtm} \qquad \frac{}{\vdash \mathtt{mhozip}^{n}_{mtm_1,mtm_2} : (mtm_1[n] \boxtimes mtm_2[n]) \boxminus (mtm_1 \boxtimes mtm_2)[n]}$$

$$\frac{}{\vdash \mathtt{mhounzip}^{n}_{mtm_1,mtm_2} : (mtm_1 \boxtimes mtm_2)[n] \boxminus (mtm_1[n] \boxtimes mtm_2[n])} \qquad \frac{}{\vdash \mathtt{mhoncoer}_{n_1,n_2} : \{n_1\} \boxminus \{n_2\}}$$

$$\frac{}{\vdash \mathtt{mholcoer}^{mtm}_{n_1,n_2} : mtm[n_1] \boxminus mtm[n_2]} \qquad \frac{\vdash m_2 : mtm_2 \boxminus mtm_3 \qquad \vdash m_1 : mtm_1 \boxminus mtm_2}{\vdash m_2 \boxdot m_1 : mtm_1 \boxminus mtm_3}$$

$$\frac{\vdash m_1 : mtm_1 \boxminus mtm_1' \qquad \vdash m_2 : mtm_2 \boxminus mtm_2'}{\vdash m_1 \boxtimes m_2 : (mtm_1 \boxtimes mtm_2) \boxminus (mtm_1' \boxtimes mtm_2')} \qquad \frac{\vdash m : (mtm_1 \boxtimes mtm_3) \boxminus (mtm_2 \boxtimes mtm_3)}{\vdash \mathtt{mhofb}^{mtm_3}_{mtm_1,mtm_2}(m) : mtm_1 \boxminus mtm_2}$$

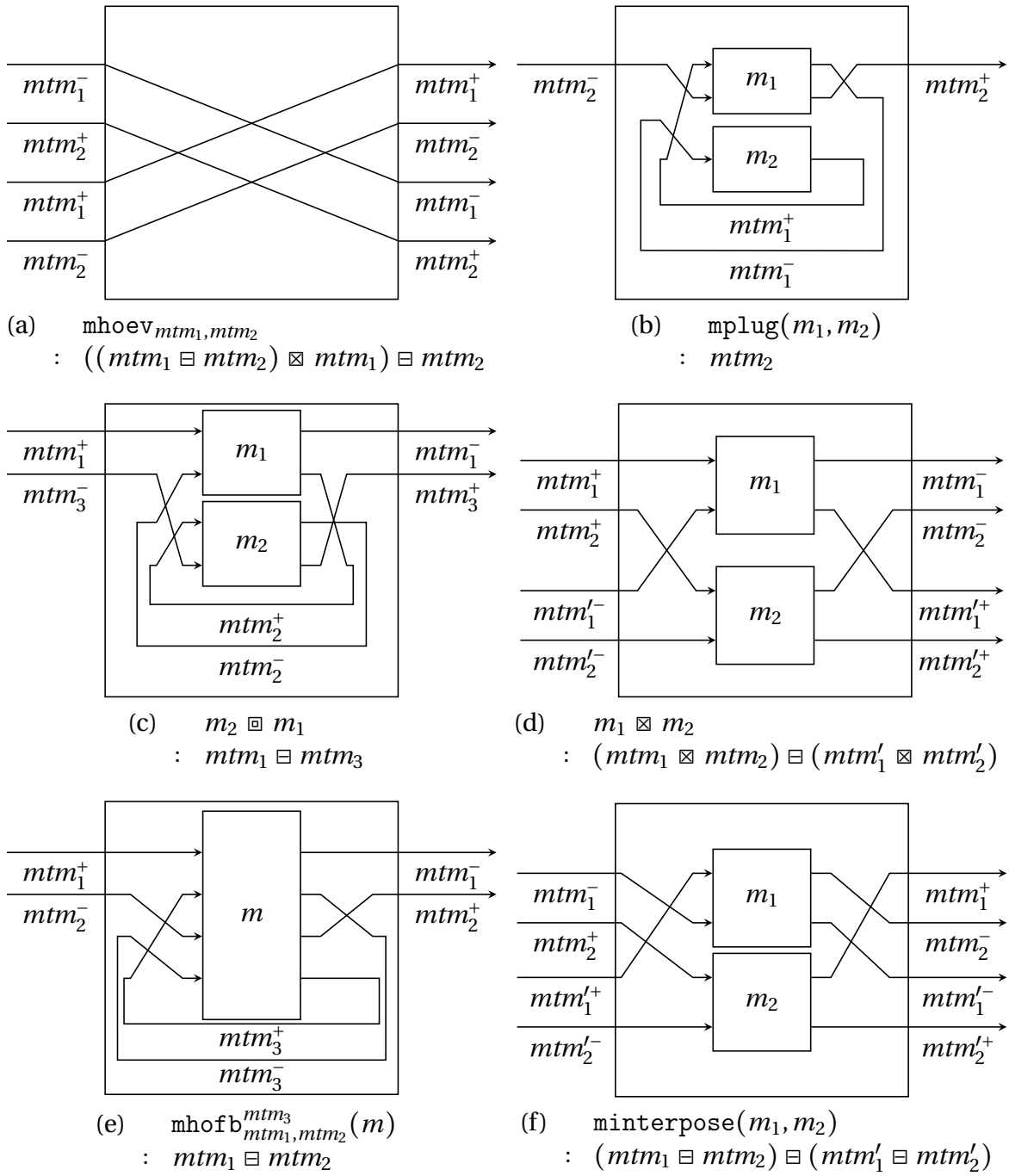
$$\frac{\vdash m : mtm_1 \boxminus mtm_2}{\vdash \mathtt{mhorepl}^{n}_{mtm_1,mtm_2}(m) : \mathtt{mint}\{n\} \boxminus mtm_1[n] \boxminus mtm_2[n]}$$

$$\frac{}{\vdash \mathtt{mcurry}^{mtm_3}_{mtm_1,mtm_2} : ((mtm_1 \boxtimes mtm_2) \boxminus mtm_3) \boxminus (mtm_1 \boxminus (mtm_2 \boxminus mtm_3))}$$

$$\frac{}{\vdash \mathtt{muncurry}^{mtm_3}_{mtm_1,mtm_2} : (mtm_1 \boxminus (mtm_2 \boxminus mtm_3)) \boxminus ((mtm_1 \boxtimes mtm_2) \boxminus mtm_3)}$$

$$\frac{\vdash m_1 : mtm_1' \boxminus mtm_1 \qquad \vdash m_2 : mtm_2 \boxminus mtm_2'}{\vdash \mathtt{minterpose}(m_1, m_2) : (mtm_1 \boxminus mtm_2) \boxminus (mtm_1' \boxminus mtm_2')}$$

Figure 4.13: Higher-order macro-machines - typing



(a) $\texttt{mhoev}_{mtm_1,mtm_2}$
: $((mtm_1 \boxminus mtm_2) \boxtimes mtm_1) \boxminus mtm_2$

(b) $\texttt{mplug}(m_1, m_2)$
: mtm_2

(c) $m_2 \boxdot m_1$
: $mtm_1 \boxminus mtm_3$

(d) $m_1 \boxtimes m_2$
: $(mtm_1 \boxtimes mtm_2) \boxminus (mtm_1' \boxtimes mtm_2')$

(e) $\texttt{mhofb}^{mtm_3}_{mtm_1,mtm_2}(m)$
: $mtm_1 \boxminus mtm_2$

(f) $\texttt{minterpose}(m_1, m_2)$
: $(mtm_1 \boxminus mtm_2) \boxminus (mtm_1' \boxminus mtm_2')$

Figure 4.14: Higher-order macro-machines - selected graphical representations

Its dual, the $\boxed{\texttt{mhoneutrinv}_{mtm}}$ machine, is built in a similar manner.

$$\texttt{mhoneutrinv}_{mtm} \stackrel{\text{def}}{=} (\texttt{mid}_{\texttt{unit}} \parallel \texttt{mhoid}_{mtm}) \bullet \langle\, mtm^{+}, \texttt{unit}, mtm^{-} \mapsto 1,0,2 \,\rangle$$

- The *lifted constant machine* $\boxed{\texttt{mhoconst}^{n}_{m}}$ is a variant of the constant machine restricted to bounded natural numbers. Its type may seem strange at first, but we will later see that its shape makes it easy to combine with other higher-order machines.

$$\texttt{mhoconst}^{n}_{m} \stackrel{\text{def}}{=} \texttt{mneutrinv}_{\{m\}} \bullet \texttt{mwrap}_{\{m\}} \bullet \texttt{mconst}_{n} \bullet \texttt{mneutr}_{\texttt{unit}}$$

- The *lifted sum machine* $\boxed{\texttt{mhosum}^{n_2}_{n_1}}$ is a variant of the summation machine adapted to higher-order types. It can be obtained by adding wires of unit types as dictated by its type.

$$\texttt{mhosum}^{n_2}_{n_1} \stackrel{\text{def}}{=} (\texttt{mid}_{\texttt{unit}} \parallel \texttt{msum}^{n_2}_{n_1}) \bullet \texttt{mswap}_{\texttt{unit} \multimap \{n_1\}[n_2]}$$

- The *lifted stream merging machine* $\boxed{\texttt{mhomerge}_{mt}(m)}$ lifts the merging macro-machine defined in the previous section to the higher-order setting. As before, we add wires of unit types where needed.

$$\texttt{mhomerge}_{mt}(m) \stackrel{\text{def}}{=} \texttt{mid}_{\texttt{unit}} \parallel ((\texttt{msrcmerge}_{mty}(m) \parallel \texttt{mid}_{\texttt{unit}}) \bullet \texttt{mneutr}_{mt[1] \times \texttt{unit}} \bullet \texttt{mswap}_{\texttt{unit} \multimap mt[1] \times \texttt{unit}})$$

- The *lifted stream sampling machine* $\boxed{\texttt{mhowhen}_{mt}(m)}$ is similar to the previous one, except that it lifts sampling rather than merging.

$$\texttt{mhowhen}_{dt}(m) \stackrel{\text{def}}{=} \texttt{mid}_{\texttt{unit}} \parallel ((\texttt{mid}_{\texttt{unit}} \parallel \texttt{msrcwhen}_{(\!|dt|\!)}(m)) \bullet \texttt{mswap}_{\texttt{unit} \multimap (\!|dt|\!)[1]})$$

- The $\boxed{\texttt{mhoev}_{mtm_1,mtm_2}}$ machine is the *evaluation machine*. It is *internal* in the sense that it makes it possible to express the application of higher-order machines as a machine itself, and thus is the machine equivalent of $\lambda f.\lambda x.f(x)$. Combined with a machine of type $mtm_1 \boxminus mtm_2$ and a machine of type mtm_1, it produces a machine of type mtm_2. This leads to the permutation

$$\texttt{mhoev}_{mtm_1,mtm_2} \stackrel{\text{def}}{=} \langle\, mtm_1^{-}, mtm_2^{+}, mtm_1^{+}, mtm_2^{-} \mapsto 2,3,0,1 \,\rangle$$

represented in Figure 4.14 (a). Like the higher-order identity machine, the internal application machine just passes values along, orchestrating the communication between production and consumption of data.

- The *application combinator* $\boxed{\texttt{mplug}(m_1, m_2)}$, sometimes called the *plugging machine*, is related to the previous one. Its typing rule shows that the machine m_1 should be of type $mtm_1 \boxminus mtm_2$ and m_2 of type mtm_1. In contrast with the application machine, the

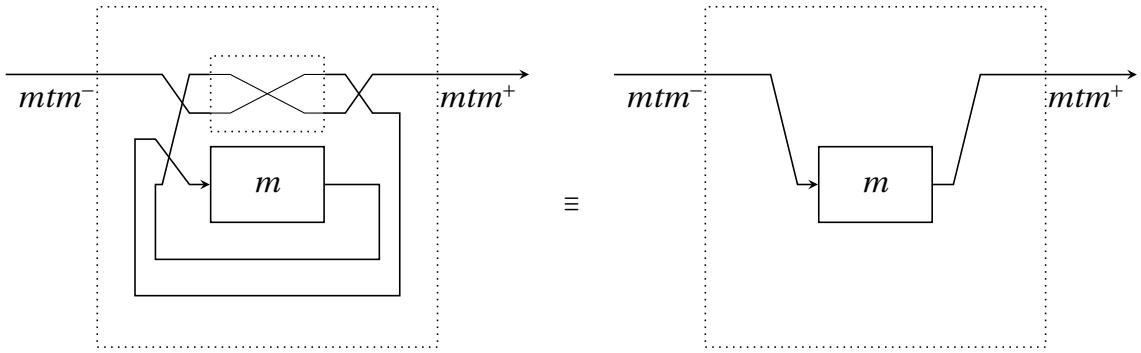


Figure 4.15: The machine $\texttt{mplug}(\texttt{mhoid}_{mtm}, m)$ is equivalent to m

plugging machine completely hides the type mtm_1 and only exposes mtm_2. However, actually applying m_1 to m_2 necessitates to connect a wire of type mtm_1^+ from the output of m_2 to the second input of m_1, and symmetrically for a wire of type mtm_1^-. These two wires should be invisible from the outside, which suggests using the only machine able to hide a value: the feedback machine. This leads to the following definition

$$\begin{aligned}\texttt{mplug}(m_1, m_2) \stackrel{\text{def}}{=}\ & \texttt{mfb}^{mtm_1^- \times mtm_1^+}_{mtm_2^-, mtm_2^+}(m) \\ & \text{where } m = \langle\, mtm_1^-, mtm_2^+, mtm_1^+ \mapsto 1,0,2 \,\rangle \bullet (m_1 \Vdash m_2) \\ & \qquad \bullet \langle\, mtm_2^-, mtm_1^-, mtm_1^+ \mapsto 1,2,0 \,\rangle\end{aligned}$$

depicted in Figure 4.14 (b). This definition shows how the feedback is applied to the wires used for communication between m_1 and m_2. The only wires still pending are those of type mtm_2^- and mtm_2^+ of m_1, which are exported for further use.

Example 22. This definition is consistent with the higher-order identity machine exposed before. Consider the right side of Figure 4.15, where one has applied the higher-order identity machine to an arbitrary machine $\vdash m : mtm$. By straightening the tangled wires, one obtains the left side, which is nothing else than m itself. Thus, informally, $\texttt{mhoev}_{\texttt{mhoid}_{mtm}, m}$ and m are equivalent.

- The $\boxed{\texttt{mhogath}^{mtm}_{n_1,n_2}(m_1, m_2)}$ and $\boxed{\texttt{mhoscatt}^{mtm}_{n_1,n_2}(m_1, m_2)}$ machines lift the base scattering and gathering machines to the higher-order macro-type system. We explain gathering, since scattering is perfectly symmetric.

  The definition of higher-order gathering obviously involves the first-order gathering machine. The fact that it also needs the (first-order) scattering machine may appear surprising at first. Completely expanding the type $mtm[n_1][n_2] \boxminus mtm[n_1 * n_2]$ gives

$$mtm^+[n_1][n_2] \times mtm^-[n_1 * n_2] \multimap mtm^-[n_1][n_2] \times mtm^+[n_1 * n_2]$$

which shows why scattering must be involved: we need to pass from type $mtm^-[n_1 * n_2]$ to type $mtm^-[n_1][n_2]$. This leads to the definition

$$\begin{aligned}\mathtt{mhogath}^{mtm}_{n_1,n_2}(m_1,m_2) \stackrel{\text{def}}{=}\ & (\mathtt{mscattct}^{mtm^-}_{n_1,n_2}(m_1,m_2) \parallel \mathtt{mgath}^{mtm^+}_{n_1,n_2}) \\ & \bullet \mathtt{mswap}_{mtm[n_1][n_2] \multimap mtm[n_1 * n_2]}\end{aligned}$$

which also shows that m_1 and m_2 are the machines producing the scattering factors. The reasoning is similar for scattering, for which the situation is simply reversed, with gathering in contravariant position.

$$\begin{aligned}\mathtt{mhoscatt}^{mtm}_{n_1,n_2}(m_1,m_2) \stackrel{\text{def}}{=}\ & (\mathtt{mgath}^{mtm^-}_{n_1,n_2} \parallel \mathtt{mscattct}^{mtm^+}_{n_1,n_2}(m_1,m_2)) \\ & \bullet \mathtt{mswap}_{mtm[n_1 * n_2] \multimap mtm[n_1][n_2]}\end{aligned}$$

- In the beginning of this section we introduced the wrapping macro-machine $\mathtt{mwrap}_{mt}$, a simply variant of stuttering where the input is repeated exactly once. This machine has an inverse in the form of $\mathtt{munwrap}_{mt}$ which removes the layer of indirection. The *lifted wrapping machine* $\boxed{\mathtt{mhowrap}_{mtm}}$ is the higher-order analogue. Its definition is

$$\mathtt{mhowrap}_{mtm} \stackrel{\text{def}}{=} (\mathtt{munwrap}_{mtm^-} \parallel \mathtt{mwrap}_{mtm^+}) \bullet \mathtt{mswap}_{mtm^+ \multimap mtm^-[1]}$$

and, as in the case of gathering/scattering, we must perform both wrapping of mtm^+ and unwrapping of mtm^-.

As an aside, observe that this definition shows that in contrast with wrapping, stuttering cannot be lifted to higher-order machines in a natural way. Indeed, this would require a machine able to undo the action of the stuttering machine in the general case, transforming values of type $mt[n]$ into values of type mt, which is only possible when the lists are always of size one. This limitation highlights the *linear* nature of the machine language: one cannot create a replication machine that is both generic and *internal*, in the sense that it does not access the body of the machine to be replicated. The best we can have is the *lifted unwrapping machine* $\boxed{\mathtt{mhounwrap}_{mtm}}$.

$$\mathtt{mhounwrap}_{mtm} \stackrel{\text{def}}{=} (\mathtt{mwrap}_{mtm^-} \parallel \mathtt{munwrap}_{mtm^+}) \bullet \mathtt{mswap}_{mtm^- \multimap mtm^+[1]}$$

- The machines $\boxed{\mathtt{mhozip}^n_{mtm_1,mtm_2}}$ and $\boxed{\mathtt{mhounzip}^n_{mtm_1,mtm_2}}$ are respectively the *lifted zipping* and *lifted unzipping* machines. Their definition is similar to the case of gathering/scattering, with higher-order zipping involving higher-order unzipping and conversely. Both machines are perfectly symmetric.

$$\begin{aligned}\mathtt{mhozip}^n_{mtm_1,mtm_2} \stackrel{\text{def}}{=}\ & (\mathtt{munzip}^n_{mtm_1^-,mtm_2^-} \parallel \mathtt{mzip}^n_{mtm_1^+,mtm_2^+}) \\ & \bullet \mathtt{mswap}_{(mtm_1^- \times mtm_2^-)[n] \multimap mtm_1^+[n] \times mtm_2^+[n]} \\ \mathtt{mhounzip}^n_{mtm_1,mtm_2} \stackrel{\text{def}}{=}\ & (\mathtt{mzip}^n_{mtm_1^-,mtm_2^-} \parallel \mathtt{munzip}^n_{mtm_1^+,mtm_2^+}) \\ & \bullet \mathtt{mswap}_{mtm_1^-[n] \times mtm_2^-[n] \multimap (mtm_1^+ \times mtm_2^+)[n]}\end{aligned}$$

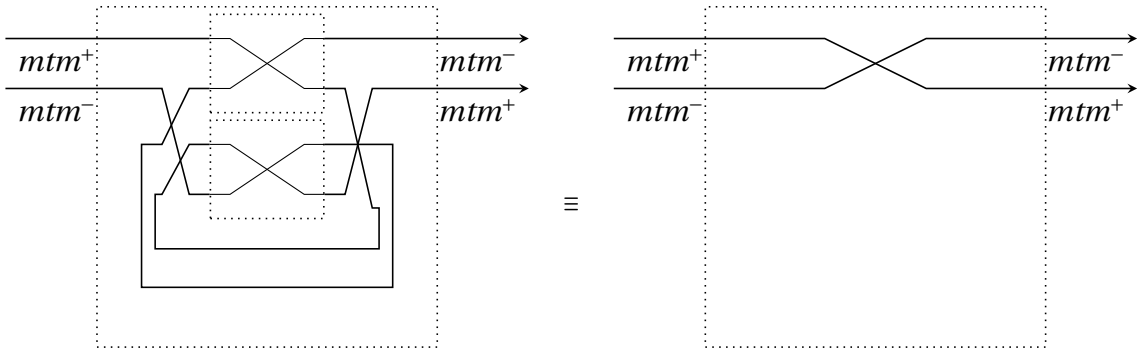


Figure 4.16: The machine $(\texttt{mhoid}_{mtm} \boxdot \texttt{mhoid}_{mtm})$ is equivalent to $\texttt{mhoid}_{mtm} : mtm \boxminus mtm$

- The *lifted bounded integer coercion* machine $\boxed{\texttt{mhoncoer}_{n_1,n_2}}$ mimics the first-order one.

$$\texttt{mhoncoer}_{n_1,n_2} \stackrel{\text{def}}{=} (\texttt{mid}_{\texttt{unit}} \Vdash \texttt{mncoer}_{n_1,n_2}) \bullet \texttt{mswap}_{\{n_2\}^- \multimap \{n_1\}^+}$$

- Similarly for the *lifted list coercion* machine $\boxed{\texttt{mholcoer}^{mtm}_{n_1,n_2}}$.

$$\texttt{mholcoer}^{mtm}_{n_1,n_2} \stackrel{\text{def}}{=} (\texttt{mlcoer}^{mtm^-}_{n_2,n_1} \Vdash \texttt{mlcoer}^{mtm^+}_{n_1,n_2}) \bullet \texttt{mswap}_{mtm^-[n_2] \multimap mtm^+[n_1]}$$

  Remark that contravariance forces us to coerce not only from n_1 to n_2 but also from n_2 to n_1 as well. Thus, this machines always drops suffixes of exactly one of its inputs, except in the trivial case when n_1 and n_2 are equal.

- The *higher-order composition* machine $\boxed{m_1 \boxdot m_2}$ has a straightforward typing rule, and makes it possible to compose higher-order machine types in a convenient fashion. It is actually the more general case of the plugging machine $\texttt{mplug}(m_1, m_2)$, in the case where some wires of m_2 have to be exposed to the external world.

$$\begin{aligned} m_2 \boxdot m_1 \stackrel{\text{def}}{=}\ & \texttt{mfb}^{mtm_2^- \times mtm_2^+}_{mtm_1^+ \times mtm_3^-, mtm_1^- \times mtm_3^+}(m) \\ & \text{where } m = \langle mtm_1^-, mtm_2^+, mtm_2^-, mtm_3^+ \mapsto 0,3,2,1 \rangle \bullet (m_1 \Vdash m_2) \\ & \qquad \bullet \langle mtm_1^+, mtm_3^-, mtm_2^-, mtm_2^+ \mapsto 0,2,1,3 \rangle \end{aligned}$$

  Figure 4.14 (c) gives a graphical representation of this machine, showing how it is actually a symmetrized version of Figure 4.14 (b). The feedback loop on $mtm_2^- \times mtm_2^+$ is represented as two feedback loops, one on each component of the pair.

  Example 23. The right side of Figure 4.16 shows the diagram obtained by composing two higher-order identity machines. By straightening the wires one gets the diagram on the left, which is nothing else than the identity machine itself: $id \circ id = id$.

- The *higher-order pairing* (or *parallel composition*) operator $\boxed{m_1 \boxtimes m_2}$ composes horizontally m_1 and m_2. The resulting machine transforms pairs of machines into pairs of machines, as shown by its typing rule. It should not be confused to the first-order product $m_1 \shortparallel m_2$, which as a distinct type shown on the right below.

$$(mtm_1 \boxtimes mtm_2) \boxminus (mtm'_1 \boxtimes mtm'_2) \quad \neq \quad (mtm_1 \boxminus mtm'_1) \boxtimes (mtm_1 \boxminus mtm'_2)$$

  Actually, the definition of $m_1 \boxtimes m_2$ combines the first-order product with the proper permutations in order to transform the (incorrect) type into the expected one.

$$\begin{aligned} m_1 \boxtimes m_2 \stackrel{\text{def}}{=} & \left\langle mtm_1^-, mtm_1'^+, mtm_2^-, mtm_2'^+ \mapsto 0,2,1,3 \right\rangle \bullet (m_1 \shortparallel m_2) \\ & \bullet \left\langle mtm_1^+, mtm_2^+, mtm_1'^-, mtm_2'^- \mapsto 0,2,1,3 \right\rangle \end{aligned}$$

  The graphical representation is given in Figure 4.14 (d).

- The *higher-order feedback* combinator $\boxed{\texttt{mhofb}^{mtm_3}_{mtm_1,mtm_2}(m)}$ is built from its first-order counterpart and permutations. The definition below is depicted in Figure 4.14 (e).

$$\begin{aligned} \texttt{mhofb}^{mtm_3}_{mtm_1,mtm_2}(m) \stackrel{\text{def}}{=} & \ \texttt{mfb}^{mtm_3^- \times mtm_3^+}_{mtm_1^+ \times mtm_2^-, mtm_1^- \times mtm_2^+}(m') \\ & \text{where } m' = \left\langle mtm_1^-, mtm_3^-, mtm_2^+, mtm_3^+ \mapsto 0,2,1,3 \right\rangle \boxdot m \\ & \qquad \boxdot \left\langle mtm_1^+, mtm_2^-, mtm_3^-, mtm_3^+ \mapsto 0,3,1,2 \right\rangle \end{aligned}$$

- The *higher-order replication* combinator $\boxed{\texttt{mhorepl}^n_{mtm_1,mtm_2}(m)}$ replicates m, a machine of type $mtm_1 \boxminus mtm_2$, according to the replication factors received on its first input. This factor is bounded by n. The resulting machine inhabits type $mtm_1[n] \boxminus mtm_2[n]$, which is different from $mtm_1 \boxminus mtm_2[n]$.

$$\begin{aligned} \texttt{mhorepl}^n_{mtm_1,mtm_2}(m) \stackrel{\text{def}}{=} & \texttt{munzip}^n_{mtm_1^-,mtm_2^+} \bullet \texttt{mrepl}_n(m) \bullet (\texttt{mid}_{\{n\}} \shortparallel \texttt{mzip}^n_{mtm_1^+,mtm_2^-}) \\ & \text{where } m = \texttt{mdrivect}^n_{mtm_1 \boxminus mtm_2}(m_1, m_2) \end{aligned}$$

  The definition uses the unzipping machine to obtain the expected type.

- The next two machines implement *currying* and *uncurrying*. Note that the macro-types $(mtm_1 \boxtimes mtm_2) \boxminus mtm_3$ and $mtm_1 \boxminus (mtm_2 \boxminus mtm_3)$ appearing in the rules of the machines $\boxed{\texttt{mcurry}^{mtm_3}_{mtm_1,mtm_2}}$ and $\boxed{\texttt{muncurry}^{mtm_3}_{mtm_1,mtm_2}}$ both unfold to

$$mtm_1^+ \times mtm_2^+ \times mtm_3^- \multimap mtm_1^- \times mtm_2^- \times mtm_3^+$$

  and thus are actually equal. This suggests that we already known how to implement these two machines: both must transform the type $(mtm_1 \boxtimes mtm_2) \boxminus mtm_3 = mtm_1 \boxminus (mtm_2 \boxminus mtm_3)$ into itself, and are thus simply the (higher-order) identity machine.

$$\texttt{mcurry}^{mtm_3}_{mtm_1,mtm_2} \stackrel{\text{def}}{=} \texttt{muncurry}^{mtm_3}_{mtm_1,mtm_2} \stackrel{\text{def}}{=} \texttt{mhoid}_{(mtm_1 \boxtimes mtm_2) \boxminus mtm_3}$$

  The triviality of their definitions implies that we could forgo the above machines and directly use the identity machine in their place. We still prefer dedicated (un)currying machines since they make things more readable.

- The *interposition* machine $\boxed{\texttt{minterpose}(m_1, m_2)}$ will prove to be convenient during the translation. It receives a machine of type $mtm_1 \boxminus mtm_2$ and transforms it into a machine of type $mtm_1' \boxminus mtm_2'$, assuming m_1 is of type $mtm_1' \boxminus mtm_1$ and m_2 is of type $mtm_2 \boxminus mtm_2'$. Acute readers may already guess that it can help with the implementation of subtyping-like rules for arrows, m_1 and m_2 respectively implementing the transformations in contravariant and covariant position. Its definition follows.

$$\begin{array}{rcl} \texttt{minterpose}(m_1, m_2) & = & \langle\, mtm_1'^{-}, mtm_1^{+}, mtm_2^{-}, mtm_2'^{+} \mapsto 1,2,0,3 \,\rangle \bullet (m_1 \Vdash m_2) \\ & & \bullet \langle\, mtm_1^{-}, mtm_2^{+}, mtm_1'^{+}, mtm_2'^{-} \mapsto 2,0,1,3 \,\rangle \end{array}$$

  This definition is represented in Figure 4.14 (f).

- The *higher-order permutation* macro-machine $\boxed{\langle\, mtm_1, \ldots, mtm_n \mapsto i_1, \ldots, i_n \,\rangle_{ho}}$ has n inputs and outputs. Its j-th output is equal to its i_j-th input.

$$\vdash \langle\, mtm_1, \ldots, mtm_n \mapsto i_1, \ldots, i_n \,\rangle_{ho} : mtm_1 \boxtimes \ldots \boxtimes mtm_n \boxminus mtm_{i_1} \boxtimes \ldots \boxtimes mtm_{i_n}$$

  Higher-order permutations reduce to first-order ones.

$$\begin{array}{rcl} & & \langle\, mtm_1, \ldots, mtm_n \mapsto i_1, \ldots, i_n \,\rangle_{ho} \\ & = & \langle\, mtm_1^{+}, \ldots, mtm_n^{+}, mtm_1^{-}, \ldots, mtm_n^{-} \mapsto i_1 + n, \ldots, i_n + n, i_1, \ldots, i_n \,\rangle \end{array}$$

4.4 The Translation

The translation from the source to the target language combines all the elements from Chapter 3 and Chapter 4. It transforms the typing derivations of Chapter 3 into well-typed machines. Its goal is two-fold. First, it reduces all source-level concepts which involve conceptually infinite objects, such as streams, to finite objects such as lists and bounded integers. In particular, clock types are translated to actual machines generating the successive values of the corresponding clock. Second, it translates away linear higher-order functions to first-order ones composed using well-behaved feedback loops.

The translation is organized as a family of type-directed functions. Each source-level type is translated to a target-level machine type, and the translation of programs respects the translation of types. In other words, writing $\mathcal{M}_{mtm}$ for the set of machines m such that $\vdash m : mtm$ and $(\!|\_|\!)$ for the translation function, we have $(\!|\Box \vdash e : t|\!) \in \mathcal{M}_{(\!|t|\!)}$. We begin with the translation of types. The translation of adaptability, gathering and scattering derivations is then described; they involve the less obvious combinations of machines. Finally, the compilation of typing derivations mostly consists in plugging together all the pieces described in this chapter in a straightforward manner.

4.4.1 Translating Types

The goal of the translation process is to produce state machines using a statically-bounded amount of memory. We have seen that these bounds appear in the types of the target language,

either as upper bounds on the possible value of numbers, or as maximum list lengths. Thus, of the main role of the translation of types is to exhibit these bounds. Where do they come from? First, we have seen that by definition the integer scalars of the source language are actually 64 bit unsigned integers. Second, each clock type actually denotes a clock with a statically-computable bound. Let us discuss this point first.

Clock bounds In order to describe the translation from clocked streams to machines computing finite amount of data per reaction, we need to define a function $\lceil ct \rceil$ computing an upper bound of the value of integers in the clock denoted by ct. One possibility would be to compute its normal form $nf(ct)$, which is an ultimately periodic word, and then traverse the word to find its largest number, $\lceil nf(ct) \rceil$. This approach gives an exact upper bound, but does not extend to languages with more expressive clock types. We adopt the more flexible definition given below.

$$\begin{array}{lcl} \lceil p \rceil & = & \lceil p \rceil \\ \lceil ct_1 \ \mathbf{on}\ ct_2 \rceil & = & \lceil ct \rceil * \lceil ct' \rceil \end{array}$$

This function is defined by structural recursion, and as such one does not need to reduce ct to its normal form. This will turn out to be important since it extends gracefully to more complex clock type languages given in Chapter 5, where $nf(ct)$ does not exist.

Remark 19. The price to pay for this additional flexibility is imprecision: we have $\lceil nf(ct) \rceil \leq \lceil ct \rceil$ in general, and the bound is not tight. The example below shows one of the simplest example where the loss of precision occurs.

$$\lceil nf((2)\ \mathbf{on}\ (1\ 0)) \rceil = \lceil (1) \rceil = 1 \quad < \quad \lceil (2)\ \mathbf{on}\ (1\ 0) \rceil = \lceil (2) \rceil * \lceil (1\ 0) \rceil = 2$$

This imprecision actually impacts the translation process by forcing us to insert additional coercions in some places. We discuss this issue after having explained the translation of types.

Data types and types The type system proposed in Chapter 3 assumes very little of the precise grammar of data types dt. In this chapter, we require that data types are only inhabited by finite set of scalars. Recall that this is actually the case for both **bool** and **int**, the latter being the type of unsigned 64 bits integers. We can thus write a translation function from a data type dt to a value type mt, given by the following clauses.

$$\begin{array}{lcl} (\!|\mathbf{bool}|\!) & = & \{1\} \\ (\!|\mathbf{int}|\!) & = & \{2^{64}-1\} \end{array}$$

The next step is to map each source type t to a machine type mtm. We have all the ingredients in hands: products correspond to higher machine products, arrows to higher machine arrows, and streams $dt :: ct$ correspond to machine with no inputs and one output which is a list of values in $(\!|dt|\!)$. The maximum size of this list is given by the maximum integer present in the clocked denoted by ct, which we abbreviate as $\lceil ct \rceil$, with $\lceil ct_1\ \mathbf{on}\ ct_2 \rceil = \lceil ct_1 \rceil * \lceil ct_2 \rceil$.

$$\begin{array}{lcl} (\!|dt :: ct|\!) & = & \mathtt{unit} \multimap (\!|dt|\!)[\lceil ct \rceil] \\ (\!|t_1 \otimes t_2|\!) & = & (\!|t_1|\!) \boxtimes (\!|t_2|\!) \\ (\!|t_1 \multimap t_2|\!) & = & (\!|t_1|\!) \boxminus (\!|t_2|\!) \end{array}$$

In order to deal with open terms, the translation has to be extended to contexts. As usual, they are interpreted as potentially large products.

$$\begin{array}{lcl} (\!|\Box|\!) & = & \texttt{munit} \\ (\!|\Gamma, x : t|\!) & = & (\!|\Gamma|\!) \boxtimes (\!|t|\!) \end{array}$$

Clock types Clock types are not only annotations: they cannot be erased as they drive the execution of some language constructs, as we have seen in the synchronous semantics. Thus, we need to translate each clock type ct to a machine producing the successive values of the clock it denotes. This machine has no input and produces a number bounded by $\lceil ct \rceil$ at each reaction.

$$\begin{array}{lcl} (\!|ct|\!) & \in & \mathcal{M}_{\texttt{mint}\{\lceil ct \rceil\}} \\ (\!|p|\!) & = & \texttt{mpw}_p(0) \\ (\!|ct_1 \textbf{ on } ct_2|\!) & = & \texttt{msum}_{\lceil ct_1 \rceil}^{\lceil ct_2 \rceil} \bullet \texttt{mdrivenict}_{\lceil ct_2 \rceil}^{\lceil ct_1 \rceil}((\!|ct_1|\!), (\!|ct_2|\!)) \end{array}$$

Ultimately periodic words are translated using the dedicated machines, and clock composition uses the summation operator. Note that, once again, this process is syntax-directed: one does not need to reduce the whole clock type to an ultimately periodic word $nf(ct)$ in order to implement it.

On coercions We can now explain concretely why the imprecision of the $\lceil \_ \rceil$ function sometimes forces us to insert coercions. This comes from the fact that while the source type system handles clock types up to equivalence, equivalent clock types may get translated to distinct target types. For instance, we know that (2) **on** $(1\,0) \equiv (1)$, yet we have

$$\begin{array}{lcl} & (\!|\textbf{bool} :: (2) \textbf{ on } (1\,0)|\!) = \texttt{unit} \to \{1\}[\lceil (2) \rceil \times \lceil (1\,0) \rceil] = \texttt{unit} \to \{1\}[2] \\ \neq & (\!|\textbf{bool} :: (1)|\!) = \texttt{unit} \to \{1\}[1] \end{array}$$

from the previous definitions. The point is that even if the machines obtained from well-typed programs of clock type (2) **on** $(1\,0)$ compute lists of length one at each reaction, this is not apparent in the translated type. The actual problem arises when one needs to move between $mtm[\lceil (2) \textbf{ on } (1\,0) \rceil]$ and $mtm[\lceil (1) \rceil]$. The solution is to use a higher-order list coercion machine $\texttt{mholcoer}^{mtm}_{\lceil (2) \textbf{ on } (1\,0) \rceil, \lceil (1) \rceil}$.

4.4.2 Translating Auxiliary Judgments

Value The role of the value judgments is to classify whether the inhabitants of a type or a context is duplicable and erasable. In the synchronous semantics this was not important, as elements of a domain are mathematical objects that can always be erased or duplicated at will. In the physical world of machines, this is no longer the case: one cannot duplicate or erase higher-order machines. We thus need to interpret a typing derivation of $\vdash t$ value either as a machine $(\!|\vdash t \text{ value}|\!)_D$ able to duplicate $(\!|t|\!)$, or as a machine $(\!|\vdash t \text{ value}|\!)_E$ able to erase it. Figure 4.17 gives the corresponding compilation functions.

$$
\begin{array}{lcl}
(\!|\vdash t \text{ value}|\!)_D & \in & \mathcal{M}_{(\!|t|\!)_D \boxminus (\!|t|\!)_D \boxtimes (\!|t|\!)_D} \\
(\!|\vdash dt :: ct \text{ value}|\!)_D & = & (\mathtt{mdup}_{(\!|dt::ct|\!)} \parallel \mathtt{mid}_{\mathtt{unit}}) \bullet (\mathtt{mid}_{(\!|dt::ct|\!)} \parallel \mathtt{mneutr}_{\mathtt{unit}}) \\
(\!|\vdash t_1 \otimes t_2 \text{ value}|\!)_D & = & \langle (\!|t_1|\!), (\!|t_1|\!), (\!|t_2|\!), (\!|t_2|\!) \mapsto 0,2,1,3 \rangle_{ho} \boxdot ((\!|\vdash t_1 \text{ value}|\!)_D \boxtimes (\!|\vdash t_2 \text{ value}|\!)_D)
\end{array}
$$

$$
\begin{array}{lcl}
(\!|\vdash t \text{ value}|\!)_E & \in & \mathcal{M}_{(\!|t|\!)^*} \\
(\!|\vdash dt :: ct \text{ value}|\!)_E & = & \mathtt{mforg}_{(\!|dt::ct|\!)} \\
(\!|\vdash t_1 \otimes t_2 \text{ value}|\!)_E & = & (\!|\vdash t_1 \text{ value}|\!)_E \parallel (\!|\vdash t_2 \text{ value}|\!)_E
\end{array}
$$

$$
\begin{array}{lcl}
(\!|\vdash \Gamma \text{ value}|\!)_E & \in & \mathcal{M}_{(\!|\Gamma|\!)^*} \\
(\!|\vdash \square \text{ value}|\!)_E & = & \mathtt{mid}_{(\!|\square|\!)} \\
(\!|\vdash \Gamma, x : t \text{ value}|\!)_E & = & (\!|\Gamma|\!)_E \parallel (\!|t|\!)_E
\end{array}
$$

Figure 4.17: Compilation - value judgment

$$
\begin{array}{lcl}
(\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!) & \in & \mathcal{M}_{(\!|\Gamma|\!) \boxminus (\!|\Gamma_1|\!) \boxtimes (\!|\Gamma_2|\!)} \\
(\!|\square \vdash \square \otimes \square|\!) & = & \mathtt{mid}_{(\!|\square|\!) \boxminus (\!|\square|\!) \boxtimes (\!|\square|\!))^-} \\
(\!|\Gamma, x : t \vdash \Gamma_1, x : t \otimes \Gamma_2, x : t|\!) & = & \langle (\!|\Gamma_1|\!), (\!|\Gamma_2|\!), (\!|t|\!), (\!|t|\!) \mapsto 0,2,1,3 \rangle_{ho} \boxdot ((\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!) \boxtimes (\!|\vdash t \text{ value}|\!)_D) \\
(\!|\Gamma, x : t \vdash \Gamma_1, x : t \otimes \Gamma_2|\!) & = & \langle (\!|\Gamma_1|\!), (\!|\Gamma_2|\!), (\!|t|\!), \mapsto 0,2,1 \rangle_{ho} \boxdot ((\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!) \boxtimes \mathtt{mhoid}_{(\!|t|\!)}) \\
(\!|\Gamma, x : t \vdash \Gamma_1 \otimes \Gamma_2, x : t|\!) & = & (\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!) \boxtimes \mathtt{mhoid}_{(\!|t|\!)}
\end{array}
$$

Figure 4.18: Compilation - context splitting judgment

The duplication machine $(\!|\vdash t \text{ value}|\!)_D$ has type $(\!|t|\!) \boxminus ((\!|t|\!) \boxtimes (\!|t|\!))$. Since stream types are implemented by scalar values, the first case simply consists in using the primitive value duplication machine. The second case uses the duplicating machines obtained recursively from premises $\vdash t_1$ value and $\vdash t_2$ value, rearranging their outputs to have the expected type. Note that the definition is relatively concise thanks to higher composition machines and the higher permutation.

The type of the erasure machine $(\!|\vdash t \text{ value}|\!)_E$ is $\vdash t \text{ value}^*$. Its type is $(\!|t|\!)^*$, expressing that it is able to completely consume a machine of type $(\!|t|\!)$. It is also found in Figure 4.17. The base case of streams is handled using the primitive value erasure machine. The case of products is handled by juxtaposing the sub-machines $(\!|\vdash t_1 \text{ value}|\!)_E$ and $(\!|\vdash t_2 \text{ value}|\!)_E$. This is well-typed since $mtm_1^* \boxtimes mtm_2^* = (mtm_1 \boxtimes mtm_2)^*$.

The context erasure machine $(\!|\Gamma|\!)_E$ is built using vertical composition.

Separation The separation judgment $\Gamma \vdash \Gamma_1 \otimes \Gamma_2$ translates into a higher-order machine transforming inputs in $(\!|\Gamma|\!)$ into outputs in $(\!|\Gamma_1|\!) \boxtimes (\!|\Gamma_2|\!)$. The interpretation is given in Figure 4.18. It is close to the one given in the synchronous semantics, with the pairing and composition

$$
\begin{array}{lcl}
(\!|\vdash t <:_k t'|\!) & \in & \mathcal{M}_{(\!|t|\!) \boxminus (\!|t'|\!)} \\
(\!|\vdash dt :: ct_1 <:_k dt :: ct_2|\!) & = & (\mathtt{mbuffct}^{(\!|dt|\!)}_{\lceil ct_1 \rceil, \lceil ct_2 \rceil, size(ct_1, ct_2)}((\!|ct_2|\!)) \parallel \mathtt{mid}_{\mathtt{unit}}) \bullet \mathtt{mswap}_{(\!|dt|\!)[[\lceil ct_1 \rceil]] \boxminus \mathtt{unit}} \\
(\!|\vdash t_1 \otimes t_2 <:_k t_1' \otimes t_2'|\!) & = & (\!|\vdash t_1 <:_k t_1'|\!) \boxtimes (\!|\vdash t_2 <:_k t_2'|\!) \\
(\!|\vdash t_1 \multimap t_2 <:_k t_1' \multimap t_2'|\!) & = & \mathtt{minterpose}((\!|\vdash t_1' <:_0 t_1|\!), (\!|\vdash t_2 <:_k t_2'|\!))
\end{array}
$$

Figure 4.19: Compilation - adaptability judgment

operators from domain theory replaced with those of machines. The compilation of rule SEP-CONTRACT relies on the duplication machine $(\!|\vdash t \text{ value}|\!)_D$. Most cases simply shuffle machine types around using permutations in order to obtain the expected output type.

Remark 20. Note that in several cases, especially the last one, the generated machines are well-typed only up to associativity of $\boxtimes$. We feel that this small sacrifice in rigor gives a large pay-off in terms of readability.

Adaptability The adaptability judgment $\vdash t <:_k t'$ is translated to a machine transforming $(\!|t|\!)$ into $(\!|t'|\!)$. The interpretation is given in Figure 4.19. As expected, it relies on first-order buffers to implement stream buffering. The function $size(ct_1, ct_2)$ computes the capacity needed for a buffer whose input clock is denoted by ct_1 and output clock by ct_2. This is the maximum of the cumulative functions of $nf(ct_1)$ and $nf(ct_2)$, as described by the formulas below.

$$
\begin{array}{lcl}
size(w_1, w_2) & = & max_{i \geq 0}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)) \\
size(ct_1, ct_2) & = & size(nf(w_1), nf(w_2))
\end{array}
$$

Notice how the $\mathtt{minterpose}((\!|\vdash t_1' <:_0 t_1|\!), (\!|\vdash t_2 <:_k t_2'|\!))$ machine is used to compile the function adaptability rule, as we announced in the previous section.

Gathering Remember that the judgment $\vdash t \uparrow_{ct} t'$ expresses that one can transform a inhabitants of t, computed at the rate described by ct, into inhabitants of t'. Thus, the corresponding machine should consume several inhabitants of $(\!|t|\!)$ and produce one inhabitant of $(\!|t'|\!)$. The exact amount of inhabitants of $(\!|t|\!)$ needed at a given reaction is determined by the corresponding integer in the clock denoted by ct; thus it is at most $\lceil ct \rceil$. Hence, a gathering judgment should be translated to a machine of type $(\!|t|\!)[[\lceil ct \rceil]] \boxminus (\!|t'|\!)$. The definition is given in the middle part of Figure 4.20.

- For rule UPSTREAM, we use the higher-order gathering machine. A coercion is needed on the output to pass from lists of size $\lceil ct \textbf{ on } ct_1 \rceil$ to lists of size $\lceil ct_2 \rceil$. We know that this coercion is benign since typing ensures that $ct_2 \equiv ct \textbf{ on } ct_1$ holds.
- For rule UPPROD, we pair the compilation of the two premises. The unziping machine is needed to convert the input from $(mtm_1 \boxtimes mtm_2)[\lceil ct \rceil]$ to $mtm_1[\lceil ct \rceil] \boxtimes mtm_2[\lceil ct \rceil]$, which is the type expected by the pair of machines.

$$
\begin{array}{lcl}
(\!|\vdash t \uparrow_{ct} t'|\!) & \in & \mathcal{M}_{(\!|t|\!)[\lceil ct\rceil]\boxminus(\!|t'|\!)} \\
(\!|\vdash dt :: ct_1 \uparrow_{ct} dt :: ct_2|\!) & = & \mathtt{mholcoer}^{\mathtt{unit}\multimap(\!|dt|\!)}_{\lceil ct\ \mathbf{on}\ ct_2\rceil,\lceil ct_1\rceil} \boxdot \mathtt{mhogath}^{\mathtt{unit}\multimap(\!|dt|\!)}_{\lceil ct\rceil,\lceil ct_2\rceil}((\!|ct|\!),(\!|ct_2|\!)) \\
(\!|\vdash t_1 \otimes t_2 \uparrow_{ct} t_1' \otimes t_2'|\!) & = & ((\!|\vdash t_1 \uparrow_{ct} t_1'|\!) \boxtimes (\!|\vdash t_2 \uparrow_{ct} t_2'|\!)) \boxdot \mathtt{mhounzip}^{\lceil ct\rceil}_{(\!|t_1|\!),(\!|t_2|\!)} \\
(\!|\vdash t_1 \multimap t_2 \uparrow_{ct} t_1' \multimap t_2'|\!) & = & (\mathtt{minterpose}((\!|\vdash t_1' \downarrow_{ct} t_1|\!),(\!|\vdash t_2 \uparrow_{ct} t_2'|\!))) \\
 & & \boxdot \mathtt{mhounzip}^{\lceil ct\rceil}_{(\!|t_1|\!)^*,(\!|t_2|\!)} \\
(\!|\vdash t \uparrow_{ct\ \mathbf{on}\ ct'} t'|\!) & = & (\!|\vdash t'' \uparrow_{ct} t'|\!) \boxdot \mathtt{mplug}(\mathtt{mhorepl}^{\lceil ct\rceil}_{(\!|t|\!)[\lceil ct'\rceil],(\!|t''|\!)}((\!|\vdash t \uparrow_{ct'} t''|\!)),(\!|ct|\!)) \\
 & & \boxdot \mathtt{mhoscatt}^{(\!|t|\!)}_{\lceil ct\rceil,\lceil ct'\rceil}((\!|ct|\!),(\!|ct'|\!)) \\
(\!|\vdash t \uparrow_{ct} t'|\!) & = & \mathtt{mhounwrap}_{(\!|t'|\!)} \boxdot \mathtt{mholcoer}^{(\!|t'|\!)}_{\lceil ct'\ \mathbf{on}\ ct\rceil,1} \\
 & & \boxdot \mathtt{mhogath}^{(\!|t'|\!)}_{\lceil ct'\rceil,\lceil ct\rceil}((\!|ct'|\!),(\!|ct|\!)) \\
 & & \boxdot \mathtt{mplug}(\mathtt{mhorepl}^{\lceil ct\rceil}_{(\!|t|\!),(\!|t'|\!)[\lceil ct'\rceil]}((\!|\vdash t \downarrow_{ct'} t'|\!)),(\!|ct|\!)) \\
 & & \\
(\!|\vdash t \downarrow_{ct} t'|\!) & \in & \mathcal{M}_{(\!|t|\!)\boxminus(\!|t'|\!)[\lceil ct\rceil]} \\
(\!|\vdash dt :: ct_1 \downarrow_{ct} dt :: ct_2|\!) & = & \mathtt{mhoscatt}^{\mathtt{unit}\multimap(\!|dt|\!)}_{\lceil ct_1\rceil,\lceil ct\rceil}((\!|ct|\!),(\!|ct_1|\!)) \boxdot \mathtt{mholcoer}^{\mathtt{unit}\multimap(\!|dt|\!)}_{\lceil ct\ \mathbf{on}\ ct_1\rceil,\lceil ct_2\rceil} \\
(\!|\vdash t_1 \otimes t_2 \downarrow_{ct} t_1' \otimes t_2'|\!) & = & \mathtt{mhozip}^{\lceil ct\rceil}_{(\!|t_1|\!),(\!|t_2|\!)} \boxdot ((\!|\vdash t_1 \downarrow_{\lceil ct\rceil} t_1'|\!) \boxtimes (\!|\vdash t_2 \downarrow_{\lceil ct\rceil} t_2'|\!)) \\
(\!|\vdash t_1 \multimap t_2 \downarrow_{ct} t_1' \multimap t_2'|\!) & = & \mathtt{mhozip}^{\lceil ct\rceil}_{(\!|t_1'|\!)^*,(\!|t_2'|\!)} \boxdot (\mathtt{minterpose}((\!|\vdash t_1' \downarrow_{ct} t_1|\!),(\!|\vdash t_2 \downarrow_{ct} t_2'|\!))) \\
(\!|\vdash t \downarrow_{ct\ \mathbf{on}\ ct'} t'|\!) & = & \mathtt{mhogath}^{(\!|t|\!)}_{\lceil ct\rceil,\lceil ct'\rceil}((\!|ct|\!),(\!|ct'|\!)) \boxdot m \boxdot (\!|\vdash t \downarrow_{ct} t''|\!) \\
 & & \text{where } m = \mathtt{mplug}(\mathtt{mhorepl}^{\lceil ct\rceil}_{(\!|t''|\!),(\!|t'|\!)[\lceil ct'\rceil]}((\!|\vdash t'' \downarrow_{ct'} t'|\!)),(\!|ct|\!)) \\
(\!|\vdash t \downarrow_{ct} t'|\!) & = & \mathtt{mplug}(\mathtt{mhorepl}^{\lceil ct\rceil}_{(\!|t|\!)[\lceil ct'\rceil],(\!|t'|\!)}((\!|\vdash t \uparrow_{ct'} t'|\!)),(\!|ct|\!)) \\
 & & \boxdot \mathtt{mhoscatt}^{(\!|t|\!)}_{\lceil ct'\rceil,\lceil ct\rceil}((\!|ct'|\!),(\!|ct|\!)) \\
 & & \boxdot \mathtt{mholcoer}^{(\!|t|\!)}_{1,\lceil ct'\ \mathbf{on}\ ct\rceil} \boxdot \mathtt{mhowrap}_{(\!|t|\!)} \\
 & & \\
(\!|\vdash \Gamma \downarrow_{ct} \Gamma'|\!) & \in & \mathcal{M}_{(\!|\Gamma|\!)\boxminus(\!|\Gamma'|\!)[\lceil ct\rceil]} \\
(\!|\vdash \square \downarrow_{ct} \square|\!) & = & \mathtt{mstutterct}^{\lceil ct\rceil}_{(\!|\square|\!)^+}((\!|ct|\!)) \\
(\!|\vdash \Gamma, x : t \downarrow_{ct} \Gamma', x : t'|\!) & = & (\!|\vdash \Gamma \downarrow_{ct} \Gamma'|\!) \boxtimes (\!|\vdash t \downarrow_{ct} t'|\!)
\end{array}
$$

Figure 4.20: Compilation - gathering and scattering judgments

- Similarly, for rule UPARROW, we pass the unzipped input to the interposition of the machines obtained by composed the premises.
- For rule UPON, we need to compose the machine $\vdash t \uparrow_{ct'} t''$ and $\vdash t'' \uparrow_{ct} t'$ in order to transform inhabitants of the input type $(\!|t|\!)[\lceil ct \rceil * \lceil ct' \rceil]$ into inhabitants of $(\!|t'|\!)$. The solution is to replicate the machine $(\!|\vdash t \uparrow_{ct'} t''|\!)$ by $(\!|ct|\!)$, which makes it composable with $(\!|\vdash t'' \uparrow_{ct} t'|\!)$. We must also scatter the input to obtain data compatible with the input type of the replicated machine, which is $(\!|t|\!)[\lceil ct' \rceil][\lceil ct \rceil]$.
- For rule UPINV, we must build an output of type $(\!|t'|\!)$ from an input of type $(\!|t|\!)[\lceil ct \rceil]$, and we have a machine of type $(\!|t|\!) \boxminus (\!|t'|\!)[\lceil ct' \rceil]$ at our disposal. The solution is to replicate this machine by $(\!|ct|\!)$. We are left with an output of type $(\!|t'|\!)[\lceil ct' \rceil][\lceil ct \rceil]$, which we convert into one of type $(\!|t'|\!)[\lceil ct \,\mathbf{on}\, ct' \rceil]$ via the gathering machine. Since $ct \,\mathbf{on}\, ct' \equiv (1)$, we coerce this type to $(\!|t'|\!)[1]$, which can finaly be unwrapped to obtain a value in $(\!|t'|\!)$.

Scattering The scattering judgment $\vdash t \downarrow_{ct} t'$ is completely symmetric to the gathering one. Its definition is also given in Figure 4.20. Unzipping, gathering, and wrapping machines are replaced with zipping, scattering, and wrapping machines. They are composed in the reverse order. Contexts are scattered component-wise.

4.4.3 Translating Typed Programs

We are almost ready to describe the compilation of typed programs to machines. Let us address the last remaining technical issue.

Clock-polymorphic operators The source language includes a family of pointwise operators *op* left abstract. We assume given a machine $(\!|op|\!) : (\!|\square \vdash \mathbf{int} :: (1) \otimes \mathbf{int} :: (1) \multimap \mathbf{int} :: (1)|\!)$ for each operator *op* corresponding to its finite-state implementation.

This leads to the last important difference between the synchronous semantics of the source language and our kit of (macro-)machines. Notice that the typing rules OP, MERGE, and WHEN from Figure 3.5 accept any base clock *ct*: these rules are *clock-polymorphic*. In contrast, the machines $\mathtt{mhoconst}^n_m$, $(\!|op|\!)$, $\mathtt{mhomerge}_{mt}(m)$ and $\mathtt{mhowhen}_{mt}(m)$ handle only lists whose size is at most one. They would thus correspond to the *monomorphic* and *binary* typing rules below.

$$\textsc{MonoConst}\ \frac{}{\square \vdash s : dtof(s) :: (1)} \qquad \textsc{MonoOp}\ \frac{}{\square \vdash op : (\mathbf{int} :: (1)) \otimes (\mathbf{int} :: (1)) \multimap (\mathbf{int} :: (1))}$$

$$\textsc{MonoMerge}\ \frac{p \leq (1)}{\square \vdash \mathtt{merge}\ p : dt :: p \otimes dt :: \overline{p} \multimap dt :: (1)} \qquad \textsc{MonoWhen}\ \frac{p \leq (1)}{\square \vdash \mathtt{when}\ p : dt :: (1) \multimap dt :: p}$$

Fortunately, our language already includes a solution to the problem of adapting a machine to work on wider data types through the use of the driving machine: use replication. One should

perform conversions at the interface between the replicated machine and the external world. Those conversions can actually be described as derivations of the gathering and scattering judgments, which are then to be compiled to obtain the required machine. The conclusions of the relevant derivations are given below; p denotes periodic binary words.

$$\begin{aligned}
\mathit{empty}\downarrow_{ct} &::: \vdash \Box \downarrow_{ct} \Box \\
\mathit{const}\uparrow_{ct}^{dt} &::: \vdash dt :: (1) \uparrow_{ct} dt :: ct \\
\mathit{merge}\uparrow_{ct}^{p,dt} &::: \vdash (dt :: p) \otimes (dt :: \overline{p}) \multimap (dt :: (1)) \uparrow_{ct} (dt :: ct\,\mathbf{on}\,p) \otimes (dt :: ct\,\mathbf{on}\,\overline{p}) \multimap (dt :: ct) \\
\mathit{when}\uparrow_{ct}^{p,dt} &::: \vdash (dt :: (1)) \multimap (dt :: p) \uparrow_{ct} (dt :: ct) \multimap (dt :: ct\,\mathbf{on}\,p) \\
\mathit{op}\uparrow_{ct} &::: \vdash (\mathbf{int} :: (1)) \otimes (\mathbf{int} :: (1)) \multimap (\mathbf{int} :: (1)) \uparrow_{ct} (\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct) \multimap (\mathbf{int} :: ct)
\end{aligned}$$

Each name above has to be understood as an abbreviation for the derivation having the expected conclusion. They are readily constructed using the rules of Figure 3.9.

The translation The translation of the typing derivation for the typed program $\Gamma \vdash e : t$ produces a machine of type $(\!|\Gamma|\!) \boxminus (\!|t|\!)$. Let us define and explain each rule, case-by-case.

- Rule VAR: the machine erases the context but for its last component, which is returned via the higher-order identity machine.

$$(\!|\Gamma, x : t \vdash x : t|\!) \quad = \quad (\!|\vdash \Gamma \text{ value}|\!)_E \Vdash \mathtt{mhoid}_{(\!|t|\!)}$$

- Rule WEAKEN: the machine erases a single component from the context, which should be a value.

$$\begin{aligned}
(\!|\Gamma, x : t' \vdash e : t|\!) \quad = \quad & \mathtt{mhoneutr}_{(\!|t|\!)} \boxdot \langle (\!|t|\!), \mathtt{unit} \mapsto 1, 0 \rangle_{ho} \\
& \boxdot ((\!|\Gamma \vdash e : t|\!) \boxtimes (\!|\vdash t' \text{ value}|\!)_E)
\end{aligned}$$

  Note that we need to apply conversions on the output to obtain the expected type.

- Rule FUN: the translation is similar to the interpretation of the rule in the synchronous semantics, with domain operators replaced with machine constructors. It curryfies the premise in order to obtain the expected type.

$$(\!|\Gamma \vdash \mathtt{fun}\, x.\, e : t \multimap t'|\!) \quad = \quad \mathtt{mplug}(\mathtt{mcurry}^{(\!|t'|\!)}_{(\!|\Gamma|\!),(\!|t|\!)}, (\!|\Gamma, x : t \vdash e : t'|\!))$$

- Rule APP: the machine simply uses the internal application machine. Context splitting is compiled to a machine that provides the inputs required by the other premises.

$$\begin{aligned}
(\!|\Gamma \vdash e\ e' : t'|\!) \quad = \quad & \mathtt{mhoev}_{(\!|t|\!),(\!|t'|\!)} \boxdot ((\!|\Gamma_1 \vdash e : t \multimap t'|\!) \boxtimes (\!|\Gamma_2 \vdash e' : t|\!)) \\
& \boxdot (\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!)
\end{aligned}$$

- Rule PROD: the translation is again similar to the synchronous semantics, with the parallel product of machine replacing the pairing of continuous functions.

$$(\!|\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2|\!) \quad = \quad ((\!|\Gamma_1 \vdash e_1 : t_1|\!) \boxtimes (\!|\Gamma_2 \vdash e_2 : t_2|\!)) \boxdot (\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!)$$

$$
\begin{array}{l}
\quad (\!|\Gamma \vdash e : t|\!) \\
\in (\!|\Gamma|\!) \boxminus (\!|t|\!)
\end{array}
$$

$$
\begin{array}{l}
\quad (\!|\Gamma, x : t \vdash x : t|\!) \\
= (\!|\vdash \Gamma \text{ value}|\!)_E \Vdash \mathtt{mhoid}_{(\!|t|\!)}
\end{array}
$$

$$
\begin{array}{l}
\quad (\!|\Gamma, x : t' \vdash e : t|\!) \\
= \mathtt{mhoneutr}_{(\!|t|\!)} \boxdot \langle (\!|t|\!), \mathtt{unit} \mapsto 1, 0 \rangle_{ho} \boxdot ((\!|\Gamma \vdash e : t|\!) \boxtimes (\!|\vdash t' \text{ value}|\!)_E)
\end{array}
$$

$$
\begin{array}{l}
\quad (\!|\Gamma \vdash \mathtt{fun}\, x.e : t \multimap t'|\!) \\
= \mathtt{mplug}(\mathtt{mcurry}^{(\!|t'|\!)}_{(\!|\Gamma|\!),(\!|t|\!)}, (\!|\Gamma, x : t \vdash e : t'|\!))
\end{array}
$$

$$
\begin{array}{l}
\quad (\!|\Gamma \vdash e\ e' : t'|\!) \\
= \mathtt{mhoev}_{(\!|t|\!),(\!|t'|\!)} \boxdot ((\!|\Gamma_1 \vdash e : t \multimap t'|\!) \boxtimes (\!|\Gamma_2 \vdash e' : t|\!)) \boxdot (\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!)
\end{array}
$$

$$
\begin{array}{l}
\quad (\!|\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2|\!) \\
= ((\!|\Gamma_1 \vdash e_1 : t_1|\!) \boxtimes (\!|\Gamma_2 \vdash e_2 : t_2|\!)) \boxdot (\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!)
\end{array}
$$

$$
\begin{array}{l}
\quad (\!|\Gamma \vdash \mathtt{let}\ (x, y) = e\ \mathtt{in}\ e' : t|\!) \\
= \mathtt{mhoev}_{(\!|t_1 \otimes t_2|\!),(\!|t|\!)} \boxdot (\mathtt{mplug}(\mathtt{mcurry}^{(\!|t|\!)}_{(\!|\Gamma|\!),(\!|t_1|\!) \boxtimes (\!|t_2|\!)}, (\!|\Gamma_2, x : t_1, y : t_2 \vdash e' : t|\!)) \boxtimes (\!|\Gamma_1 \vdash e : t_1 \otimes t_2|\!)) \boxdot (\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!)
\end{array}
$$

$$
\begin{array}{l}
\quad (\!|\Gamma \vdash \mathtt{fix}\, e : t'|\!) \\
= \mathtt{mhofb}^{(\!|t'|\!)}_{(\!|\Gamma|\!),(\!|t'|\!)}((\!|\vdash t' \text{ value}|\!)_D \boxdot (\mathtt{mplug}(\mathtt{muncurry}^{(\!|t'|\!)}_{(\!|\Gamma|\!),(\!|t|\!)}, (\!|\Gamma \vdash e : t \multimap t'|\!))) \boxdot (\mathtt{mhoid}_{(\!|\Gamma|\!)} \boxtimes (\!|\vdash t' <:_1 t|\!)))
\end{array}
$$

$$
\begin{array}{l}
\quad (\!|\square \vdash s : dtof(s) :: ct|\!) \\
= (\!|const \uparrow^{dtof(s)}_{ct}|\!) \boxdot \mathtt{mplug}(\mathtt{mhorepl}^{\lceil ct \rceil}_{(\!|\square|\!),(\!|\mathbf{int} :: (1)|\!)}(\mathtt{mhowrap}_{(\!|dtof(s)|\!)} \boxdot \mathtt{mhoconst}^{s}_{dtof(s)}), (\!|ct|\!)) \boxdot (\!|empty \downarrow_{ct}|\!)
\end{array}
$$

$$
\begin{array}{l}
\quad (\!|\square \vdash op : (\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct) \multimap (\mathbf{int} :: ct)|\!) \\
= (\!|op \uparrow_{ct}|\!) \boxdot \mathtt{mplug}(\mathtt{mhorepl}^{\lceil ct \rceil}_{(\!|\square|\!),(\!|(\mathbf{int} :: (1)) \otimes (\mathbf{int} :: (1)) \multimap (\mathbf{int} :: (1))|\!)}((\!|op|\!)), (\!|ct|\!)) \boxdot (\!|empty \downarrow_{ct}|\!)
\end{array}
$$

$$
\begin{array}{l}
\quad (\!|\square \vdash \mathtt{merge}\ p : (dt :: ct\ \mathbf{on}\ p) \otimes (dt :: ct\ \mathbf{on}\ \overline{p}) \multimap (dt :: ct)|\!) \\
= (\!|merge \uparrow^{p,dt}_{ct}|\!) \boxdot \mathtt{mplug}(\mathtt{mhorepl}^{\lceil ct \rceil}_{(\!|\square|\!),(\!|(dt :: p) \otimes (dt :: \overline{p}) \multimap (dt :: (1))|\!)}(\mathtt{mhomerge}_{(\!|dt|\!)}(\mathtt{mpw}_p(0))), (\!|ct|\!)) \boxdot (\!|empty \downarrow_{ct}|\!)
\end{array}
$$

$$
\begin{array}{l}
\quad (\!|\square \vdash \mathtt{when}\ p : (dt :: ct) \multimap (dt :: ct\ \mathbf{on}\ p)|\!) \\
= (\!|when \uparrow^{p,dt}_{ct}|\!) \boxdot \mathtt{mplug}(\mathtt{mhorepl}^{\lceil ct \rceil}_{(\!|\square|\!),(\!|(dt :: (1)) \multimap (dt :: p)|\!)}(\mathtt{mhowhen}_{(\!|dt|\!)}(\mathtt{mpw}_p(0))), (\!|ct|\!)) \boxdot (\!|empty \downarrow_{ct}|\!)
\end{array}
$$

$$
\begin{array}{l}
\quad (\!|\Gamma \vdash e : t'|\!) \\
= (\!|\vdash t <:_k t'|\!) \boxdot (\!|\Gamma \vdash e : t|\!)
\end{array}
$$

$$
\begin{array}{l}
\quad (\!|\Gamma \vdash e : t|\!) \\
= (\!|\vdash t' \uparrow_{ct} t|\!) \boxdot \mathtt{mplug}(\mathtt{mhorepl}^{\lceil ct \rceil}_{(\!|\Gamma'|\!),(\!|t'|\!)}((\!|\Gamma' \vdash e : t'|\!)), (\!|ct|\!)) \boxdot (\!|\vdash \Gamma \downarrow_{ct} \Gamma'|\!)
\end{array}
$$

Figure 4.21: Compilation - main typing judgment

- Rule LET: this case is similar to that of application, except that some currying has to be performed in order to obtain a machine expecting inputs in $(\!|t_1 \otimes t_2|\!)$ from the compilation of the judgment $\Gamma, x : t_1, y : t_2 \vdash e : t$.

$$\begin{array}{rl} & (\!|\Gamma \vdash \texttt{let}\ (x,y) = e\ \texttt{in}\ e' : t|\!) \\ = & \texttt{mhoev}_{(\!|t_1 \otimes t_2|\!),(\!|t|\!)} \boxdot (m \boxtimes (\!|\Gamma_1 \vdash e : t_1 \otimes t_2|\!)) \boxdot (\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!) \\ & \text{where } m = \texttt{mplug}(\texttt{mcurry}^{(\!|t|\!)}_{(\!|\Gamma|\!),(\!|t_1|\!) \boxtimes (\!|t_2|\!)}, (\!|\Gamma_2, x : t_1, y : t_2 \vdash e' : t|\!)) \end{array}$$

- Rule FIX: we use the higher-order feedback loop to compile `fix` e. The premises are used to build a machine of the type expected by the feedback operator. The compiled adaptability judgment $(\!|\vdash t' <:_1 t|\!)$ is placed in front of the compiled machine of e while the duplication machine $(\!|\vdash t'\ \text{value}|\!)$ is placed behind it.

$$\begin{array}{rcl} (\!|\Gamma \vdash \texttt{fix}\, e : t'|\!) & = & \texttt{mhofb}^{(\!|t'|\!)}_{(\!|\Gamma|\!),(\!|t'|\!)}((\!|\vdash t'\ \text{value}|\!)_D \boxdot m \boxdot (\texttt{mhoid}_{(\!|\Gamma|\!)} \boxtimes (\!|\vdash t' <:_1 t|\!))) \\ & & \text{where } m = \texttt{mplug}(\texttt{muncurry}^{(\!|t'|\!)}_{(\!|\Gamma|\!),(\!|t|\!)}, (\!|\Gamma \vdash e : t \multimap t'|\!)) \end{array}$$

- Rule CONST: constants are handled using the strategy at the beginning of this section. They are replicated and enclosed in conversion code obtained from the compilation of the relevant gathering and scattering judgments.

$$\begin{array}{rl} & (\!|\square \vdash s : dtof(s) :: ct|\!) \\ = & (\!|const \uparrow^{dtof(s)}_{ct}|\!) \boxdot \texttt{mplug}(\texttt{mhorepl}^{\lceil ct \rceil}_{(\!|\square|\!),(\!|\mathbf{int} :: (1)|\!)}(m), (\!|ct|\!)) \boxdot (\!|empty \downarrow_{ct}|\!) \\ & \text{where } m = \texttt{mhowrap}_{(\!|dtof(s)|\!)} \boxdot \texttt{mhoconst}^{s}_{dtof(s)} \end{array}$$

- Rule OP: similar to constants.

$$\begin{array}{rl} & (\!|\square \vdash op : (\mathbf{int} :: ct) \otimes (\mathbf{int} :: ct) \multimap (\mathbf{int} :: ct)|\!) \\ = & (\!|op \uparrow_{ct}|\!) \boxdot \texttt{mplug}(\texttt{mhorepl}^{\lceil ct \rceil}_{(\!|\square|\!),(\!|(\mathbf{int} :: (1)) \otimes (\mathbf{int} :: (1)) \multimap (\mathbf{int} :: (1))|\!)}(m), (\!|ct|\!)) \boxdot (\!|empty \downarrow_{ct}|\!) \\ & \text{where } m = (\!|op|\!) \end{array}$$

- Rule MERGE: again, similar to constants.

$$\begin{array}{rl} & (\!|\square \vdash \texttt{merge}\ p : (dt :: ct\ \mathbf{on}\ p) \otimes (dt :: ct\ \mathbf{on}\ \overline{p}) \multimap dt :: ct|\!) \\ = & (\!|merge \uparrow^{p,dt}_{ct}|\!) \boxdot \texttt{mplug}(\texttt{mhorepl}^{\lceil ct \rceil}_{(\!|\square|\!),(\!|(dt :: p) \otimes (dt :: \overline{p}) \multimap (dt :: (1))|\!)}(m), (\!|ct|\!)) \boxdot (\!|empty \downarrow_{ct}|\!) \\ & \text{where } m = \texttt{mhomerge}_{(\!|dt|\!)}(\texttt{mpw}_p(0)) \end{array}$$

- Rule WHEN: again, similar to constants.

$$\begin{array}{rl} & (\!|\square \vdash \texttt{when}\ p : (dt :: ct) \multimap (dt :: ct\ \mathbf{on}\ p)|\!) \\ = & (\!|when \uparrow^{p,dt}_{ct}|\!) \boxdot m \boxdot (\!|empty \downarrow_{ct}|\!) \\ & \text{where } m = \texttt{mplug}(\texttt{mhorepl}^{\lceil ct \rceil}_{(\!|\square|\!),(\!|(dt :: (1)) \multimap (dt :: p)|\!)}(\texttt{mhowhen}_{(\!|dt|\!)}(\texttt{mpw}_p(0))), (\!|ct|\!)) \end{array}$$

- Rule ADAPT: as usual with subtyping-like rules, we compose the machine obtained from the adaptability judgment.

$$(\!|\Gamma \vdash e : t'|\!) \quad = \quad (\!|\vdash t <:_k t'|\!) \boxdot (\!|\Gamma \vdash e : t|\!)$$

- Rule RESCALE: the strategy is the same as in the case of constants, operators, sampling and merging, except that in this case we scatter an arbitrary program rather than a fixed operator.

$$(\!|\Gamma \vdash e : t|\!) \quad = \quad (\!|\vdash t' \uparrow_{ct} t|\!) \boxdot \mathtt{mplug}(\mathtt{mhorepl}^{\lceil ct \rceil}_{(\!|\Gamma'|\!),(\!|t'|\!)}((\!|\Gamma' \vdash e : t'|\!)),(\!|ct|\!)) \boxdot (\!|\vdash \Gamma \downarrow_{ct} \Gamma'|\!)$$

Figure 4.21 recapitulates all the cases of the translation in one place. The next section discusses what it means for a machine to correctly implement a source-level program and then builds on top of this to prove that the translation is actually correct.

4.4.4 Soundness

This section studies the correctness of the translation using a binary logical relation. The relation defines what it means for a machine to implement a domain element, for example a stream or a function. The soundness theorem expresses that the compilation of a typing derivation implements its typed semantics.

Operational equivalence We construct the logical relation using a *biorthogonality* technique [Girard, 1987; Pitts, 2000]. Biorthogonality techniques rely on a notion of well-behaved interaction between a program and its surrounding context. In our case, orthogonality is a kind of bisimulation defined as follows.

Definition 18 (Orthogonality)**.** *Orthogonality is a coinductive relation between machines of complementary types. Two machines m_1 and m_2 are* orthogonal*, written $m_1 \perp m_2$, when the following conditions hold for all values x and y:*

- *if $m_1/x \to m_1'/y$ then there exists m_2' such that $m_2/y \to m_2'/x$ and $m_1' \perp m_2'$;*
- *if $m_2/y \to m_2'/x$ then there exists m_1' such that $m_1/x \to m_1'/y$ and $m_1' \perp m_2'$.*

Remark 21. Readers familiar with synchronous programming may notice an analogy with the notion of *synchronous observer* [Halbwachs et al., 1994]. However here the machine m_2 does not simply observe the output of m_1, but also affects its input—orthogonality is symmetric.

Definition 19 (Orthogonal set)**.** *The* orthogonal *of a typed set of machines $X \subseteq \mathcal{M}_{mtm}$ is the typed set of machines written $X^\perp \subseteq \mathcal{M}_{mtm^*}$ defined by $m_* \in X^\perp \Leftrightarrow \forall m \in X, m \perp m_*$.*

Property 42 (Classic properties of orthogonal sets)**.** *For any set of machines X we have $X \subseteq X^{\perp\perp}$ and $X^{\perp\perp\perp} = X^\perp$. Given another set of machines Y such that $X \subseteq Y$, we have $Y^\perp \subseteq X^\perp$. For any sets of machines X, Y we have $X^\perp \cap Y^\perp = (X \cup Y)^\perp$*

Proof. Each property is immediately proved by unfolding the definition of $X^\perp$. They do not depend on the definition of the orthogonality relation. □

The machines in $X^\perp$ are precisely those that combine well with all machines in X: their interaction with elements of X never blocks. The machines of $X^\perp$ can be thought of as *tests* for X, and vice-versa. The interest of this construction is that we can now define sets of machines negaitvely, by taking the orthogonal of the set of tests they must pass. Moreover, the orthogonal builds closed sets, which intuitively describe sets of machine that behave in the same way. The importance of closed sets motivate the following definition.

Definition 20 (Behaviors)**.** *A set of machines X is a* behavior *if $X^{\perp\perp} = X$ holds, or, equivalently, if X is of the form $Y^\perp$ for some Y.*

Property 43 (Intersection of behaviors)**.** *The intersection of two behaviors is a behavior.*

Proof. Let H_1 and H_2 be behaviors. By Definition 20 and Property 43, we have

$$H_1 \cap H_2 = H_1^{\perp\perp} \cap H_2^{\perp\perp} = \left(H_1^\perp \cup H_2^\perp\right)^\perp = \left(H_1 \cap H_2\right)^{\perp\perp}$$

which proves that $H_1 \cap H_2$ is a behavior. □

Notations We lift all the machine combinators to sets of machines. For instance, given sets of machines X, Y, we write $X \bullet Y$ for the set of machines $\{m_1 \bullet m_2 \mid m_1 \in X, m_2 \in Y\}$. This also applies to macro-machines or higher-order machine combinators, for instance $X \boxdot Y$ stands for the set of machines $\{m_1 \boxdot m_2 \mid m_1 \in X, m_2 \in Y\}$. As an exception to the above rule, we write $X \boxtimes Y$ for $\{m_1 \parallel m_2 \mid m_1 \in X, m_2 \in Y\}$. Finally, using these notations, we write $X \boxminus Y$ for $X^\perp \boxtimes Y = \{m_1 \parallel m_2 \mid m_1 \in X^\perp, m_2 \in Y\}$.

Compatibility Next, we need properties that express that all our machine constructors behave well with respect to orthogonality. For example, given two machines m_1 and m_2 respectively orthogonal to m_3 and m_4, the horizontal composition $m_1 \bullet m_2$ must be orthogononal to $m_3 \bullet m_4$. As before, we write $X_1 \bullet X_2$ for $\{m_1 \bullet m_2 \mid m_1 \in X_1, m_2 \in X_2\}$, and similarly for other machine combinators.

Lemma 12 (Compatibility)**.** *Orthogonality is compatible with the machine combinators of the basic language. More precisely, for any sets of machines X, Y, we have*

- $X^\perp \bullet Y^\perp \subseteq (X \bullet Y)^\perp$;
- $X^\perp \boxtimes Y^\perp \subseteq (X \boxtimes Y)^\perp$;
- $\mathtt{mfb}^{mt_3}_{mt_1,mt_2}(X^\perp) \subseteq (\mathtt{mfb}^{mt_3}_{mt_1,mt_2}(X))^\perp$;
- $\mathtt{mrepl}_n(X^\perp) \subseteq (\mathtt{mrepl}_n(X))^\perp$.

$$\frac{\vdash^{n} xl : mt}{\vdash \mathtt{mtest}_{mt}(xl) : mt \multimap \mathtt{unit}}$$

$$\overline{\mathtt{mtest}_{mt}(x;xl)/x \to \mathtt{mtest}_{mt}(xl)/()} \qquad \overline{\mathtt{mtest}_{mt}([])/\_ \to \mathtt{mtest}_{mt}([])/()}$$

Figure 4.22: Typing and reaction rules for the testing machine

Proof. The proofs are done by coinduction. Let us detail the vertical composition case. We are given two machines $m_1 \in X_1^{\perp}$ and $m_2 \in X_2^{\perp}$. We must prove that for any $m \in X_1 \boxtimes X_2$, we have $(m_1 \Vdash m_2) \perp m$. Since $m \in X_1 \boxtimes X_2$, m is by definition of the form $m_3 \Vdash m_4$, with $m_3 \in X_1$ and $m_4 \in X_2$. Moreover, our hypotheses give $m_1 \perp m_3$ and $m_2 \perp m_4$. By coinduction, we show $(m_1 \Vdash m_2) \perp (m_3 \Vdash m_4)$.

- Assume $(m_1 \Vdash m_2)/(x_1, x_2) \to (m_1' \Vdash m_2')/(y_1, y_2)$ holds. By definition of the reaction judgment, this means that $m_1/x_1 \to m_1'/y_1$ and $m_2/x_2 \to m_2'/y_2$. Since $m_1 \perp m_3$, there exists m_3' such that $m_3/y_1 \to m_3'/x_1$ and $m_1' \perp m_3'$. Similarly, since $m_2 \perp m_4$, there exists m_4' such that $m_4/y_2 \to m_4'/x_2$ and $m_2' \perp m_4'$. Thus $(m_3 \Vdash m_4)/(y_1, y_2) \to (m_3' \Vdash m_4')/(x_1, x_2)$ holds by definition of the reaction judgment. We apply the coinduction hypothesis to obtain $(m_1' \Vdash m_2') \perp (m_3' \Vdash m_4')$ and conclude $(m_1 \Vdash m_2) \perp (m_3 \Vdash m_4)$.
- The other direction, when $m_3 \Vdash m_4$ makes a step, is completely symmetric. □

The Compatiblity Lemma must be lifted to the higher-order machine types defined in Section 4.3 in a similar way. This makes it possible to prove that, for instance, the higher-order application and composition combinators built using feedback loops are sound. Let us state some important cases.

Property 44. *For any set $X \subseteq \mathcal{M}_{mtm}$, the machine $\mathtt{mhoid}_{mtm}$ belongs to $(X \boxtimes X^{\perp})^{\perp}$.*

Property 45. *Let H_1 and H_2 be two behaviors. Let m_1 be a machine in $(H_1 \boxtimes H_2^{\perp})^{\perp}$ and m_2 be a machine in H_1. Then $\mathtt{mplug}(m_1, m_2)$ belongs to H_2.*

Proof sketch. We must show that $\mathtt{mplug}(m_1, m_2)$ is orthogonal to any machine $m_y \in H_2^{\perp}$. The definition of the application combinator applies the feedback machine to a machine m built by applying permutations to $m_1 \Vdash m_2$. Using the compatibility lemma we can show that m belongs to $(H_1 \boxtimes H_1^{\perp} \boxtimes H_2^{\perp})^{\perp}$. From this we deduce that $m \perp (\mathtt{mhoid}_{mtm_1} \Vdash m_y)$, with mtm_1 the type of the behavior H_1. This hypothesis is sufficient to show that $\mathtt{mplug}(m_1, m_2) \perp m_y$ holds by a direct coinductive argument. □

Testing To capture the correctness of compilation using orthogonal sets, we need a sufficiently expressive set of tests. It turns out that the machine language presented in the previous section is not expressive enough. We thus enrich it with a *testing machine*, $\mathtt{mtest}_{mt}(xl)$. This

$$
\begin{aligned}
Impl_t(\_ \in \mathbf{S}[\![t]\!]) &\subseteq \mathcal{M}_{(\!|t|\!)} \\
Impl_{dt :: ct}(xs) &= (\bigcup_{xl \sqsubseteq_{fin} xs} \{\mathtt{mtest}_{(\!|dt|\!)[\lceil ct \rceil]}((\!|xl|\!))\})^{\perp} \\
Impl_{t_1 \otimes t_2}(x_1, x_2) &= (Impl_{t_1}(x_1) \boxtimes Impl_{t_2}(x_2))^{\perp\perp} \\
Impl_{t_1 \multimap t_2}(f) &= \bigcap_{x \in \mathbf{S}[\![t_1]\!]} (Impl_{t_1}(x) \boxtimes Impl_{t_2}(f(x))^{\perp})^{\perp}
\end{aligned}
$$

$$
\begin{aligned}
Impl_{\Gamma}(\_ \in \mathbf{S}[\![\Gamma]\!]) &\subseteq \mathcal{M}_{(\!|\Gamma|\!)} \\
Impl_{\square}(\_) &= \{\mathtt{mforg}_{\mathtt{unit}}\}^{\perp\perp} \\
Impl_{\Gamma, x:t}(\gamma, v) &= (Impl_{\Gamma}(\gamma) \boxtimes Impl_t(v))^{\perp\perp}
\end{aligned}
$$

$$
\begin{aligned}
Impl_{\Gamma \vdash t}(f \in \mathbf{S}[\![\Gamma \vdash t]\!]) &\subseteq \mathcal{M}_{(\!|\Gamma|\!) \boxminus (\!|t|\!)} \\
Impl_{\Gamma \vdash t}(f) &= \bigcap_{\gamma \in \mathbf{S}[\![\Gamma]\!]} (Impl_{\Gamma}(\gamma) \boxtimes Impl_t(f(\gamma))^{\perp})^{\perp}
\end{aligned}
$$

Figure 4.23: Soundness proof - implementations of types, contexts, and typings

machine has one input of type mt and no outputs. Its state is a finite list xl of type mt. When xl is empty, the machine reacts without looking at its input. When it is non-empty, the machine reacts only if its argument is the current head of xl, which is then removed from its state. Thus, if xl is a list of length n, this machine checks whether its first n inputs are the elements of xl.

The typing and reaction rules of the testing machine are given in Figure 4.22. They respect all the properties given in Section 4.2.2. In particular, the size of its state strictly decreases at each reaction, and thus the machine is finite state.

The logical relation Given a source-type t and element x belonging to the domain $\mathbf{S}[\![t]\!]$, the behavior $Impl_t(x)$ is the set of machines of type $(\!|t|\!)$ which are valid implementations of x. We call $Impl_t(x)$ the set of *implementations* of x. It is defined by induction over t.

- A machine implements a stream xs if it combines safely with all the machines testing its finite prefixes. More precisely, to implement a stream xs living in $\mathbf{S}[\![dt :: ct]\!]$, one should pass the test $\mathtt{mtest}_{(\!|dt :: ct|\!)}(xl)$, for any list xl that corresponds to a finite prefix of xs.

$$Impl_{dt :: ct}(xs) \stackrel{\text{def}}{=} (\bigcup_{xl \sqsubseteq_{fin} xs} \{\mathtt{mtest}_{(\!|dt :: ct|\!)}((\!|xl|\!))\})^{\perp}$$

  The notation $xs' \sqsubseteq_{fin}$ means that $xs' \sqsubseteq xs$ and xs' converges *only* up to some n, and $(\!|xs'|\!)$ is the translation of such a finitely-converging xs' into a list of values of size n.

- A machine implements a pair (x_1, x_2) when it cannot be distinguished from a parallel pair of an implementation of x_1 and one of x_2.

$$Impl_{t_1 \otimes t_2}(x_1, x_2) \stackrel{\text{def}}{=} (Impl_{t_1}(x_1) \boxtimes Impl_{t_2}(x_2))^{\perp\perp}$$

  The notation $X \boxtimes Y$ denotes the set $\{m_1 \parallel m_2 \mid m_1 \in X, m_2 \in Y\}$.

- A machine implements a function f when, for any argument x, it passes all the tests that provide an implementation of x and a test of an implementation of $f(x)$.

$$Impl_{t_1 \multimap t_2}(f) \stackrel{\text{def}}{=} \bigcap_{x \in \mathbf{S}[\![t_1]\!]} (Impl_{t_1}(x) \boxtimes Impl_{t_2}(f(x))^{\perp})^{\perp}$$

The complete definition is recalled in Figure 4.23, together with the administrative definition of implementations of a context Γ and of a typing $\Gamma \vdash t$. Observe that any set of implementations is a behavior, since they are all defined using orthogonals and intersections.

Theorem 8 (Soundness)**.** *The machine resulting from the compilation of a well-typed program implements its typed semantics. In other words, given a derivation of* $\Gamma \vdash e : t$*, we have*

$$(\!|\Gamma \vdash e : t|\!) \in Impl_{\Gamma \vdash t}(\mathbf{S}[\![\Gamma \vdash e : t]\!])$$

Proof. The proof is done by induction on typing derivations. The theorem must be generalized to auxiliary judgments in the expected way to obtain proper induction hypotheses. Each step of the proof falls into one of the two following cases.

- Stream-specific language constructs—such as gathering, scattering, and buffering—are handled by induction on the length of stream prefixes. Such cases make direct use of the definition of the reaction judgment and are routine.

- Language constructs that are not specific to streams but the traditional features of a linear lambda-calculus with pairs—such as function application, pairing, and so on—are proved using compatiblity properties such as Property 45.

This shows that the compiled machines implement the typed semantics. □

Note that Theorem 8 does not implies that the machines are deadlock-free in isolation. This comes from the definition of our orthogonality reaction, which does not ensure that making a step is possible. However, using the results obtained in Chapter 3, we may prove the following result.

Corollary 2 (Reactivity)**.** *For any e, dt, ct, the machine* $(\!|\Box \vdash e : dt :: ct|\!)$ *is able to react forever.*

Proof. By Theorem 4, the stream $\mathbf{S}[\![\Box \vdash e : dt :: ct]\!]$ is total. Hence, it has prefixes of all lengthes. By Theorem 8 and definition of the testing machine, this implies that $(\!|\Box \vdash e : dt :: ct|\!)$ is able to react for n steps, with n arbitrary. □

Remark 22. The characterization of a stream via the set of tests of all its finite prefixes is directly inspired from Melliès and Vouillon [2005]. We are grateful to Paul-André Melliès for having drawn our attention to this technique and explained the underlying intuitions.

4.5 From Machines to Circuits

The discussion at the end of Section 4.2 outlined a naive way of compiling our state machines to circuits, seen as explicit finite automata. This technique is unrealistic since it manipulates an explicit automaton which may have a size exponential in that of the source program. Also, industrial circuit synthesis toolchains process hierarchical descriptions, written in Hardware Description Languages (HDLs) such as VHDL or Verilog, rather than flat ones. This section briefly discusses the few steps that remain to be performed to implement machines on top of an HDL. We begin with an informal discussion of some potential optimizations that could be performed during the translation.

4.5.1 Optimizations

The translation process described in this chapter has the typical strengths and drawbacks of type-directed interpretations. Once both the translation of types and the type of the translation have been defined, the rest follows in a mechanical fashion. The other side of the coin is that the generated code includes a lot of conversion machines that often perform no useful work. Let us discuss some optimizations that can be performed at the target-level to simplify the generated machines.

Removing boilerplate Because the translation is type-preserving, all source programs of the same type are compiled to the same target type. In particular, all clocked streams are translated to machines computing lists. It might be more efficient to replace those types to more specialized ones in some cases, such as translating a program of type $dt :: (1)$ to a machine with outputs in $(\!|dt|\!)$ rather than $(\!|dt|\!)[1]$. Similarly, the clock type (2) could be translated to pairs rather than bounded lists, and so on.

A related point is the removal or fusion of "administrative" machines a compiled program has been sprinkled with. For instance, the translation introduces a lot of permutations, sometimes applied consecutively. It might be a good idea to fuse them by composing the permutations. Also, the generated code may exhibit composition of conversion machines that compose to the identity function when applied to programs coming from well-typed programs, such as unzipping followed by zipping. It should be possible to prove that those are equivalent using the logical relation from the previous section.

Clock types and periodic words Another performance question is the efficient compilation of ultimately periodic words that arise from clock types. This issue has less to do with language-level concerns than with optimization questions, potentially of a combinatorial nature. A given closed typing derivation might feature a variety of ultimately periodic words. In particular, some of them might not have been present in source code but rather inserted by some type inference process. Translating these words to machines involve a variety of interacting questions and trade-offs.

- Specialized representations: the direct and most general implementation $\mathtt{mpw}_{u(v)}(i)$ uses an ultimately periodic counter for i and an array of length $|u|+|v|$ storing the integers in $u(v)$. Clearly, some words have more space-efficient representations, such as constant ones. Others words might have special characteristics that lead to circuits with better critical paths, and so on.

- Sharing: in general one is interested in the efficient implementation of several words rather than a single one. Equivalent words can be implemented by a unique machine and shared between all their consumers.

- Factorization: sharing makes it beneficial to express redundancy between distinct clock types. For instance one can share $(\!|ct|\!)$ when computing $(\!|ct \textbf{ on } ct_1|\!)$ and $(\!|ct \textbf{ on } ct_2|\!)$ in the same expression. Note that the converse is not true for $(\!|ct_1 \textbf{ on } ct|\!)$ and $(\!|ct_2 \textbf{ on } ct|\!)$ as ct is driven by a distinct clock in each case. Thus, one might want to try to factor equivalent prefixes out of the set of clock types.

The answer to these questions probably depend on the underlying platform. In particular, software implementations might have performance needs very different from hardware ones. We hope however that the relationship between 2-adic numbers and digital circuits uncovered by Vuillemin [1994] might help organize the design space.

Remark 23. The optimizations discussed above have to be performed during the translation. This is unsatisfying, since it increases its complexity; we would rather like to explain them as source-to-source passes on one intermediate language. Unfortunately, this is possible neither at the source nor target level. On the one hand, the source language misses the operators and types for expressing the above operations. On the other, useful clocking information is lost during the translation, and moreover equivalent programs may not be compiled to equivalent machines. There are at least two solutions.

- We could design a new intermediate language that enriches the source with additional operators, so that the translation to machines would factor through this new language. Optimizations would then be performed at the relevant level.

- A potentially less ambitious approach would be to make the machine type system more precise. In particular, a low-hanging fruit is to have lower bounds in addition to upper ones in type constructors. This would make more optimizations feasible at the machine level, at the cost of a slightly more complex translation and correctness proof.

We have not investigated either one since only concrete experiments on non-toy programs would reveal significant differences.

4.5.2 Desconstruction Replication

Readers versed in digital hardware circuits might have found the replicating machine suspicious. We know that in theory this machine can be implemented as a circuit since it is finite state. However, and in contrast with all other machines, a direct translation to an HDL does not

$$\frac{\vdash m : mt_1 \multimap mt_2}{\vdash \texttt{mgate}(m) : \{1\} \times mt_1[1] \multimap mt_2[1]} \qquad \frac{\vdash m : mtm}{\vdash \texttt{mturbo}_n(m) : mtm[n]}$$

$$\frac{}{\vdash \texttt{mselect}_n : \{n\} \multimap \{1\}[n]}$$

(a) Typing rules

$$\frac{}{\texttt{mgate}(m)/(0,[]) \to \texttt{mgate}(m)/[]} \qquad \frac{m/x \to m'/y}{\texttt{mgate}(m)/(1,[x]) \to \texttt{mgate}(m')/[y]}$$

$$\frac{m/xl \to_n m'/yl}{\texttt{mturbo}_n(m)/xl \to \texttt{mturbo}_n(m')/yl} \qquad \frac{select(n,a,xl)}{\texttt{mselect}_n/a \to \texttt{mselect}_n/xl}$$

(b) Reaction rules

Figure 4.24: Machines - special cases of replication

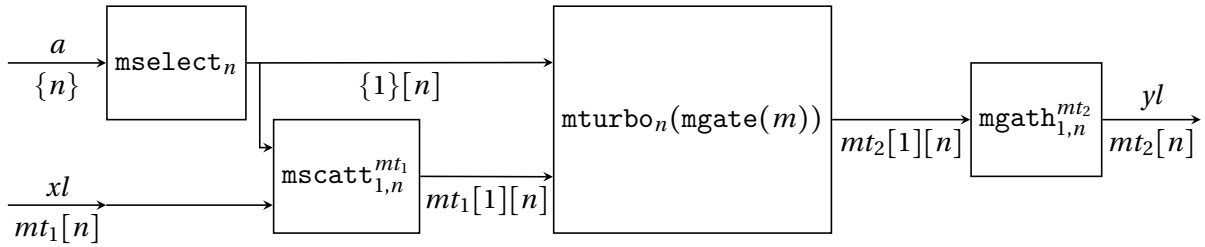


Figure 4.25: Replication - implementing $\texttt{mrepl}_n(m)$ using machines from Figure 4.24

seem trivial as it performs a varying number of reactions of the replicated machine for each of its own reactions.

Figure 4.24 introduces new primitive machines that can be used to implement replication. The first two machines are specialized cases of replication. The machine $\texttt{mgate}(m)$ corresponds to binary replication: the machine m performs at most one reaction. We call this machine the *gating* machine, in analogy to the clock gating of circuits. The machine $\texttt{mturbo}_n(m)$ corresponds to constant replication: the machine m always performs exactly n reactions. We call this machine the *turbo* machine, in a loose analogy with the clock scaling features of recent processors. The stateless *selection* machine $\texttt{mselect}_n$ computes a unary representation xl of its input $a \leq n$ on n bits. This is expressed formally by the predicate $select(n,a,xl)$ given below.

$$select(n,a,xl) \quad = \quad |xl| = n \text{ and } \forall 0 \leq i < n, xl[i] = \text{if } a \leq i \text{ then 1 else 0}$$

We will soon explain a straightforward implementation of these machines, assuming that any circuit implementing a machine has a uniform interface. What we would like to do for now

is to reduce any replication machine to a composition of the new machines. Remember that at each reaction the machine $\texttt{mrepl}_n(m)$ receives an input $a \leq n$ expressing how many reactions of n should be performed at once. The idea is to use a turbo machine to perform exactly n reaction of a gating machine which itself may or may not react according to the current value of a This gating machine processes a list of n booleans that controls whether the machine m should react at each of the local n steps induced by the turbo machine. This list of booleans is computed by the $\texttt{mselect}_n$ machine from the input a. The construction requires the use of conversion machines in some places in order to be well-typed. The precise definition can be found below, assuming the machine m has type $mt_1 \multimap mt_2$.

$$\begin{aligned}\texttt{mrepl}_n(m) \;=\; & \texttt{mgath}^{mt_2}_{1,n} \bullet \texttt{mturbo}_n(\texttt{mgate}(m)) \bullet \texttt{mzip}^n_{\{1\},mt_1} \bullet (\texttt{mid}_{\{1\}[n]} \parallel \texttt{mscatt}^{mt_1}_{1,n}) \\ & \bullet ((\texttt{mdup}_{\{1\}[n]} \bullet \texttt{mselect}_n) \parallel \texttt{mid}_{mt_1[n]})\end{aligned}$$

Figure 4.25 gives a graphical version of the above definition which is easier to read. The label below each wire gives its type and the label above, if any, gives the name of the corresponding values in the reaction rule of the replication machine (Figure 4.7). The zipping conversion in front of the central machine has been omitted.

Remark 24. Note that the reaction rule of the selection machine is somewhat arbitrary, as the above scheme works for any boolean list of length n as long as it contains exactly a ones. This can be understood by considering the analogous phenomenon at the source level. Semantically, the construction above corresponds to the decomposition of a local time scale driven by some clock $w \leq (n)^\omega$ into two new time scales applied successively, the outer one driven by $(n)^\omega$ and the inner one by any binary clock type w_b such that $w = (n)^\omega \; on \; w_b$. There are in general multiple choices for w_b, with distinct choices reflected as a distinct control lists being transmitted to $\texttt{mgate}(m)$ at runtime. The reaction rule for $\texttt{mselect}_{\lceil ct \rceil}$ given in Figure 4.24 actually corresponds to the *earliest* such w_b, that is the clock in which ones are inserted as soon as possible

4.5.3 Towards Circuits

We finish this section by discussing how machines can be implemented in real hardware description languages.

HDLs Hardware Description Languages such as VHDL [IEEE, 2009] and Verilog [IEEE, 2006] were born as high-level languages designed for programming and driving discrete event circuit simulations, they slowly became standard input formats for synthesis tools. The current practice is to single out a *synthesizable* subset of the language that corresponds to actual circuits. The rest of the language contains useful facilities for driving simulation and testing that do not generally make sense as circuits, such as unbounded loops.

In theory, the synthesizable subsets should be clearly defined and common to all tools. Indeed, the synthesizable part of VHDL has even been standardized [IEEE, 2000]. In practice, the precise subset allowed depends on the synthesis tools. Fortunately, this problem occurs mostly for circuits at a lower abstraction level than ours, such as circuits using tri-state logic.

The circuits generated from machines follow what we ought to call the *computer scientist abstraction*: wires hold zeroes and ones and the final value computed does not depend on physical issues, including timing. This is realistic since our machines are well-behaved, and in particular have no combinatorial feedback loops.

Let us now explain the semantics of VHDL and Verilog in very broad strokes. We use the VHDL terminology since it is the HDL we are the most familiar with. The core of synthesizable VHDL and Verilog consists in sets of *processes* executing concurrently. A process specifies a list of inputs called its *sensitivity list*. Its body is re-executed whenever a variable in its sensitivity list changes. Execution proceeds until a fixpoint is reached. For synthesis purpose, one should distinguish *combinatorial* and *sequential* processes. Combinatorial processes are stateless, only assigning combinations of their inputs to their outputs, while sequential processes contain sequential logic, such as latches and flip-flops. Best practices guides typically recommend that combinatorial processes be sensitive to all their inputs, while sequential processes are only sensitive to a distinguished input, the *clock*. The statements inside a typical sequential process are guarded and execute only on a rising edge of the clock. Processes can be regrouped into modules which can be *instantiated* any number of times.

From machines to HDLs We can now sketch an implementation of machines in terms of processes. The description below is informal since no commonly-agreed formal syntax nor semantics of VHDL or Verilog exists as far as we know. We should also stress that it has not been implemented yet.

The type systems of VHDL and Verilog are well-adapted to the description of finite-state values. They include integers of arbitrary (finite) precision, including booleans. Bounded lists can be represented as bounded arrays, which are natively supported, together with an integer describing the number of valid elements in the array. Products are not supported since HDLs are n-ary rather than unary, but the translation from a unary language with product to an n-ary one is routine, if tedious.

The general implementation scheme is to translate any machine of type to a combinatorial process and a type definition describing its state. In addition to the original inputs of the machine, this process receives its current state, and in addition to the original outputs, the process also produces the next state. Each machine should also define an initial state of the proper type. Let us stress that the state type may be abstract, as it is never manipulated by any other machine.

Looking back at Figure 4.5, it is clear that most primitive combinatorial machines have nothing special: they are readily programmed in any HDL, respecting the above mentioned type scheme. Since they feature variables rather than raw, "point-free" combinators, VHDL and Verilog are actually higher-level than our machine language. They have the exchange and weakening rules built-in. This makes the translation of data movement machines trivial. In particular, permutations do not have to be reduced to swap machines. Feedback machines are also programmed naturally using recursive equations. The only case remaining is that of replication machines.

We assume that the replication machines have been simplified. The question is thus to

implement the gating and turbo machines. The implementation of $\texttt{mturbo}_n(m)$ is immediate: generate n copies of the process implementing m and plug the "next state" output of the i-th copy into the "current state" input of the $i+1$-th copy, if any. The implementation of $\texttt{mgate}(m)$ is hardly more complex. When no reaction of m has to be performed, simply copy the current state to the next state, ignoring the outputs of m. This concludes the implementation of machines.

Finally, the generated combinatorial process has to be turned into a sequential process at some point. This does not formally take part of the compilation process but happens when the programmer wants to actually process inputs, like the generation of a simulation function in existing Lustre-like languages. The only thing to do is simply to relate the current and next state signal. Concretely, one simply needs a sequential process connecting the next state output to the current state input through a register initialized with the initial state defined by the machine.

4.6 Bibliographic Notes

We conclude this chapter by a brief discussion of inspirations and related work.

Compiling synchronous functional languages The compilation of synchronous languages has received a lot of attention. We restrict ourselves to synchronous functional languages in the vein of Lustre for now. Let us remark that the circuit translation of other synchronous languages, such as Esterel, often factors through a Lustre-like intermediate representation.

The general strategy looks much like the one exposed in the previous section: one generates a transition function that computes the outputs and next state from the inputs and current state. When this function is implemented in a sequential language such as C, the code has to be scheduled. This is explained in the first part of the PhD thesis of Raymond [1991].

Historically, Lustre programs involve numerous if statements and boolean expressions, and it was perceived that the optimization of the control structure was of crucial importance. In practice, various heuristics were used, including powerful techniques based on binary decision diagrams. The second part of the PhD thesis of Raymond and the paper from Halbwachs et al. [1991] are good entry points.

Even if most compilation work in the Lustre community has dealt with software, some researchers have studied the generation of digital circuits. Rocheteau and Halbwachs [1992] describe a simple translation process from a dialect of Lustre with booleans and arrays to an FPGA. The thesis of Rocheteau [1992] gives additional details and a complete description of the circuit synthesis system Pollux, built on top of the fourth version of Lustre. The translation was a simple, syntactic translation mapping logical operators to boolean gates and `pre` operators to registers. The focus of the work is rather on language and toolchain design aspects, which are blatantly absent from this thesis you are reading.

The work of Caspi and Pouzet [1996] enriched Lustre with a (binary) clock type system which is the ancestor of the one found in this thesis. The code generation aspects for a small first-order fragment were later revisited [Biernacki et al., 2008]. The paper explains in a lucid

manner how clock types can be used to generate if statements, a technique that was already present in the thesis of Raymond. This optimization can be understood as the special case of the compilation of local time scales where the driving clock is binary. In particular, it should be possible to describe this optimization as a source-to-source transformation in a system such as our, easing its correctness proof.

All the works evoked above handle only first-order languages, with the exception of Lucid Synchrone. However, as far as we know all published work on the compilation of Lucid Synchrone assumes the presence of higher-order features in the target language, including the co-iterative characterization of Caspi and Pouzet [1998]. The third version of the Lucid Synchrone offers the command-line switch `-realtime` which rejects programs that do not work within bounded memory, while still accepting some higher-order programs. However, this works by exporting the type of the internal state in the generated OCaml code, which is not modular. Also, we do not know whether the information could be exploited to produce finite-state code, be it hardware or software.

Monoidal categories We have explained how the compilation of linear higher-order functions works by building a linear higher-order strata on top of our first-order machine language, using the feedback machine in a crucial way for higher-order machine composition.

This construction is in fact a very specific instance of a general category-theoretical result discovered by Joyal, Street, and Verity [1996]. This result, the *Int() construction*, builds a *compact-closed category*[1] out of a *traced monoidal* one. Let us explains what this means very briefly. A *monoidal category* is a category in which there is a notion of monoidal product of objects. Like the tensor product of linear algebra, monoidal products do not necessarily admit projections. A *symmetric monoidal category* is a monoidal category in which, roughly speaking, one can swap the components of a monoidal product. A *traced monoidal category* is a symmetric monoidal category endowed with a feedback-like operator, the *trace*. A *compact-closed category* is a monoidal category with additional properties, the most relevant to our purposes being that it is *closed*. This means that it admits internal hom objects, that is, the space of morphisms from an object A to an object B is reflected as an object $A \multimap B$ of the category.

In our case, the traced monoidal category would correspond to the machine language, with the feedback machine providing the trace and the exchange machine providing the symmetry. The compact-closed category would be the linear higher-order macro-machines. Note that this is only an informal intuition and guide, since machines are syntactic objects that do not directly form a category. We will discuss the links with category theory further in Chapter 6.

Remark 25. The idea of considering a traced category of synchronous machines and applying the Int() construction comes from joint work with M. Bagnol. The curious reader will find a precise description of this category in the report [Bagnol and Guatto, 2012], which has in many ways inspired the first sections of the present chapter.

Geometry of Synthesis The use of monoidal closed categories in hardware compilation is not new. In fact, the higher-order aspects of our work were partly inspired by the seminal

[1] Or, more generally, a *tortile category*.

Geometry of Synthesis (GoS) of Ghica and collaborators [Ghica, 2007; Ghica and Smith, 2010, 2011; Ghica et al., 2011].

This line of works starts from the realization that game models for a certain fragment of Algol are finite-state. Thus, it is actually possible to compute and manipulate the denotations, for instance to implement them in hardware, or to model-check them. Later works explore language and type system extensions aimed at improving the expressiveness of the language as well as the performance of the generated circuits.

Like ours, one of the defining features of the Geometry of Synthesis work is its insistence on compositionality and separate compilation. This is usually one of the main characteristics of techniques based on denotational or categorical models compared to other approaches. Apart from this general fact and that both interpreted in a specific closed monoidal categories, the languages are quite different. The input language is a variant of concurrent Algol, a general-purpose imperative language with concurrency. Concurrency is deterministic since the type system enforces the absence of data races. Our language is functional and entirely dedicated to stream processing, with its dedicated clock type system. The compilation technology is also quite different: as the game model the GoS compiler is based on is asynchronous, the computed denotations have either to be implemented as asynchronous circuits, or transformed into synchronous ones. Since asynchronous circuits are incompatible with current synthesis toolchains, the GoS compiler has to apply a synchronizing transformation, the *round abstraction* of Alur and Henzinger [1999]. The compositionality of this transformation is a difficult question [Ghica and Menaa, 2010], and it is probably expensive in practice.

Chapter 5

Extensions

The language considered up to now, μAS, is deliberately simplistic. This makes concepts, constructions and proofs easier to explain and describe in full formal details. Yet, from a programming point of view this is unsatisfactory. Indeed, some programs that were simple to write in previous synchronous functional languages become very convoluted or even impossible to express. This is mostly due to the low expressiveness of the clock type language on the one hand, and to the linearity restrictions of the type system on the other.

This chapter discusses language extensions that remedy these issues, starting from simple features and moving progressively to more invasive ones. We do not repeat the whole development from the previous chapter in each case, but describe how each feature can be typed, its typed semantics, and its compilation to machines in a semi-formal manner. The interpretation of new types or judgments is in general sufficient to understand how the extension integrates with the rest of the language. The general theme is to show how integer clocks and linear types do not clash with features from existing synchronous languages, but rather integrate with them smoothly

This chapter consists in four section, each describing a distinct extension. Section 5.1 introduces a modality qualifying types that can be used more than once, inspired from Bounded Linear Logic [Girard et al., 1992]. This makes it possible to apply functions of any order more than once, as long as this number is statically known. Section 5.2 discusses the addition of functions which can be applied an arbitrary number of times. This correspond to the notion of *node* in Lustre and its offspring. The price to pay is that nodes have to be closed and cannot be abstracted over. Section 5.3 extends the clock type language with clock polymorphism à la Lucid Synchrone [Caspi and Pouzet, 1996]. This has a drastic impact on the type system since clock types may now contain free variables. Section 5.4 considers a more prospective change: the addition of data-dependent clocks, which contain program variables as in Lucid Synchrone. This extension has an even more disruptive effect on the type system than clock polymorphism since it introduces a form of type dependency. We describe its impact and discuss possible points in the design space.

5.1 Bounded Linear Types

We have seen in Chapter 4 how enforcing that every function received or passed is used exactly once leads to a simple, type-directed compilation scheme to first-order code. Remember that the idea is to turn higher-order functions into first-order ones with additional arguments used for orchestrating the exchanges of arguments and results with their functional parameters. For example a second-order function with exactly one argument is compiled to code with one input, corresponding to the result of its functional argument, and two outputs, corresponding to the value passed to its functional argument and to its own result. This explains in retrospect why functional values have to be used exactly once: using a function twice would require having not one but two additional inputs and outputs, and so on for higher usage counts.

In this section, we introduce a new type constructor, the *bounded exponential modality* $!_n\, t$, which characterizes a type t that can be duplicated n times. For example, $!_2\,(t_1 \multimap t_2)$ describes a function from t_1 to t_2 that has to be called twice. The general idea is that a program inhabiting type $!_n\, t$ is like a product of n inhabitants of t, with the caveat that all the components of this product behave in the same way. One can also think of it as n identical copies of the same program of type t. We first revisit the development of Chapters 3 and 4, explaining how to adapt them to the new type constructor. A later part discusses its expressiveness, limitations, and relationship to Bounded Linear Logic.

> In all the sections of this chapter, we follow the convention that rules or premises added to an existing judgment are typeset in grey. So are new or modified judgments. We do not pretend that these changes are modular: in general, properties of the original system do not transfer to the modified one.

5.1.1 Type System and Semantics

This extension affects the type grammar and system of the source language, and has to be accommodated for in the compilation to machines. The untyped semantics remains unchanged. We study its impact on the type system and typed semantics first.

Type system The grammar of types from Chapter 3 is extended with our new type constructor, and the type system with three new judgments. The updated grammar of types is given below and the modification to the typing rules in Figure 5.1.

$$t \quad += \quad !_n\, t \qquad\qquad \text{Bounded exponential type, to be used } n \text{ times}$$

The type system changes are built on top of a new subtyping judgment $\vdash t \leq_\circ t'$ which expresses whether one can transform an inhabitant of t into an inhabitant of t' by folding or unfolding exponential modalities. We call this judgment *linear subtyping* since it concerns linear aspects of the type system. Its first rules express that it is both a preorder and a congruence. The LINARROW rule displays the contravariance characteristic of arrow subtyping.

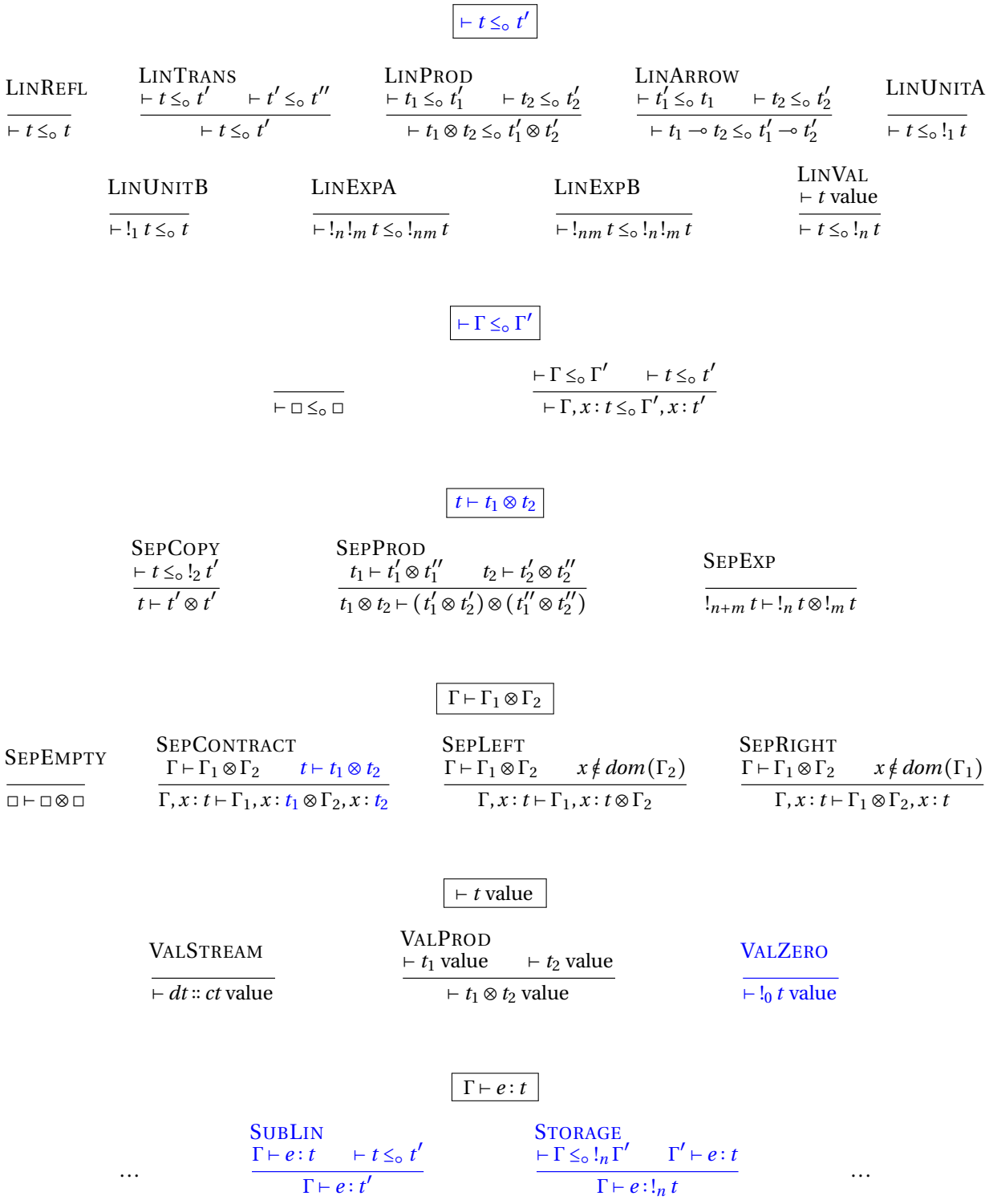


Figure 5.1: Bounded exponential modality - additions and modifications to the type system

The LINUNITA and LINUNITB rules express that t and $!_1\, t$ are isomorphic, since the set of inhabitants of t and the set of a one component products of inhabitants of t are morally equal. The LINEXPA and LINEXPB rules express that $!_n !_m\, t$ and $!_{nm}\, t$ are isomorphic, recalling the identity $x^{n^m} = x^{nm}$ of high-school algebra. Finally, rule LINVAL is the most interesting one. It expresses that since any type t which is a value can be duplicated at will, it can be used an arbitrary number of times. The judgment $\vdash \Gamma \leq_\circ \Gamma'$ lifts the linear subtyping rule to contexts in the expected way.

Our intuitions suggest that should be able to split any inhabitant of $!_{n+m}\, t$ into one inhabitant of $!_n\, t$ and one of $!_m\, t$, as in the identity $x^{n+m} = x^n \times x^m$. This idea enters the type system through the new *type splitting* judgment $t \vdash t_1 \otimes t_2$. The splitting judgment expresses that an inhabitant of t can be split into an inhabitant of t_1 and one of t_2. We say that such a type t is *splittable*. This inhabitants behave in the same way but impose different usage constraints and hence have different types. Rule SEPCOPY splits inhabitants of $!_2\, t$ into two inhabitants of t. Together with LINVAL this rule also ensures that all value types are splittable. Rule SEPPROD is the congruence-closure of this judgment for products. We do not have a similar rule for arrows since the shape of this judgment makes it impossible to express contravariance. Finally, rule SEPEXP implements the expected identity. This concludes the description of the new judgments.

The original type system presented in Chapter 3 relies on the value judgment $\vdash t$ value to distinguish which types can be erased or duplicated. The classification of types was brutal then: a value type can be duplicated *at will* while other types simply could not be duplicated at all. With bounded exponentials, things get slightly better. Indeed, one can view the judgment $t \vdash t_1 \otimes t_2$ as expressing that t can be duplicated *once*, obtaining copies with types t_1 and t_2 which may or may not be splittable themselves. To take advantage of this fact we should alter the context splitting judgment to allow contraction not only on values but also on splittable types. This modification affects rule SEPCONTRACT, with the value judgment replaced with the type splitting one. The new version of the rule is able to express how the uses of a variable x of type $!_n\, t$ are to be shared between the two contexts. In extreme cases, Γ_1 may obtain all the n uses and Γ_2 none of them. Still, the form of the rule guarantees that Γ_2 contains a binding for x of type $!_0\, t$, which describes a unusable inhabitant of t. This type can be freely erased, and should thus be a value. We modify the value judgment to include rule VALZERO.

The main typing judgment is where our efforts pay off. We add two new typing rules. The first one, SUBLIN, is the usual subsumption rule applied to the linear subtyping judgment $\vdash t \leq_\circ t'$. The second one, STORAGE, is the only rule capable of introducing new non-trivial exponentials. It relies on the $!_n\, \Gamma$ notation, which abbreviates a context where all the types in Γ have been annotated with $!_n\, \_$.

$$\begin{array}{lcl} !_n\, \square & = & \square \\ !_n\, \Gamma, x : t & = & !_n\, \Gamma, x : !_n\, t \end{array}$$

The STORAGE rule expresses how a computation e can be made reusable. If one sees types and contexts as resources, the rule can read as follows: if the program e can produce the resource t consuming the resources in Γ', and that the resource Γ can be consumed to obtain n resources Γ', then from Γ' one can obtain n resources t by running e on each of the n resources.

Remark that this rule implies that a closed expression, where Γ is empty, can be replicated an arbitrary number of times.

Typed semantics The linear aspects of the type system played in minimal rôle in the typed semantics of Chapter 3. This comes from the fact that the linear arrow and tensor product type constructors are interpreted as ordinary cartesian products and continuous functions. Bounded exponentials are similarly transparent at this level: t and $!_n\, t$ denote the same domain. Thus, we formally add $\mathbf{S}[\![!_n\, t]\!] = \mathbf{S}[\![t]\!]$ to the interpretation of types.

The typed semantics needs to interpret all the rules in the typing judgment $\Gamma \vdash e : t$, as well as the context splitting judgment $\Gamma \vdash \Gamma_1 : \Gamma_2$. Since the rule CONTRACT in the latter judgment has been modified, we need to change its old interpretation, recalled below.

$$\mathbf{S}[\![\Gamma, x : t \vdash \Gamma_1, x : t \otimes \Gamma_2, x : t]\!] \in \mathbf{S}[\![\Gamma]\!] \times \mathbf{S}[\![t]\!] \Rightarrow_c \mathbf{S}[\![\Gamma_1]\!] \times \mathbf{S}[\![t]\!] \times \mathbf{S}[\![\Gamma_2]\!] \times \mathbf{S}[\![t]\!]$$
$$\mathbf{S}[\![\Gamma, x : t \vdash \Gamma_1, x : t \otimes \Gamma_2, x : t]\!] = \lambda(\gamma, v).((\gamma_1, v), (\gamma_2, v)) \text{ where } (\gamma_1, \gamma_2) = \mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!]\ \gamma$$

In the new CONTRACT the conclusion is different: the type t on the right of the turnstile has become two distinct types t_1 and t_2. For the new interpretation to be well-defined, it should belong to the following domain.

$$\mathbf{S}[\![\Gamma, x : t \vdash \Gamma_1, x : t_1 \otimes \Gamma_2, x : t_2]\!] \quad \in \quad \mathbf{S}[\![\Gamma]\!] \times \mathbf{S}[\![t]\!] \Rightarrow_c \mathbf{S}[\![\Gamma_1]\!] \times \mathbf{S}[\![t_1]\!] \times \mathbf{S}[\![\Gamma_2]\!] \times \mathbf{S}[\![t_2]\!]$$

Now, given the trivial interpretation of exponentials and the rules of the type splitting judgment, we see that here the domains $\mathbf{S}[\![t_1]\!]$ and $\mathbf{S}[\![t_2]\!]$ are actually the same. It would thus be tempting to keep the interpretation of the contraction judgment unchanged. We prefer making it well-formed by defining trivial interpretations for the linear subtyping and type splitting judgment. In addition, to insist on the fact that these interpretation only perform trivial work, we interpret them not as continuous functions but as continuous isomorphisms. The judgment $t \vdash t_1 \otimes t_2$ actually has two interpretations, each corresponding to the isomorphism between $\mathbf{S}[\![t]\!]$ and $\mathbf{S}[\![t_1]\!]$ or $\mathbf{S}[\![t_2]\!]$, respectively.

$$\mathbf{S}[\![\vdash t \leq_\circ t']\!] \in \mathbf{S}[\![t]\!] \cong \mathbf{S}[\![t']\!] \qquad \mathbf{S}[\![t \vdash t_1 \otimes t_2]\!]_L \in \mathbf{S}[\![t]\!] \cong \mathbf{S}[\![t_1]\!] \qquad \mathbf{S}[\![t \vdash t_1 \otimes t_2]\!]_R \in \mathbf{S}[\![t]\!] \cong \mathbf{S}[\![t_2]\!]$$

We omit the uninteresting definitions of these interpretations. They mostly consist in straightforward compositions of isomorphisms, with the axioms interpreted as identities. The case of rule CONTRACT can now be handled cleanly.

$$\begin{aligned}\mathbf{S}[\![\Gamma, x : t \vdash \Gamma_1, x : t_1 \otimes \Gamma_2, x : t_2]\!] = \lambda(\gamma, v).((\gamma_1, v_1), (\gamma_2, v_2)) \text{ where } (\gamma_1, \gamma_2) &= \mathbf{S}[\![\Gamma \vdash \Gamma_1 \otimes \Gamma_2]\!]\ \gamma \\ v_1 &= \mathbf{S}[\![t \vdash t_1 \otimes t_2]\!]_L\ v \\ v_2 &= \mathbf{S}[\![t \vdash t_1 \otimes t_2]\!]_R\ v\end{aligned}$$

Finally, the new typing rules also have simple interpretations using the semantics of the linear subtyping judgment. We lift the interpretation of linear subtyping to contexts as usual.

$$\begin{aligned}\mathbf{S}[\![\Gamma \vdash e : t]\!] &\in \mathbf{S}[\![\Gamma]\!] \Rightarrow_c \mathbf{S}[\![t]\!] \\ &\cdots \\ \mathbf{S}[\![\Gamma \vdash e : t']\!] &= \mathbf{S}[\![\vdash t \leq_\circ t']\!] \circ \mathbf{S}[\![\Gamma \vdash e : t]\!] \\ \mathbf{S}[\![\Gamma \vdash e : !_n\, t']\!] &= \mathbf{S}[\![\Gamma' \vdash e : t]\!] \circ \mathbf{S}[\![\vdash \Gamma \leq_\circ \Gamma']\!]\end{aligned}$$

5.1.2 The Translation

More interesting things happen during the compilation process, since in contrast with the synchronous semantics this process relies essentially on linearity. The list type constructor $mt[n]$ of the machine language is a good match for interpreting bounded exponentials $!_n\, t$. Programs of type $!_n\, t$ will be translated to machines computing elements of type $(\!|t|\!)[n]$. Note that the latter type describes lists containing *at most n* elements, but that the lists arising from the compilation of programs in $!_n\, t$ will always contain *exactly n* elements. We recap the translation of types below.

$$\begin{array}{rcl}(\!|dt :: ct|\!) & = & \mathtt{unit} \multimap (\!|dt|\!)[\lceil ct \rceil] \\ (\!|t_1 \otimes t_2|\!) & = & (\!|t_1|\!) \boxtimes (\!|t_2|\!) \\ (\!|t_1 \multimap t_2|\!) & = & (\!|t_1|\!) \boxminus (\!|t_2|\!) \\ (\!|!_n\, t|\!) & = & (\!|t|\!)[n]\end{array}$$

New machines In order to compile bounded exponentials, we need to translate the linear subtyping and type splitting judgments of Figure 5.1. In particular, we need machines to implement the two isomorphisms $!_{nm}\, t \cong !_n !_m\, t$ and $!_{n+m}\, t \cong !_n\, t \otimes !_m\, t$ derivable from the system. We already have such machines for the first isomorphism, in the guise of gathering and scattering. For the second isomorphism, we enrich the target language with the new primitive machines *concatenation* and *decatenation* machines $\mathtt{mconcat}^{mt}_{n_1,n_2}$ and $\mathtt{mdecat}^{mt}_{n_1,n_2}$, and define the corresponding higher-order macros. We also need $\mathtt{mhoturbo}^{n}_{mtm_1,mtm_2}(m)$, the higher-order analogue to the turbo machine $\mathtt{mturbo}_n(m)$, as well as $\mathtt{mhodupl}^{n}_{mtm}(m_1,m_2)$ which duplicates its higher-order input n-times. All these machines have straightforward HDL implementations.

The new machines are handled in Figure 5.2. Figure 5.2 (a) gives their typing rules; the named rules are for primitive machines and the anonymous rules can be derived from the definition of the macro-machines. The latter are given in Figure 5.2 (c). Figure 5.2 (b) gives the reaction rules for the concatenation and decatenation. The reaction rules rely on the predicate $concat(xl, yl, zl)$ which is true whenever zl is the concatenation of xl and yl.

$$\begin{array}{rl} & concat(xl, yl, zl) \\ = & \forall 0 \leq i < |xl|, zl[i] = xl[i] \text{ and } \forall |xl| \leq i < |yl|, zl[|xl| + i] = yl[i] \text{ and } |xl| + |yl| = |zl| \end{array}$$

Note that in contrast with scattering, the decatenation machine does not receive and additional argument expressing how the input is split. This is sufficient for our purpose since the size of all its input and output lists will reach their bounds.

Translation Figure 5.3 describes the additions and modifications to the translation of Chapter 4 required to accommodate for bounded exponentials.

Figure 5.3 (a) gives the translation of linear subtyping. The first four cases are straightforward analogue to similar rules in Chapter 4. The next four rules, LINUNITA/LINUNITB and LINEXPA/LINEXPB, express the type isomorphisms $t \cong !_1\, t$ and $!_n !_m\, t \cong !_{nm}\, t$. These isomorphisms are implemented at the machine level by the higher-order versions of the wrapping/unwrapping and scattering/gathering couples. The LINVAL rule expresses that one can

$$\textsc{MConcat}\quad \frac{}{\vdash \mathtt{mconcat}^{mt}_{n_1,n_2} : mt[n_1] \times mt[n_2] \multimap mt[n_1+n_2]} \qquad \textsc{MDecat}\quad \frac{}{\vdash \mathtt{mdecat}^{mt}_{n_1,n_2} : mt[n_1+n_2] \multimap mt[n_1] \times mt[n_2]}$$

$$\frac{}{\vdash \mathtt{mhoconcat}^{mtm}_{n_1,n_2} : mtm[n_1] \boxtimes mtm[n_2] \boxminus mtm[n_1+n_2]}$$

$$\frac{}{\vdash \mathtt{mhodecat}^{mtm}_{n_1,n_2} : mtm[n_1+n_2] \boxminus mtm[n_1] \boxtimes mtm[n_2]}$$

$$\frac{\vdash m : mtm_1 \boxminus mtm_2}{\vdash \mathtt{mhoturbo}^{n}_{mtm_1,mtm_2}(m) : mtm_1[n] \boxminus mtm_2[n]} \qquad \frac{\vdash m_1 : mtm^{*} \qquad \vdash m_2 : mtm \boxminus mtm \boxtimes mtm}{\vdash \mathtt{mhodupl}^{n}_{mtm}(m_1,m_2) : mtm \boxminus mtm[n]}$$

(a) - Typing rules, primitive and derived

$$\frac{concat(xl,yl,zl)}{\mathtt{mconcat}^{mt}_{n_1,n_2}/(xl,yl) \to \mathtt{mconcat}^{mt}_{n_1,n_2}/zl} \qquad \frac{concat(yl,zl,xl)}{\mathtt{mdecat}^{mt}_{n_1,n_2}/xl \to \mathtt{mdecat}^{mt}_{n_1,n_2}/(yl,zl)}$$

(b) - Reaction rules for new primitive macro-machines

$$\begin{array}{lcl}
\mathtt{mhoconcat}^{mtm}_{n_1,n_2} & \stackrel{\text{def}}{=} & \mathtt{mdecat}^{mtm^{+}}_{n_1,n_2} \parallel \mathtt{mconcat}^{mtm^{-}}_{n_1,n_2} \\
\mathtt{mhodecat}^{mtm}_{n_1,n_2} & \stackrel{\text{def}}{=} & \mathtt{mconcat}^{mtm^{+}}_{n_1,n_2} \parallel \mathtt{mdecat}^{mtm^{-}}_{n_1,n_2} \\
\mathtt{mhoturbo}^{n}_{mtm_1,mtm_2}(m) & \stackrel{\text{def}}{=} & \mathtt{munzip}^{n}_{mtm_1^{-},mtm_2^{+}} \bullet \mathtt{mturbo}_{n}(m) \bullet \mathtt{mzip}^{n}_{mtm_1^{+},mtm_2^{-}} \\
\mathtt{mhodupl}^{0}_{mtm}(m_1,m_2) & \stackrel{\text{def}}{=} & m_1 \parallel (\mathtt{mconst}_{[]} \bullet \mathtt{mforg}_{mtm^{+}[0]}) \\
\mathtt{mhodupl}^{1+n}_{mtm}(m_1,m_2) & \stackrel{\text{def}}{=} & \mathtt{mhoconcat}^{mtm}_{n,1} \boxdot (\mathtt{mhodupl}^{n}_{mtm}(m_1,m_2) \boxtimes \mathtt{mhowrap}_{mtm}) \boxdot m_2
\end{array}$$

(c) - Expansions of the new macro-machines

Figure 5.2: Bounded exponential modality - machines and macro-machines

$$
\begin{array}{lcl}
(\!|\vdash t \leq_\circ t'|\!) & \in & \mathcal{M}_{(\!|t|\!) \boxminus (\!|t'|\!)} \\
(\!|\vdash t \leq_\circ t|\!) & = & \mathtt{mhoid}_{(\!|t|\!)} \\
(\!|\vdash t \leq_\circ t'|\!) & = & (\!|\vdash t'' \leq_\circ t'|\!) \boxdot (\!|\vdash t \leq_\circ t''|\!) \\
(\!|\vdash t_1 \otimes t_2 \leq_\circ t_1' \otimes t_2'|\!) & = & (\!|\vdash t_1 \leq_\circ t_1'|\!) \boxtimes (\!|\vdash t_2 \leq_\circ t_2'|\!) \\
(\!|\vdash t_1 \multimap t_2 \leq_\circ t_1' \multimap t_2'|\!) & = & \mathtt{minterpose}((\!|\vdash t_1' \leq_\circ t_1|\!), (\!|\vdash t_2 \leq_\circ t_2'|\!)) \\
(\!|\vdash t \leq_\circ !_1\, t|\!) & = & \mathtt{mhowrap}_{(\!|t|\!)} \\
(\!|\vdash !_1\, t \leq_\circ t|\!) & = & \mathtt{mhounwrap}_{(\!|t|\!)} \\
(\!|\vdash !_{nm}\, t \leq_\circ !_n !_m\, t|\!) & = & \mathtt{mhoscatt}^{(\!|t|\!)}_{n,m}(\mathtt{mconst}_n, \mathtt{mconst}_m) \\
(\!|\vdash !_n !_m\, t \leq_\circ !_{nm}\, t|\!) & = & \mathtt{mhogath}^{(\!|t|\!)}_{n,m}(\mathtt{mconst}_n, \mathtt{mconst}_m) \\
(\!|\vdash t \leq_\circ !_n\, t|\!) & = & \mathtt{mhodupl}^{n}_{(\!|t|\!)}((\!|t|\!)_E, (\!|t|\!)_D)
\end{array}
$$

(a) - Interpretation of the subtyping judgment

$$
\begin{array}{lcl}
(\!|t \vdash t_1 \otimes t_2|\!) & \in & \mathcal{M}_{(\!|t|\!) \boxminus (\!|t_1|\!) \boxtimes (\!|t_2|\!)} \\
(\!|t \vdash t' \otimes t'|\!) & = & (\mathtt{mhounwrap}_{(\!|t'|\!)} \boxtimes \mathtt{mhounwrap}_{(\!|t'|\!)}) \boxdot \mathtt{mhodecat}^{(\!|t'|\!)}_{1,1} \\
 & & \boxdot (\!|\vdash t \leq_\circ !_2\, t'|\!) \\
(\!|t_1 \otimes t_2 \vdash t_1' \otimes t_2' \otimes t_1'' \otimes t_2''|\!) & = & \langle (\!|t_1'|\!), (\!|t_1''|\!), (\!|t_2'|\!), (\!|t_2''|\!) \mapsto 0,2,1,3 \rangle_{ho} \\
 & & \boxdot ((\!|t_1 \vdash t_1' \otimes t_1''|\!) \boxtimes (\!|t_2 \vdash t_2' \otimes t_2''|\!)) \\
(\!|!_{n+m}\, t \vdash !_n\, t \otimes !_m\, t|\!) & = & \mathtt{mhodecat}^{(\!|t|\!)}_{n,m}
\end{array}
$$

(b) - Interpretation of the type splitting judgment

$$
\begin{array}{lcl}
(\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!) & \in & \mathcal{M}_{(\!|\Gamma|\!) \boxminus (\!|\Gamma_1|\!) \boxtimes (\!|\Gamma_2|\!)} \\
 & \cdots & \\
(\!|\Gamma, x : t \vdash \Gamma_1, x : t_1 \otimes \Gamma_2, x : t_2|\!) & = & \langle (\!|\Gamma_1|\!), (\!|\Gamma_2|\!), (\!|t_1|\!), (\!|t_2|\!) \mapsto 0,2,1,3 \rangle_{ho} \\
 & & \boxdot ((\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!) \boxtimes (\!|t \vdash t_1 \otimes t_2|\!))
\end{array}
$$

(c) - Modified interpretation of the context splitting judgment

$$
\begin{array}{lcl}
(\!|\Gamma \vdash e : t|\!) & \in & \mathcal{M}_{(\!|\Gamma|\!) \boxminus (\!|t|\!)} \\
 & \cdots & \\
(\!|\Gamma \vdash e : t'|\!) & = & (\!|\vdash t \leq_\circ t'|\!) \boxdot (\!|\Gamma \vdash e : t|\!) \\
(\!|\Gamma \vdash e : !_n\, t|\!) & = & \mathtt{mhoturbo}^{n}_{(\!|\Gamma'|\!), (\!|t|\!)}((\!|\Gamma' \vdash e : t|\!)) \boxdot \mathit{zip}_\Gamma \boxdot (\!|\vdash \Gamma \leq_\circ !_n \Gamma'|\!) \\
 & & \text{where } \mathit{zip}_\Gamma \quad \in \mathcal{M}_{(\!|!_n \Gamma|\!) \boxminus (\!|\Gamma'|\!)[n]} \\
 & & \qquad \mathit{zip}_\Box \quad = \mathtt{mid}_{\mathtt{unit}} \parallel (\mathtt{mconst}_{[]} \bullet \mathtt{mforg}_{(\!|\Box|\!)^-[0]}) \\
 & & \qquad \mathit{zip}_{\Gamma, x:t} = \mathtt{mhozip}^{n}_{(\!|\Gamma|\!), (\!|t|\!)} \boxdot (\mathit{zip}_\Gamma \boxtimes \mathtt{mhoid}_{(\!|t|\!)[n]})
\end{array}
$$

(d) - Enriched interpretation of the main typing judgment

Figure 5.3: Bounded exponential modality - compilation

turn an inhabitant of $(\!|t|\!)$ into n inhabitants, as long as t is a value. This is done by instantiating the duplication machine $(\!|\vdash t \text{ value}|\!)_D$ n times.

Figure 5.3 (b) gives the translation of the type splitting judgment. For rule SEPCOPY, the translation of the premise $(\!|\vdash t \leq_\circ !_2 t'|\!)$ is a list of size two whose elements are in $(\!|t'|\!)$. We decatenate this list and then unwrap each component to obtain a machine in $(\!|t'|\!) \boxtimes (\!|t'|\!)$, the expected type. The SEPPROD case is straightforward. Rule SEPEXP expresses the forward part of the type isomorphism $!_{n+m}\, t \cong !_n\, t \otimes !_m\, t$, which is implemented by the concatenation machine.

Figure 5.3 (c) gives the modifications to the translation of the context splitting judgment. As expected, disruption is minimal: one simply replaces the duplication machine $(\!|\vdash t \text{ value}|\!)_D$ with the one obtained from $(\!|t \vdash t_1 \otimes t_2|\!)$. Only this machine and the corresponding type structure change; the surrounding code stays untouched.

Figure 5.3 (d) gives the two new cases for the translation of the main typing judgment. The translation of the SUBLIN rule follows the usual pattern of subtyping rules: we compose the compiled premise with the compiled subtyping judgment. Implementing the remaining rule STORAGE is slightly more complex: the judgment $\vdash \Gamma \leq_\circ !_n \Gamma'$ gives us n copies of $(\!|\Gamma'|\!)$ which can be transmitted to the machine $(\!|\Gamma' \vdash e : t|\!)$, replicated n times. This replication is performed using the higher-order turbo machine defined before. However, strictly speaking, this is ill-typed: $(\!|!_n \Gamma|\!)$ is not equal to $(\!|\Gamma|\!)[n]$ but isomorphic to it. The former is a product of lists while the latter is a list of products. The machine zip_Γ, defined by induction over Γ, implements the forward part of this isomorphism by lifting the higher-order zip machine to contexts $!_n \Gamma$.

5.1.3 Discussion

The addition of such as simple form of bounded exponential types is easy in theory. Indeed, most of the development above reuses ingredients that are needed in the base language. In a sense this is unsurprising since from an operational point of view a linear exponential $!_n\, t$ is not very different from the n-component product $t \otimes \cdots \otimes t$ definable in the base language. Still, following Felleisen [1990], we argue that it makes the language more expressive, since the untyped source-to-source translation expressing exponentials is a whole-program one.

The practical impact of bounded exponentials remains to be investigated. First, there is the issue of type inference, as it may be unreasonable to expect programmers to annotate programs with usage counts everywhere. In a language without recursion, it should not be difficult. Second, it is unclear whether exponentials should be presented in their raw form, rather than packaged around other type constructors. We could for instance imagine providing a *reusable function type* $t_1 \multimap_n t_2$ as syntactic sugar for $!_n (t_1 \multimap t_2)$. The higher-level surface language would then be translated to the uniform low-level syntax presented in this section.

5.2 Nodes

Even with the introduction of bounded exponential types as presented in the previous section, our language is still less modular than the very first version of Lustre. Indeed, a Lustre program is a list of top-level declarations of *closed* (first-order) functions called *nodes*. Once it has

been defined a node can be applied any number of times—including zero—in the rest of the program. It is important to note that since Lustre does not have dynamic recursion, the number of applications of a given node in a complete program is always finite. However, in a modular compilation setting, this number is *not* known statically at definition time. Thus, nodes are strictly more expressive than bounded exponentials from a reusability point of view. The price to pay for this is the restriction to closed first-order functions, a limitation we managed to avoid up to now while still modularly enforcing the bounded memory discipline.

In this section we show how μAS can be extended with nodes à la Lustre. This extensions closely follows the original Lustre design. Nodes are global and declared at the top of the program. They suffer from the same expressiveness and suffer from the same limitations. In particular, *nodes cannot be abstracted over* and thus are not first-class entities. A node can still be of any type, including higher-order ones.

The section is organized as follows. First, we revisit the development of the source-level language with the node constructs. In particular, the language acquires a new syntactic category, the one of *complete programs*. Typing judgments have to be modified to accommodate for the new notion, and a new type modality $!_\infty\, t$ is added to the type system. In a second part we explain the compilation of nodes. The machine language must now be able to refer to previously defined named machines. We describe this extension and the corresponding target-level typing changes. The translation from source to this extended language is modular and compile programs to sets of machines. Finally, we discuss the implementation of the extended machine language in an HDL, and its proximity with circuits.

5.2.1 Syntax, Type System, and Semantics

Syntax and type system Nodes are global entities, and in particular cannot be nested into one another. This implies the appearance of a new syntactic category p of complete programs. Figure 5.4 (a) gives its grammar and Figure 5.4 (b) its untyped semantics. The function $\mathbf{K}[\![p]\!]$ computes the denotation of a complete program, which is an environment mapping (node) names to elements of the domain $\mathbb{K}$. The denotation of a new node is computed in the environment holding the denotation of its predecessor in program order.

Let us turn to the type system. We need a way to express the fact that nodes can be freely reused (contracted) or erased (weakened). For this, we introduce a new exponential modality $!_\infty\, t$, denoting a program of linear type t which can be reused as often as one wishes. However, remember that nodes are second-class entities that a function cannot receive or return. This means that this new modality *must only qualify a variable*. Thus, we add a new binding form $x : !_\infty\, t$ to contexts Γ. We also need to express that a program p defines a list of well-typed nodes. Formally such a list is new kind of context Θ that only contains node bindings. From now we say that a context Γ is an *expression context* and that a context Θ is a *node context*. The changes to the grammar are summed up in Figure 5.4 (c).

Figure 5.5 presents the modifications applied to the type system. We rely on two new judgment $\Theta \hookrightarrow \Gamma$ and $\vdash p : \Theta$. The former judgment injects the node context Θ into the more general syntactic category of expression contexts Γ. The latter judgment expresses that the nodes of the program p have the type described in Θ. All the nodes in a program should have

p	::=	$\texttt{nil}$	Empty program
	$\mid$	$p\ \texttt{in}\ x = e$	Node definition

(a) Enriched syntax of expressions (Figure 3.1) and programs

$\mathbf{K}[\![p]\!]$	$\in$	$Env(\mathbb{K})$
$\mathbf{K}[\![\texttt{nil}]\!]$	$=$	$\bot$
$\mathbf{K}[\![p\ \texttt{in}\ x = e]\!]$	$=$	$\sigma[x \mapsto \mathbf{K}[\![e]\!]\ \sigma]$ where $\sigma = \mathbf{K}[\![p]\!]$

(b) Enriched untyped semantics (Figure 3.3)

Γ	+=	$\Gamma, x :!_\infty t$	Node binding
Θ	::=	$\square$	Empty node context
	$\mid$	$\Theta, x :!_\infty t$	Node declaration

(c) Enriched syntax of typing contexts

Figure 5.4: Nodes - syntax and untyped semantics

distinct names, as enforced by the $x \notin dom(\Theta)$ premise of the NODE rule. The new binding form $\Gamma, x :!_\infty t$ needs to be handled in the separation and typing judgments. Conceptually a variable $x :!_\infty t$ is value-like in that it admits weakening and contraction, as reflected by the rules SEPCONTRACTNODE and WEAKENNODE. Rule INST expresses that a program can *instantiate* a node x of type $!_\infty t$, obtaining a value of type t in the context Γ. As in the VAR rule this context should be erasable. The judgment $\vdash \Gamma$ value is extended to express that any node is implicitly erasable or duplicable.

Typed semantics The modifications to the typed semantics are minimal in we only describe them briefly. The exponential modality $!_\infty t$ is transparent in the sense that $\mathbf{S}[\![!_\infty t]\!] = \mathbf{S}[\![t]\!]$, as was already the case for the bounded exponential modality. Formally, we have

$$\begin{array}{lcl} \mathbf{S}[\![\square]\!] & = & \varnothing_\bot \\ \mathbf{S}[\![\Gamma, x : t]\!] & = & \mathbf{S}[\![\Gamma]\!] \times \mathbf{S}[\![t]\!] \\ \mathbf{S}[\![\Gamma, x :!_\infty t]\!] & = & \mathbf{S}[\![\Gamma]\!] \times \mathbf{S}[\![t]\!] \end{array}$$

and a similar interpretation for node contexts Θ.

The interpretations for the rules added to existing judgments are exactly the same as the ones for their pre-existing counterparts; for instance, node weakening works exactly like ordinary weakening. The INST rule is interpreted like the VAR one.

The judgment $\Theta \hookrightarrow \Gamma$ is interpreted as a function from $\mathbf{S}[\![\Theta]\!]$ to $\mathbf{S}[\![\Gamma]\!]$. Furthermore, and while this is not strictly necessary for the development to work, we interpret it not as any function but as an embedding-projection pair $\mathbf{S}[\![\Theta]\!] \lhd \mathbf{S}[\![\Gamma]\!]$ to insist in the fact that it models an injection

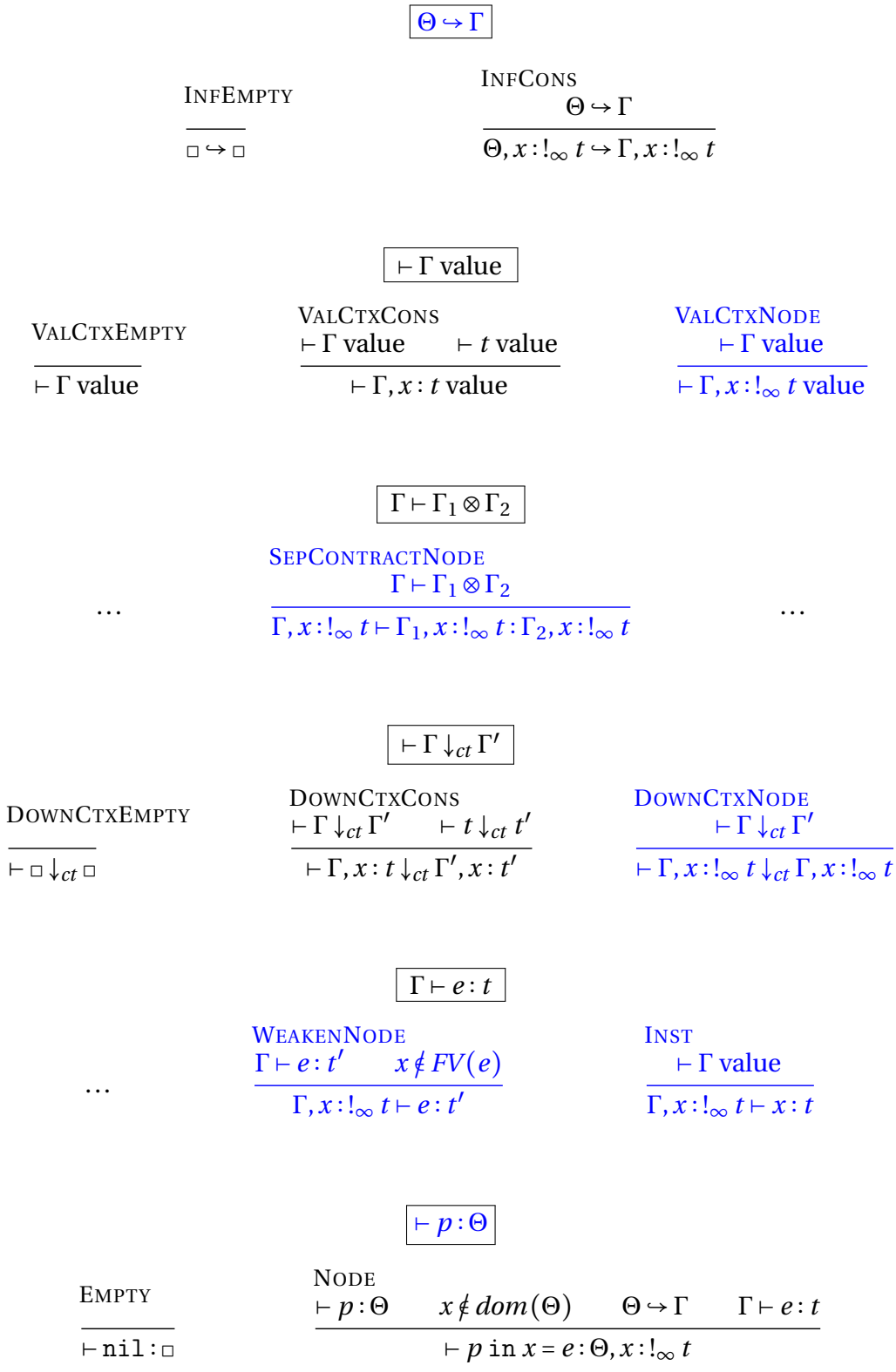


Figure 5.5: Nodes - additions and modifications to the type system

of Θ inside Γ.

$$\begin{array}{lcl}
\mathbf{S}[\![\Theta \hookrightarrow \Gamma]\!] & \in & \mathbf{S}[\![\Theta]\!] \triangleleft \mathbf{S}[\![\Gamma]\!] \\
\mathbf{S}[\![\square \hookrightarrow \square]\!] & = & (id, id) \\
\mathbf{S}[\![\Theta, x : !_\infty\, t \hookrightarrow \Gamma, x : !_\infty\, t]\!] & = & \mathbf{S}[\![\Theta \hookrightarrow \Gamma]\!] \times (id, id)
\end{array}$$

The cartesian product used in the second case above denotes the pairing of embedding-projection pairs defined in Section 3.3.

Finally, the program typing judgment $\vdash p : \Theta$ is interpreted as an element of $\mathbf{S}[\![\Theta]\!]$.

$$\begin{array}{lcl}
\mathbf{S}[\![\vdash p : \Theta]\!] & \in & \mathbf{S}[\![\Theta]\!] \\
\mathbf{S}[\![\vdash \mathtt{nil} : \square]\!] & = & \bot \\
\mathbf{S}[\![\vdash p \ \mathtt{in}\ x = e : \Theta, x : !_\infty\, t]\!] & = & (\theta, \mathbf{S}[\![\Gamma \vdash e : t]\!]\ (\mathbf{S}[\![\Theta \hookrightarrow \Gamma]\!]\ \theta)) \text{ where } \theta = \mathbf{S}[\![p]\!]
\end{array}$$

Metatheory We have to say a word about the proof of totality given in Chapter 3, and in particular Lemma 7. Indeed, rule DOWNCTXNODE might look a bit surprising at first: how can we not change the type of x when entering a local time scale? Remember that the context Γ in $\Gamma \vdash e : t$ describes the view that e has of the surrounding context. But since a binding $x : !_\infty\, t$ describes a node to be instantiated inside e, x is not really defined *outside* of e and thus does not belong to the external world. A more formal version of this fact is that, while t and $!_\infty\, t$ are interpreted by the same domains, their realizers are totally different. The idea is that since elements of $(\!|!_\infty\, t|\!)$ are closed and have, in a sense, already been completely defined, they realize t for all n instead of just for the current one. This leads to the following definition which can be used to reprove Lemma 7.

$$\begin{array}{rcl}
\Downarrow_n^\Gamma & \subseteq & \mathbf{S}[\![\Gamma]\!] \\
\bot \Downarrow_n^\square & & \\
(\gamma, v) \Downarrow_n^{\Gamma, x:t} & \Leftrightarrow & \gamma \Downarrow_n^\Gamma \wedge v \Downarrow_n^t \\
(\gamma, v) \Downarrow_n^{\Gamma, x:!_\infty\, t} & \Leftrightarrow & \gamma \Downarrow_n^\Gamma \wedge \forall n' \in \mathbb{N}, v \Downarrow_{n'}^t
\end{array}$$

The same idea leads to the following set of realizers for whole-programs. Notice that, in contrast with the ones used in contexts and (non-node) types, they are not step-indexed.

$$\begin{array}{rcl}
\Downarrow_\infty^\Theta & \subseteq & \mathbf{S}[\![\Theta]\!] \\
\bot \Downarrow_\infty^\square & & \\
(\theta, v) \Downarrow_\infty^{\Gamma, x:!_\infty\, t} & \Leftrightarrow & \gamma \Downarrow_\infty^\Gamma \wedge \forall n \in \mathbb{N}, v \Downarrow_n^t
\end{array}$$

5.2.2 The Translation

The synchronous semantics gives the same interpretation to the VAR and INST rules. Their compilation, however, is very different. Let us discuss the design of our translation and the rationale for our extensions to the language of machines.

In Lustre nodes are instantiated at each call point, meaning that in hardware each distinct call generates a distinct copy of callee node in the caller node. In software the code for the transition function of the callee node does not have to be copied, but each call still reserves the space for a copy for the internal state of the callee node inside the caller node's own state.

To rephrase this mechanism in our parlance, each call to the node x should *ultimately* give rise to a new copy of the machine $(\!|e|\!)$ generated from the body e of x. Thus, the most direct solution for compiling nodes seems to be a form of inlining: the compilation of expressions takes as input the body of all previously defined nodes and each INST rule for a node x looks for the compiled body of x in this environment. In this scheme the compilation function would have the type

$$(\!|\Gamma \vdash e : t|\!) \quad : \quad \mathcal{M}_{(\!|\Gamma|\!)_N} \to \mathcal{M}_{(\!|\Gamma|\!) \boxminus (\!|t|\!)}$$

with $(\!|\Gamma|\!)_N$ a description of the nodes appearing in Γ and $(\!|\Gamma|\!)$ extended to ignore node bindings.

The trouble with the above scheme is that it breaks separate compilation. To obtain the final object we are interested in, a machine in $(\!|\Gamma|\!) \boxminus (\!|t|\!)$, the interpretation $(\!|\Gamma \vdash e : t|\!)$ must be applied to an environment containing the body of the nodes in Γ. In other words, the compiled code does not simply depend on the types of the nodes present in Γ, but also on their definitions. In particular, an expression must be recompiled when the definitions of the node it calls change, *even if their types do not.*

On the one hand, one may argue that this phenomenon is, in a sense, unavoidable. Circuits are physical objects and a stateful node that is instantiated n-times must exist as n distinct sub-circuits in the final physical piece of matter, barring the use of external memory. Thus, some sort of inlining process has to happen at some point. This phenomenon is avoided in software only through the use of dynamic memory allocation and pointer indirections, which are mandatory to compile Lustre in a modular way.

On the other hand, from a practical point of view the obstacles to separate compilation are less formidable than what we just described. Remember that the compilation process described in Chapter 4 translates a well-typed program to machines, which are then implemented by a direct translation to an HDL, which is finally processed by a specific synthesis tool. This last step is not modular in the slightest since the synthesis frontend unfolds the hierarchical HDL description into a flat representation adequate for the technology mapping, place-and-route and other low-level optimization processes.

This suggests an alternative plan. Instead of inlining node bodies during the translation, we enrich the machine language with a simple form of modularity similar to the one in the source language. This is readily translated to any HDL such as Verilog or VHDL. The synthesis tool is responsible for the inlining process, and the extended translation is exactly as modular as the base one. To sum up, the compilation from μAS to the HDL is modular in the sense that it enjoys separate compilation, while the compilation of HDL code to a circuit has never been.

New machines The addition of named machines impacts all the technical development of machines, since they were previously devoid of any form of binding. Machine names belong to the same set V of variables as source names. However, we denote machine names as $l, l', \ldots$ to avoid confusion in the reaction rules of Figure 4.7 where we have used $x, y, \ldots$ to denote values.

The machines and syntactic categories added to the grammar of machine are given in Figure 5.6. A *machine environment* M is a list of machine declarations of the form $l = m$. The *instance machine* $\texttt{minst}_{mtm}(mi)$ holds an *instance* mi. Such an instance is either an indirection to a previously defined machine named l, or a concrete machine m. We say that the latter

m	+=	$\texttt{minst}_{mtm}(mi)$	Named machine
mi	::=	$l \in V$	Indirection
	|	m	Embedded machine
M	::=	$\varnothing$	Empty machine environment
	|	$M, l = m$	Machine definition
I	::=	$\square$	Empty machine typing context
	|	$I, l : mtm$	Named machine binding

Figure 5.6: Nodes - additions and modifications to the machine syntax

machine is *embedded* in the instance. This mechanism is used so that each instance actually gives rise to a *copy* of the machine named l during execution. A *machine context* I is a list of bindings of machine names to machine types.

Figure 5.7 gives the extended machine type system. Typing judgments now depend on a machine context I for typing the free variables appearing in machine instances. This context is handled in an intuitionistic rather than linear fashion: machine names can be freely weakened and contracted. Moreover, since we are interested in the mere fact that a machine is well-typed rather than in its typing derivations, we manipulate contexts less formally than in the source language. For instance, we write $I(l) = mtm$ to express that the rightmost binding for l in I exists and binds l to mtm. In practice, the fact that this bindings is the rightmost does not matter since machine names will be unique.

Figure 5.7 (a) gives the modified typing rules for machines. The new machine context I is added to all previous typing rules in a transparent fashion: it is unused by atomic machines and simply passed along in machine combinators. The weakening rule is unsurprising; free variables are defined as usual. The last two rules type the new instance machines. The named machine l referred to by a machine $\texttt{minst}_{mtm}(l)$ should be present in the context and have type mtm. The machine m embedded into the machine $\texttt{minst}_{mtm}(m)$ should have type mtm.

Figure 5.7 (b) gives the typing judgment $\vdash M : I$ expressing that a machine environment M has the type I. This closely follows the rules for source programs given in Figure 5.5. A machine binding $M, x = m$ types m in the context I obtained by typing M. The resulting type mtm is then added to I. As in the source language, this rule enforces that machine names are actually unique in a program.

The reaction rules are now parametrized by a machine environment M, as shown in Figure 5.7 (c). As for contexts, we write $M(l)$ for the rightmost machine m bound to l in the environment M. This environment is added transparently in preexisting rules, as was the case for typing. Reaction rules for machine instances are the only interesting one. The reaction of a named machine $\texttt{minst}_{mtm}(l)$ looks for l in M and executes it with the current input value x. The resulting machine m' replaces l in the state of the instance machine. This makes sure that each instance correctly remembers the state of the machine it refers to. When the instance machine $\texttt{minst}_{mtm}(m)$ already wraps a machine m, this machine is executed.

Remark 26. The addition of machine contexts and environment alters the statements of the

$$\boxed{I \vdash m : mtm}$$

$$\frac{}{I \vdash \mathtt{mid}_{mt} : mt \multimap mt}\ \textsc{MId} \qquad \cdots \qquad \frac{I \vdash m_1 : mt_2 \multimap mt_3 \qquad I \vdash m_2 : mt_1 \multimap mt_2}{I \vdash m_1 \bullet m_2 : mt_1\, mt_3}\ \textsc{MComp} \qquad \cdots$$

$$\frac{I \vdash m : mtm \qquad l \notin FV(m)}{I, l : mtm \vdash m : mtm}\ \textsc{MWeaken} \qquad \frac{I(l) = mtm}{I \vdash \mathtt{minst}_{mtm}(l) : mtm}\ \textsc{MInstVar} \qquad \frac{I \vdash m : mtm}{I \vdash \mathtt{minst}_{mtm}(m) : mtm}\ \textsc{MInstEmbed}$$

(a) - Extension and modification of the machine typing judgment (Figure 4.6)

$$\boxed{\vdash M : I}$$

$$\frac{}{\vdash \varnothing : \square}\ \textsc{NEmpty} \qquad \frac{\vdash M : I \qquad l \notin dom(I) \qquad I \vdash m : mtm}{\vdash M, l = m : I, l : mtm}\ \textsc{NCons}$$

(b) - Machine environment typing judgment

$$\boxed{M \vdash m/x \to m'/y} \text{ and } \boxed{M \vdash m/x \to_n m'/y}$$

$$\frac{}{M \vdash \mathtt{mid}_{mt}/x \to \mathtt{mid}_{mt}/x} \qquad \cdots \qquad \frac{M \vdash m_1/x \to m_1'/y \qquad M \vdash m_2/y \to m_2'/z}{M \vdash m_2 \bullet m_1/x \to m_2' \bullet m_1'/z} \qquad \cdots$$

$$\frac{M(l) = m \qquad M \vdash m/x \to m'/y}{M \vdash \mathtt{minst}_{mtm}(l)/x \to \mathtt{minst}_{mtm}(m')/y} \qquad \frac{M \vdash m/x \to m'/y}{M \vdash \mathtt{minst}_{mtm}(m)/x \to \mathtt{minst}_{mtm}(m')/y}$$

(c) - Extension and modification of the reaction judgment (Figure 4.7)

Figure 5.7: Nodes - additions and modifications to the machine type system

metatheoretical results of Section 4.4. The subject reduction result now depends on the fact that the machine m of type $I \vdash m : mtm$ executes within an environment M that is itself of type I. Similarly, the skeletal equivalence is now parametrized by an environment M, forming a relation $M \vdash m \equiv m'$. Thus, the finiteness result now states that the set of machines reachable from any given well-typed machine *in a fixed well-typed environment* is finite.

The translation We can now give the translation of the extended source language to the extended machine language. Unsurprisingly, source programs are translated to machine environments, node instances to machine instances, and program contexts to machine contexts.

Arguably, the only difficulty is in the expression of the type of the translation: what should be the type of $(\!|\Gamma \vdash e : t|\!)$? The result must now express that the resulting machine is well-typed in some machine context I which describes the types of the translated nodes which e refers to. These nodes appear in Γ as bindings $x :!_\infty t$. This suggests having two translations for the same typing context Γ. The first one is the usual one, which gives the higher-order type of the generated machine; it simply ignores the node bindings in Γ. On the contrary, the second translation only remembers the node bindings in Γ, translating them to a machine context $(\!|\Gamma|\!)_N$. Program contexts Θ are also translated to machine contexts $(\!|\Theta|\!)$. The three translations are given in Figure 5.8 (a), (b) and (c).

This definition explains why the SEPCONTRACTNODE and VALCTXNODE rules work. The role of this judgment is to characterize how the elements in $(\!|\Gamma|\!)$ can be duplicated or erased. However, since the node bindings $x :!_\infty t$ in Γ do not appear in $(\!|\Gamma|\!)$, these rules have virtually no work to do: no duplication nor erasure machine needs to be generated for $!_\infty t$. This is shown in Figure 5.8 (d) and (e).

We can now give the type of the translation of expressions. Writing $\mathcal{M}_{I \vdash mtm}$ for the set machines inhabiting type mtm in the machine context I, the machine generated from a well-typed expression $\Gamma \vdash e : t$ has type

$$(\!|\Gamma \vdash e : t|\!) \quad \in \quad \mathcal{M}_{(\!|\Gamma|\!)_N \vdash (\!|\Gamma|\!) \boxminus (\!|t|\!)}$$

as expressed in Figure 5.8 (f). The compilation of preexisting rules does not change. Node weakening is a *no-op* since $(\!|\Gamma, x :!_\infty t|\!) = (\!|\Gamma|\!)$. A node instantiation for $x :!_\infty t$ is translated to the machine instance $\mathtt{minst}_{(\!|t|\!)}(x)$. Complete programs are translated to machine environments in the expected structural manner, given in Figure 5.8 (g). We write $\mathcal{M}_{\vdash I}$ for the set of well-typed program environments of type I.

Is the interpretation well-defined? The translation of nodes to machines is much simpler than the translation of bounded exponentials given in Section 5.1. There is, however, a technical point to consider. When we define the result of the interpretation function as belonging to $\mathcal{M}_{(\!|\Gamma|\!)_N \vdash (\!|\Gamma|\!) \boxminus (\!|t|\!)}$ (for example), we implicitly assume that the right-hand sides of the equations in Figure 5.8 (f) actually belongs to this set, that is that the corresponding machines are well-typed. This fact is not always completely immediate, as the following example shows.

Consider the compilation of node declarations in Figure 5.8 (g). Here the compilation of $\Gamma \vdash e : t$ has type $\mathcal{M}_{(\!|\Gamma|\!)_N \vdash (\!|\Gamma|\!) \boxminus (\!|t|\!)}$ while rule NCONS from the target language expects it to have

$$\begin{array}{lcl} (\!|\square|\!) & = & \square \\ (\!|\Theta, x :!_\infty t|\!) & = & (\!|\Theta|\!), x : (\!|t|\!) \end{array}$$

(a) - Translation of program contexts to machine contexts

$$\begin{array}{lcl} (\!|\square|\!)_N & = & \square \\ (\!|\Gamma, x : t|\!)_N & = & (\!|\Gamma|\!)_N \\ (\!|\Gamma, x :!_\infty t|\!)_N & = & (\!|\Gamma|\!)_N, x : (\!|t|\!) \end{array}$$

(b) - Translation of typing contexts to machine contexts

$$\begin{array}{lcl} & \cdots & \\ (\!|\Gamma, x :!_\infty t|\!) & = & (\!|\Gamma|\!) \end{array}$$

(c) - Extended translation of typing contexts to higher-order machine types

$$\begin{array}{lcl} (\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!) & \in & \mathcal{M}_{\square \vdash (\!|\Gamma|\!) \boxminus (\!|\Gamma_1|\!) \boxtimes (\!|\Gamma_2|\!)} \\ & \cdots & \\ (\!|\Gamma, x :!_\infty t \vdash \Gamma_1, x :!_\infty t \otimes \Gamma_2, x :!_\infty t|\!) & = & (\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!) \end{array}$$

(d) - Extended translation of the separation judgment

$$\begin{array}{lcl} (\!|\vdash \Gamma \text{ value}|\!)_E & \in & \mathcal{M}_{\square \vdash (\!|\Gamma|\!)^*} \\ & \cdots & \\ (\!|\vdash \Gamma, x :!_\infty t \text{ value}|\!)_E & = & (\!|\vdash \Gamma \text{ value}|\!)_E \end{array}$$

(e) - Extended erasure translation of the context value judgment

$$\begin{array}{lcl} (\!|\Gamma \vdash e : t|\!) & \in & \mathcal{M}_{(\!|\Gamma|\!)_N \vdash (\!|\Gamma|\!) \boxminus (\!|t|\!)} \\ & \cdots & \\ (\!|\Gamma, x :!_\infty t \vdash e : t'|\!) & = & (\!|\Gamma \vdash e : t'|\!) \\ (\!|\Gamma, x :!_\infty t \vdash x : t|\!) & = & (\!|\vdash \Gamma \text{ value}|\!)_E \Vdash \mathtt{minst}_{(\!|t|\!)}(x) \end{array}$$

(f) - Modified and enriched translation of well-typed expressions

$$\begin{array}{lcl} (\!|\vdash p : \Theta|\!) & \in & \mathcal{M}_{(\!|\Theta|\!)} \\ (\!|\vdash \mathtt{nil} : \square|\!) & = & \varnothing \\ (\!|\vdash p \text{ in } x = e : \Theta, x :!_\infty t|\!) & = & (\!|\vdash p : \Theta|\!), x = (\!|\Gamma \vdash e : t|\!) \end{array}$$

(g) - Translation of programs to machine sets

Figure 5.8: Nodes - compilation

$$
\begin{array}{lcl}
close(\_,\_) & \in & \mathcal{M}_{\vdash I} \times \mathcal{M}_{I \vdash mtm} \to \mathcal{M}_{\square \vdash mtm} \\
close(M, m_1 \bullet m_2) & = & close(M, m_1) \bullet close(M, m_2) \\
close(M, m_1 \Vdash m_2) & = & close(M, m_1) \Vdash close(M, m_2) \\
close(M, \mathtt{mfb}^{mt_3}_{mt_1,mt_2}(m)) & = & \mathtt{mfb}^{mt_3}_{mt_1,mt_2}(close(M, m)) \\
close(M, \mathtt{mrepl}_n(m)) & = & \mathtt{mrepl}_n(close(M, m)) \\
close(M, \mathtt{mgate}(m)) & = & \mathtt{mgate}(close(M, m)) \\
close(M, \mathtt{mturbo}_n(m)) & = & \mathtt{mturbo}_n(close(M, m)) \\
close(M, \mathtt{minst}_{mtm}(l)) & = & close(M, M(l)) \\
close(M, m) & = & m \text{ (remaining cases)}
\end{array}
$$

Figure 5.9: Nodes - closure

the type of $(\!|p|\!)$, that is $(\!|\Theta|\!)$. Is the resulting machine ill-typed and thus the whole interpretation function ill-defined? No, because of the following property, whose proof is immediate.

Property 46. *For any two contexts* Θ *and* Γ*, if* $\Theta \hookrightarrow \Gamma$ *then* $(\!|\Theta|\!) = (\!|\Gamma|\!)_N$.

The same is true for several other cases. The compilation of the NODEWEAKEN source-level rule is well-typed because of the MWEAKEN target rule. Similarly, even if the existing cases where not modified, the reasons for which some of them are now well-typed may have changed. We let the reader checks that all cases actually type-check.

5.2.3 Discussion

HDLs We have already discussed whether it is realistic to add name binding to a language supposed to model finite-state circuits. We argue that in a pragmatic way it is, since the translation to HDLs is not difficult. VHDL and Verilog both offer facilities for reuse in the guise of named *entities* (for VHDL) or *modules* (for Verilog). These modularity units closely correspond to our named machines, as they are closed pieces of code with globally unique names. HDL tools recursively inline all such abstractions before synthesis, leaving a flat netlist.

For the sake of completeness we can describe the inlining process as a source-to-source transformation at the target level. This takes the form of a function $close(M, m)$ that replaces the free variables of the machine $I \vdash m : mtm$ with their bodies taken from $\vdash M : I$, resulting in the machine $\square \vdash close(M, m) : mtm$. The function is defined in Figure 5.9. It traverses the body of m, calling itself recursively when replacing an indirection $\mathtt{minst}_{mtm}(l)$ with the machine $M(l)$. The function terminates on well-typed programs because machine names are unique and definitions non-recursive. As expected, this process requires the bodies of all the nodes instantiated directly or indirectly in m, and is thus a whole-program transformation.

Expressiveness From the expressiveness point of view, the presence of nodes enables new forms of code reuse compared to plain linear types or bounded exponentials. In particular,

$$\dfrac{\dfrac{}{\vdash f : !_\infty\,(2) \multimap (2) \downarrow_{(2)} f : !_\infty\,(2) \multimap (2)}\ \dots \qquad \dfrac{\dfrac{}{f : !_\infty\,(2) \multimap (2), x : (2) \vdash fx : (2)}\ \dots}{f : !_\infty\,(2) \multimap (2) \vdash \mathtt{fun}\, x.\,(fx) : (4) \multimap (4)} \qquad \dfrac{}{\vdash (2) \multimap (2) \uparrow_{(2)} (4) \multimap (4)}\ \dots}{f : !_\infty\,(2) \multimap (2) \vdash \mathtt{fun}\, x.\,(fx) : (4) \multimap (4)}$$

Figure 5.10: Example - Gathering and nodes

nodes ease the use of local time scales. The two examples below illustrate this point. We elide data types, assuming that they are all equal to, say, **bool**.

Example 24. Imagine that we want to write a program that applies a function of type $(2) \multimap (2)$ to some stream of clock type $(4\ 0)$. In other words, we want to build a derivation of conclusion

$$\text{(a)} \qquad \square \vdash \mathtt{fun}\, f.\mathtt{fun}\, x.\,(fx) : ((2) \multimap (2)) \multimap ((4\ 0) \multimap ?)$$

where ? is some unknown clock type. This can be done either through adaptability, since one has $\vdash (4\ 0) <:_0 (2)$, or through a local time scale driven by $(2)^\omega$. In both cases the best output clock, in the sense of latency, is $(2)^\omega$. The resulting code is reusable in the sense that, once it has been put inside a node, one may call it from anywhere with a distinct argument f.

Example 25. Now, consider the following conclusion.

$$\text{(b)} \qquad f : !_\infty\,(2) \multimap (2) \vdash \mathtt{fun}\, x.\,(fx) : (4\ 0) \multimap (2)$$

It is derivable for the same reason the previous one was. However, it is quite distinct: here f refers to a fixed node that was previously defined in the program. Since nodes are not first class entities, we cannot abstract over f to make this program generic. This version is thus less desirable than the previous one.

Example 26. For our final example, we tweak the desired clock types a little bit. Our goal is to write a typing derivation for the following conclusion.

$$\text{(c)} \qquad \square \vdash \mathtt{fun}\, f.\mathtt{fun}\, x.\,(fx) : ((2) \multimap (2)) \multimap ((4) \multimap ?)$$

where ? is once again some unknown clock type. Unfortunately, there is no such derivation. To see why, notice that a natural type of $\mathtt{fun}\, f.\mathtt{fun}\, x.\,(fx)$, which is an η-expanded version of the identity function, is $((1) \multimap (1)) \multimap ((1) \multimap (1))$. Observe how all clock types appearing in this type have the same rate. Unfortunately, the only operations that we have at our disposal to transform this type into the expected one are buffering and local time scales. Both preserve the relative rates of all the clock types appearing in the type being adapted or gathered. Thus it is not possible to find a derivation having the proper conclusion.

Example 27. In contrast with the previous one, the following conclusion is derivable.

$$\text{(d)} \qquad f : !_\infty\,(2) \multimap (2) \vdash \mathtt{fun}\, x.\,(fx) : (4) \multimap (4)$$

A derivation is given in Figure 5.10. It relies crucially on rule DOWNCTXNODE, which makes it possible not to link the type of the node, which appears in the context, with the clock type

driving the local time scale. We see that in fact this works for an arbitrary local time scale, obtaining arbitrary high throughput! In this case the version using a node, while still not generic over f, is well-typed while its higher-order analogue plainly does not exist.

Those examples show one way in which nodes and linear higher-order functions are very different. The problem with (c), stated in a down-to-earth fashion, is that to obtain the desired type we need to replicate f twice, but that the function has been defined elsewhere and thus its body is unavailable. This is not the case when f is a node, as by definition node bodies are always available to play with. The large difference between the code generated for (a) and (b) or (d) reflects this fact. In (b) and (d) the call to f is actually a node instantiation, resulting in a copy of the node body at the instantiation point. Since this body is inside a machine which it itself replicated twice because of the local time scale, at the end of the day three distinct copies of the node body are present in the generated circuit. In contrast, the code generated for (a) does not include any copy of the body of f and is thus completely modular. The price to pay for this modularity is the simplicity of the corresponding time transform, which only corresponds to buffering.

In a sense, none of the constructions above feel satisfactory. Local time scales are expressive when used in conjunction with nodes and more generally closed expressions, but using nodes impede reuse and genericity. Linear higher-order functions on the other hand are very modular but can only implement relatively trivial time transforms. The next section introduces polymorphic clocks types, which can be used to describe generic code in a modular fashion.

5.3 Clock Polymorphism

It is clear that for some programs the types assigned by the system described in Chapter 3 are too specific. Consider the identity function `fun` $x.x$. It may be assigned any type of the form $dt :: ct \multimap dt :: ct$, with dt and ct fixed data and clock types, respectively. A traditional solution for expressing this genericity over dt is the introduction of *data type polymorphism*, as featured in the polymorphic lambda-calculus of Girard and Reynolds. In this section, we are rather interested in genericity over ct and thus in what we call *clock type polymorphism*. The immediate idea for providing this form of polymorphism is to introduce *universal clock quantification*, written $\forall \alpha.t$. The identity function then receives the type $\forall \alpha.dt :: \alpha \multimap dt :: \alpha$ for any fixed dt. With this extension, a clock type no longer denotes a single clock but rather a set of clocks, or more precisely a function from (the interpretation of) its free variables to the domain of clocks.

Unfortunately, clock quantification as presented above, while appealing, does not work directly in our setting. The problem is that since we have no information on the bound variable α, we are no longer able to bound the integers appearing in the clocks denoted by a universally quantified type. Having a bound on the integers appearing in each clock is needed for two reasons. First, as seen in Chapter 4, bounds describe the maximum size of lists in the generated code. Second, it is also important in the type system. Remember that to check the productivity of fixpoints, we need to check whether two clock types are 1-adaptable. In the existing clock type language, this is simple, since one can compute the exact unique

clock denoted by a clock. In the richer polymorphic language one has to settle for syntactic approximations, as shown in the example below.

Example 28. Imagine that we want to find a general rule for checking whether $\vdash \alpha$ **on** $p_1 <:_k \alpha$ **on** p_2, with p_1 and p_2 arbitrary ultimately binary clock. This rule should ensure that for any clock w giving the value of α at runtime, one has $w\ on\ p_1 <:_k w\ on\ p_2$. In Chapter 2, we have seen that if $p_1 <:_k p_2$, then $w\ on\ p_1 <:_0 w\ on\ p_2$. However, we have $(1) <:_1 0(1)$ yet

$$\begin{array}{rrcl} & (2)^\omega\ on\ (1)^\omega & <:_1 & (2)^\omega\ on\ 0(1)^\omega \\ \Leftrightarrow & (2)^\omega & <:_1 & 1(2)^\omega \end{array}$$

does *not* hold. Similar examples can be devised for arbitrary values of w, p_1, p_2 and $k \geq 1$. This implies that, without a bound on the maximum value appearing in the clocks w being denoted by α, it is impossible to find a sound rule for $\vdash \alpha$ **on** $p_1 <:_k \alpha$ **on** p_2 with $k \geq 1$.

To avoid these problems, we adopt a restricted form of clock type polymorphism that we call *bounded clock polymorphism*, in analogy with the bounded polymorphism found in type systems that exhibit both parametric polymorphism and subtyping [Pierce, 2002]. Here the bounds are simply natural numbers. The type $\forall(\alpha \leq n).t$, with α possibly free in t, has as inhabitants programs that work for any clock type *ct* instantiating α, such that the clocks w denoted by *ct* feature integers no larger than n.

Remark 27. Polymorphic clock types were introduced by Caspi and Pouzet [1996] in Lucid Synchrone. They were later adopted in the SCADE6 [Esterel Technologies, 2015] and Heptagon [Delaval et al., 2012] dialects of Lustre, as well as in Lucy-n [Mandel et al., 2010; Plateau, 2010]. Clock polymorphism works essentially in the same way in all these languages, a small difference being that Heptagon and SCADE6 types can only be polymorphic in a single variable. It is important to note that since these languages feature binary clocks only, quantification is implicitly bounded by 1.

This section is structured as follows. First, we describe the new grammar of clock types and the related typing judgments. Even if the general design stays the same, the addition of polymorphic clock types adds formal complexity to the type system given in Chapter 3. The typed semantics has to be modified accordingly, and the construction of the interpretation of polymorphic clock types requires the use of more complex domain constructions. We then explain the compilation to machines, which is relatively simple. The last part of this section discusses the limitations of this approach to clock polymorphism, and details the interaction with local time scales and nodes.

5.3.1 Type System

Syntax The addition of bounded polymorphism only affects the syntax of types, but alters it in a drastic way. Figure 5.11 gives the modified syntax of types and clock types, as well as the new notion of *clock type context*, often abbreviated *clock context*. We add the aforementioned type constructor $\forall(\alpha \leq n).t$ to types. In order to accommodate for clock type variables, we change the grammar of clock types completely. A clock type *ct* can now be either atomic or

t	+=	$\forall(\alpha \leq n).t$	Bounded quantification
ct	::=	**base**	Base clock $(1)^\omega$
	|	α	Polymorphic clock type
	|	ct **on** p	Compound clock type
Δ	::=	$\square$	Empty clock type context
	|	$\Delta, \alpha \leq n$	Clock type binding

Figure 5.11: Clock polymorphism - extended type syntax

$$
\begin{array}{lcllcl}
FTV(\textbf{base}) & = & \varnothing & FTV(dt :: ct) & = & FTV(ct) \\
FTV(\alpha) & = & \{\alpha\} & FTV(t_1 \otimes t_2) & = & FTV(ct_1) \cup FTV(ct_2) \\
FTV(ct\ \textbf{on}\ p) & = & FTV(ct) & FTV(t_1 \multimap t_2) & = & FTV(ct_1) \cup FTV(ct_2) \\
FTV(\square) & = & \varnothing & FTV(\forall(\alpha \leq n).t) & = & FTV(t) \setminus \{\alpha\} \\
FTV(\Gamma, x : t) & = & FTV(\Gamma) \cup FTV(t) &&&
\end{array}
$$

Figure 5.12: Clock polymorphism - extended type system - free type variables

$$\boxed{ct_1 \equiv ct_2}$$

$$\frac{}{ct \equiv ct} \qquad \frac{ct_1 \equiv ct_2 \qquad ct_2 \equiv ct_3}{ct_1 \equiv ct_3} \qquad \frac{ct_2 \equiv ct_1}{ct_1 \equiv ct_2} \qquad \frac{ct_1 \equiv ct_2}{ct_1\ \textbf{on}\ p \equiv ct_2\ \textbf{on}\ p}$$

$$\frac{ct_1 \equiv ct_2}{ct_1\ \textbf{on}\ p_1\ \textbf{on}\ p_2 \equiv ct_2\ \textbf{on}\ (p_1\ on\ p_2)} \qquad \frac{ct_1 \equiv ct_2}{ct_1 \equiv ct_2\ \textbf{on}\ (1)}$$

Figure 5.13: Clock polymorphism - extended type system - equivalence judgment

compound. Atomic clock types are either the base clock **base**, which denotes the $(1)^\omega$ stream, or some clock type variable α. A compound clock is always of the form ct **on** p, with p an ultimately periodic integer word. Note that all the clock types of Chapter 3 can be expressed in the new grammar by prefixing them with **base** and shifting the periodic words to the right. A clock context Δ is a list of bounded clock type variables. Figure 5.12 gives the precise definition of the set $FV(\_)$ of free clock type variables appearing in a clock type, type or typing context.

Clock type equivalence A first question we need to address is clock equivalence. In Chapter 3, the equivalence of two clock types ct_1 and ct_2 was easy since we could simply test the equivalence of the exact representation of ct_1 and ct_2 as ultimately periodic words $nf(ct_1)$ and $nf(ct_2)$. In the richer clock type language of this section, having an explicit judgment expressing that two clock types are equivalent will prove more convenient. Figure 5.13 gives the (untyped) equivalence judgment $ct_1 \equiv ct_2$. The rules are simple. They express that clock type equivalence is an equivalence relation and is thus reflexive, symmetric and transitive, that

$$
\begin{array}{lcllcl}
nf(\mathbf{base}) & = & (\mathbf{base},(1)) & r(ct) & = & r \text{ where } (r,p) = nf(ct) \\
nf(\alpha) & = & (\alpha,(1)) & p(ct) & = & p \text{ where } (r,p) = nf(ct) \\
nf(ct\ \mathbf{on}\ p) & = & (r, p'\ on\ p) \text{ where } (r,p') = nf(ct) & & &
\end{array}
$$

Figure 5.14: Clock polymorphism - clock type normalization

it is a congruence, that compositions of ultimately periodic clock types can be simplified and that $(1)^\omega$ is neutral for clock composition.

Remark 28. The equivalence judgment can be decided using the function $nf(ct)$ given in Figure 5.14. This function computes the *root* $r(ct)$ and *periodic part* $p(ct)$ of a clock type ct, the root being either **base** or a variable α. Two clock types are equivalent if they have the same root and their periodic parts are equivalent in the sense of Chapter 2.

Well-formedness Now that the (clock) type language involves variables for which quantifiers act as binders, their manipulation becomes more complicated and scoping issues appear. In particular, we must be able to statically express that a clock type ct is *well-formed* in a given clock context, which means that its free variables (of which there is at most one) appear in the context. Since the clock context provides bounds for each variable, any well-formed clock type denotes a set of bounded clocks. This notion of well-formedness must be extended to types in the obvious manner: if all clock types appearing in a type are well-formed then so is the type.

Figure 5.15 presents the judgments that express well-formedness of clock types, types and contexts. The judgment $\Delta \vdash ct \leq n$ expresses that ct is bounded by n in the context Δ. Let us explain its rules.

- The clock type **base** is well-formed in any context, and is binary (rule CtBase).
- Rules CtVar and CtWeaken are the usual variable and weakening rules expressed at the clock type level.
- The bound of ct **on** p is, in the worst case, the bound of ct multiplied by the maximum integer in p, that is $\lceil p \rceil$. This is what CtOn expresses.
- Rule CtCong express that boundedness is compatible with equivalence. This makes it possible to refine the bound of a clock type by simplifying some of its ultimately periodic words, for example.

The judgment $\Delta \vdash t$ type, whose rules are given in the lower part of Figure 5.15, expresses that the type t is well-formed in clock context Δ. Basically, a type is well-formed if all its internal clock types are well-formed; the bounded quantifier $\forall(\alpha \leq n).t$ adds $\alpha \leq n$ to Δ. A program context is well-formed if all the types that appear in it are well-formed.

Auxiliary judgments Since the clock type language has changed we must update the definition of the adaptability, gathering and scattering judgments. The changes mostly consist in changing the AdaptStream, UpStream and DownStream rules, which are the only one

$\boxed{\Delta \vdash ct \leq n}$

$$\textsc{CtBase}\ \frac{}{\square \vdash \mathbf{base} \leq 1} \qquad \textsc{CtVar}\ \frac{}{\Delta, \alpha \leq n \vdash \alpha \leq n} \qquad \textsc{CtWeaken}\ \frac{\Delta \vdash ct \leq n' \qquad \alpha \notin FTV(ct)}{\Delta, \alpha \leq n \vdash ct \leq n'}$$

$$\textsc{CtOn}\ \frac{\Delta \vdash ct \leq n}{\Delta \vdash ct\ \mathbf{on}\ p \leq n \times \lceil p \rceil} \qquad \textsc{CtCong}\ \frac{\Delta \vdash ct' \leq n \qquad ct \equiv ct'}{\Delta \vdash ct_1 \leq n}$$

$\boxed{\Delta \vdash t\ \text{type}}$

$$\textsc{TyStream}\ \frac{\Delta \vdash ct \leq n}{\Delta \vdash dt :: ct\ \text{type}} \qquad \textsc{TyProd}\ \frac{\Delta \vdash t_1\ \text{type} \qquad \Delta \vdash t_2\ \text{type}}{\Delta \vdash t_1 \otimes t_2\ \text{type}} \qquad \textsc{TyArrow}\ \frac{\Delta \vdash t_1\ \text{type} \qquad \Delta \vdash t_2\ \text{type}}{\Delta \vdash t_1 \multimap t_2\ \text{type}} \qquad \textsc{TyForall}\ \frac{\Delta, \alpha \leq n \vdash t\ \text{type}}{\Delta \vdash \forall(\alpha \leq n).t\ \text{type}}$$

$\boxed{\Delta \vdash \Gamma\ \text{ctx}}$

$$\textsc{CtxEmpty}\ \frac{}{\Delta \vdash \square\ \text{ctx}} \qquad \textsc{CtxCons}\ \frac{\Delta \vdash \Gamma\ \text{ctx} \qquad \Delta \vdash t\ \text{type}}{\Delta \vdash \Gamma, x : t\ \text{ctx}}$$

Figure 5.15: Clock polymorphism - extended type system - well-formedness judgments

that manipulate clock types. Their updated versions rely on the new *clock adaptatability* judgment $\Delta \vdash ct_1 <:_k ct_2$, *clock gathering* judgment $\Delta \vdash ct_1 \uparrow_{ct} ct_2$, and *clock scattering* judgment $\Delta \vdash ct_1 \downarrow_{ct} ct_2$. The updated and new judgments are given in Figure 5.16.

The clock adaptability judgment consists in two rules. The first rule, AdaptOn, is the most interesting one. It expresses a sufficient condition on the delay between $ct\ \mathbf{on}\ p_1$ and $ct\ \mathbf{on}\ p_2$. If the largest integer in the clocks denoted by ct is bounded by n, in the worst case the delay k between p_1 and p_2 should be decremented by $n-1$. The second rule, AdaptCong, enables the use of clock equivalence in adaptability judgments.

The clock gathering and scattering judgment have a dual structure. The most important rules are the simplest ones, UpCkBase and DownCkBase. Rule UpCkBase shows how the base clock, gathering by ct, is mapped to ct itself. Dually, in DownCkBase ct, scattered by itself, is mapped to the base clock. Rules UpCkOn and DownCkOn express that clock composition is compatible with gathering and scattering. The last rules, UpCkCong and DownCkCong, makes it possible to use clock equivalence in gathering and scattering judgments; note that the gathered input clock and scattered output clock are tested for equivalence in the empty context. The reason for this will become clear once we explain the updated Rescale expression

$\boxed{\Delta \vdash t <:_k t'}$

ADAPTSTREAM
$$\frac{ct_1 \equiv ct\ \mathbf{on}\ p_1 \qquad ct_2 \equiv ct\ \mathbf{on}\ p_2 \qquad \Delta \vdash ct \leq n \qquad p_1 <:_k p_2}{\Delta \vdash dt :: ct_1 <:_{k+1-n} dt :: ct_2}$$

ADAPTPROD
$$\frac{\Delta \vdash t_1 <:_k t_1' \qquad \Delta \vdash t_2 <:_k t_2'}{\Delta \vdash t_1 \otimes t_2 <:_k t_1' \otimes t_2'}$$

ADAPTARROW
$$\frac{\Delta \vdash t_1' <:_0 t_1 \qquad \Delta \vdash t_2 <:_n t_2'}{\Delta \vdash t_1 \multimap t_2 <:_n t_1' \multimap t_2'}$$

$\boxed{\Delta \vdash ct_1 \uparrow_{ct} ct_2}$

$$\frac{\square \vdash ct_1 \leq n_1 \qquad ct_2 \equiv ct_1[\mathbf{base}/ct] \qquad \Delta \vdash ct_2 \leq n_2}{\Delta \vdash ct_1 \uparrow_{ct} ct_2}$$

$\boxed{\Delta \vdash t \uparrow_{ct} t'}$ and $\boxed{\Delta \vdash t \downarrow_{ct} t'}$

UPSTREAM
$$\frac{\Delta \vdash ct_1 \uparrow_{ct} ct_2}{\Delta \vdash dt :: ct_1 \uparrow_{ct} dt :: ct_2}$$

DOWNSTREAM
$$\frac{\Delta \vdash ct_2 \uparrow_{ct} ct_1}{\Delta \vdash dt :: ct_1 \downarrow_{ct} dt :: ct_2}$$

UPPROD
$$\frac{\Delta \vdash t_1 \uparrow_{ct} t_1' \qquad \Delta \vdash t_2 \uparrow_{ct} t_2'}{\Delta \vdash t_1 \otimes t_2 \uparrow_{ct} t_1' \otimes t_2'}$$

DOWNPROD
$$\frac{\Delta \vdash t_1 \downarrow_{ct} t_1' \qquad \Delta \vdash t_2 \downarrow_{ct} t_2'}{\Delta \vdash t_1 \otimes t_2 \downarrow_{ct} t_1' \otimes t_2'}$$

UPARROW
$$\frac{\Delta \vdash t_1' \downarrow_{ct} t_1 \qquad \Delta \vdash t_2 \uparrow_{ct} t_2'}{\Delta \vdash t_1 \multimap t_2 \uparrow_{ct} t_1' \multimap t_2'}$$

DOWNARROW
$$\frac{\Delta \vdash t_1' \downarrow_{ct} t_1 \qquad \Delta \vdash t_2 \downarrow_{ct} t_2' \qquad \Delta \vdash ct \leq 1}{\Delta \vdash t_1 \multimap t_2 \downarrow_{ct} t_1' \multimap t_2'}$$

UPON
$$\frac{\Delta \vdash t \uparrow_{ct_2} t'' \qquad \Delta \vdash t'' \uparrow_{ct_1} t' \qquad \Delta \vdash ct \uparrow_{ct_1} ct_2}{\Delta \vdash t \uparrow_{ct} t'}$$

DOWNON
$$\frac{\Delta \vdash t \downarrow_{ct_1} t'' \qquad \Delta \vdash t'' \downarrow_{ct_2} t' \qquad \Delta \vdash ct \uparrow_{ct_1} ct_2}{\Delta \vdash t \downarrow_{ct} t'}$$

UPINV
$$\frac{\Delta \vdash t \downarrow_{ct'} t' \qquad \Delta \vdash ct' \uparrow_{ct} \mathbf{base}}{\Delta \vdash t \uparrow_{ct} t'}$$

DOWNINV
$$\frac{\Delta \vdash t \uparrow_{ct'} t' \qquad \Delta \vdash ct' \uparrow_{ct} \mathbf{base}}{\Delta \vdash t \downarrow_{ct} t'}$$

UPCONG
$$\frac{\Delta \vdash t_1 \uparrow_{ct'} t_2 \qquad ct \equiv ct'}{\Delta \vdash t_1 \uparrow_{ct} t_2}$$

DOWNCONG
$$\frac{\Delta \vdash t_1 \downarrow_{ct'} t_2 \qquad ct \equiv ct'}{\Delta \vdash t_1 \downarrow_{ct} t_2}$$

$\boxed{\Delta \vdash \Gamma \downarrow_{ct} \Gamma'}$

DOWNCTXEMPTY
$$\frac{}{\Delta \vdash \square \downarrow_{ct} \square}$$

DOWNCTXCONS
$$\frac{\Delta \vdash \Gamma \downarrow_{ct} \Gamma' \qquad \Delta \vdash t \downarrow_{ct} t'}{\Delta \vdash \Gamma, x : t \downarrow_{ct} \Gamma', x : t'}$$

DOWNCTXWEAKEN
$$\frac{\Delta \vdash \Gamma \downarrow_{ct} \Gamma' \qquad \vdash t\ \text{value} \qquad x \notin dom(\Gamma')}{\Delta \vdash \Gamma, x : t \downarrow_{ct} \Gamma'}$$

Figure 5.16: Clock polymorphism - extended type system - auxiliary judgments

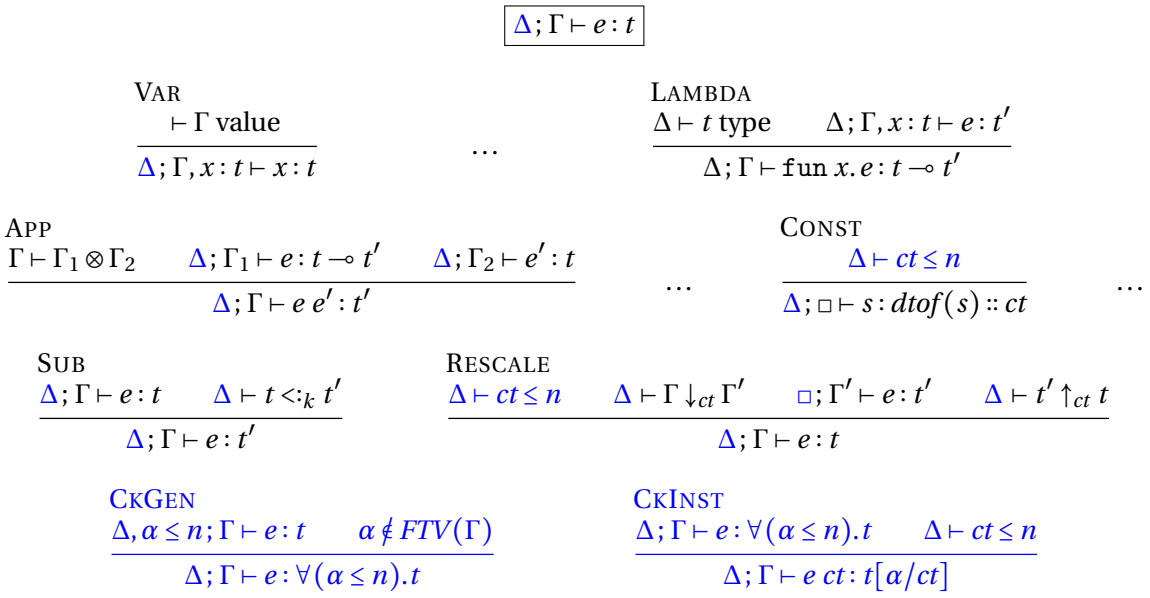


Figure 5.17: Clock polymorphism - extended type system - main judgment

typing rule.

The type adaptability, gathering and scattering judgments are modified to use on the corresponding notions on clocks. Existing rules are updated to pass the clock context along. Also, for the gathering and scattering judgments rules UPCONG and DOWNCONG makes it possible to reason up to equivalence on the clock type ct driving the local time scale. Furthermore, we add the DOWNCTXWEAKEN rule which removes a binding when entering a local time scale. Its presence makes it possible to remove bindings that cannot be scattered according to ct.

Main judgment The last part of the type system that has to be modified is the main expression typing judgment. The judgment is modified to include a clock context Δ, leading to the formulation $\Delta; \Gamma \vdash e : t$. The new rules are in Figure 5.17. There are three main changes. First, the clock context is passed along in every existing rule. It is handled in an intuitionistic fashion, as seen for example in rule APP where Δ is duplicated. Second, rules which make clock types appear from nowhere, such as CONST, must ensure that they are well-formed in the current clock context. Also, the RESCALE rule types its body in the empty clock context: polymorphic clock variables defined outside are not available inside the local time scale. Third, we add the traditional rules for polymorphic generalization and instantiation, CKGEN and CKINST. For generalization, the generalized variable α should not appear free in the types of the program context Γ, as in ordinary generalization. For instantiation, the clock type ct being substituted in the type must be well-formed in Δ. The operation $t[\alpha/ct]$ is the usual capture-avoiding substitution, applied to types and clock types.

Syntactic properties To develop the typed semantics of polymorphic clock types in a rigorous manner, we need to state some formal results relating the typing and well-formedness judgments. This is necessary since the synchronous semantics no longer interprets arbitrary types as domains, but limit this interpretation to well-formed types, or more precisely to well-formedness derivations. The results needed are the following.

Property 47. *Given derivations* $d_1 ::: (\Delta, \alpha \leq n \vdash t\ type)$ *and* $d_2 ::: (\Delta \vdash ct \leq n)$*, one may build a derivation* $subst(d_1, d_2) ::: (\Delta, \alpha \leq n \vdash t\ type)$.

Property 48. *If* $\Delta \vdash \Gamma\ ctx$ *and* $\Gamma \vdash \Gamma_1 \otimes \Gamma_2$ *hold, one may derive* $\Delta \vdash \Gamma_1\ ctx$ *and* $\Delta \vdash \Gamma_2\ type$.

Property 49. *If* $\Delta \vdash t <:_k t$ *holds, one may derive* $\Delta \vdash t\ type$ *and* $\Delta \vdash t'\ type$.

Property 50. *If* $\Delta \vdash t \uparrow_{ct} t'$ *holds, one may derive* $\Box \vdash t\ type$, $\Delta \vdash ct\ type n$ *and* $\Delta \vdash t'\ type$. *Dually, if* $\Delta \vdash t \downarrow_{ct} t$ *holds, one may derive* $\Delta \vdash t\ type$, $\Delta \vdash ct\ type n$ *and* $\Box \vdash t'\ type$. *Similarly for contexts.*

These properties are used to prove that the main typing judgment only produces well-formed terms. More precisely, for a typing derivation $\Delta; \Gamma \vdash e : t$, we need to express that Γ and t are well-formed in Δ. In practice, it is inconvenient to manipulate two judgments, one for contexts and one for type. This motivates the introduction of a compound judgment $\Delta \vdash \Gamma \multimap t$ wf whose only rule is given below.

$$\frac{\Delta \vdash \Gamma\ \text{ctx} \qquad \Delta \vdash t\ \text{type}}{\Delta \vdash \Gamma \multimap t\ \text{wf}}$$

The main syntactic result on well-typed expressions can now be stated as follows.

Property 51. *Any derivation* d *of the judgment* $\Delta; \Gamma \vdash e : t$ *can be examined to build a derivation* $wf(d ::: (\Delta; \Gamma \vdash e : t))$ *of* $\Delta \vdash \Gamma \multimap t$ *wf*.

Proof. Straightforward induction on d. The proof relies on a similar property on the separation judgment: given derivations of $\Delta \vdash \Gamma$ ctx and $\Gamma \vdash \Gamma_1 \otimes \Gamma_2$, one may build derivations of $\Delta \vdash \Gamma_1$ ctx and $\Delta \vdash \Gamma_2$ ctx. The case of the CkInst rule uses Property 47. □

We now have all the tools needed to discuss the impact of clock polymorphism on the typed semantics. This is the subject of the next subsection.

5.3.2 Typed Semantics

The typed semantics we have adopted for the basic language described in Chapter 3 follows a simple scheme. In this chapter, the expressions that could be proved to inhabit type t with free variables in Γ were naturally interpreted as continuous functions from the domain $\mathbf{S}[\![\Gamma]\!]$ to the domain $\mathbf{S}[\![t]\!]$. The introduction of clock polymorphism complicates this matter a lot. Now that clock types and thus types contain free variables, a type no longer denotes a domain but rather a family of domains indexed by the interpretation of its free variables. For instance, the

type $\mathbf{bool} :: \alpha$ does not represent a domain but rather some kind of function that associates to any clock w corresponding to α the domain $CStream_w(\mathbf{S}[\![\mathbf{bool}]\!])$.

The remaining problem is the interpretation of the type $\forall(\alpha \leq n).t$, which should somehow turn a function from clocks to domains into a domain. It happens that this is a traditional question arising in the semantic analysis of dependent and polymorphic type theories in domains; indeed, our polymorphic type constructor can be seen as a dependent function from **Ck** to something. The rest of this subsection explains how the general solution of this problem can be adapted to suit our need, without delving too much on details. Its understanding is not required for the rest of the thesis. We start with the required dose of category theory.

Category-theoretic background Given two categories C_1 and C_2, we write $C_1 \times C_2$ for the product category whose objects (resp. morphisms) are pairs of objects (resp. morphisms) from C_1 and C_2, and composition is performed componentwise. We write *Fst*, *Snd* for the first and second projection functors from $C_1 \times C_2$ to C_1 and C_2, respectively.

Any partial order P can be seen as a category with at most one morphism between two objects. More precisely, its objects are the elements of $|P|$ and there is one morphism between $x, x' \in |P|$ if and only if $x \leq x'$. Abusing notation, we write $x \leq x'$ for the corresponding morphism in P, seen as a category. From this point of view, a predomain (CPO) is a category with (small) directed colimits, and a domain (pointed CPO) has an initial object. Viewing (pre)domains as categories makes it possible to speak of functors coming from or going to a (pre)domain. The definition of a functor, specialized to the case where both source and target categories are partial orders, boils down to that of a monotonic function.

The category **CPO** has complete partial orders as objects and continuous functions as morphisms. We know that it is cartesian-closed. The subcategory $\mathbf{CPO}^{\mathbf{EP}}$ of **CPO** has the same objects but embedding-projection pairs $A \triangleleft B$ as morphisms between two cpos A and B.

Given a domain D, a functor $F : D \to \mathbf{CPO}^{\mathbf{EP}}$ associates to any element x of D a CPO $F(x)$, and to any $x \leq x'$ an embedding-projection pair $F(x \leq x') : F(x) \triangleleft F(x')$. Such a functor can be seen as a family of domains indexed by D in a natural way. Our main goal is to transform this functor into a domain, using a construction originally due to Grothendieck and specialized for the semantics of dependent types by a variety of authors, including Ehrhard [1988] and Coquand, Gunter, and Winskel [1989]. We base most of our description on the latter account.

Given a functor F, we construct a domain ΣF as follows. Its carrier is the set of pairs (x, y) where $x \in D$ and $y \in F(x)$. Its ordering relation is given by $(x, y) \leq_{\Sigma F} (x', y')$ whenever $x \leq_D x'$ and $F(x \leq_D x')(y) \leq_{F(x')} y'$. Its least element $\bot_{\Sigma F}$ is $(\bot_D, \bot_{F(\bot_D)})$. This domain should be thought of as a kind of disjoint union of all the domains $F(x)$. There is a natural continuous function p from ΣF to D that maps (x, y) to x. We sometimes call p together with ΣF a *fibration*, with D its *base*. The inverse image $p^{-1}(x)$ of $x \in D$ is called the *fiber* of x. Now, the domain ΠF is the sub-domain of $D \Rightarrow_c \Sigma F$ restricted to functions s such that $p \circ s = id_D$. Such functions are called *sections*. A section is thus a function that maps to each element x of D an element of the form (x, y) of ΣF in a continuous manner. Thus a section selects for each element of D an element in its fiber.

$$\begin{array}{lcl}
\mathbf{S}[\![\Delta]\!] & : & \mathbf{CPO} \\
\mathbf{S}[\![\square]\!] & = & \{\bot\} \\
\mathbf{S}[\![\Delta, x \leq n]\!] & = & \mathbf{S}[\![\Delta]\!] \times \mathbf{Ck}
\end{array}$$

(a) - Interpretation of clock contexts

$$\begin{array}{lcl}
\mathbf{S}[\![\Delta \vdash ct \leq n]\!] & \in & \mathbf{S}[\![\Delta]\!] \Rightarrow_c \mathbf{Ck} \\
\mathbf{S}[\![\square \vdash \mathbf{base} \leq 1]\!] & = & \lambda\delta.(1)^\omega \\
\mathbf{S}[\![\Delta, \alpha \leq n \vdash \alpha \leq n]\!] & = & \pi_r \\
\mathbf{S}[\![\Delta, \alpha \leq n \vdash ct \leq n']\!] & = & \mathbf{S}[\![\Delta \vdash ct \leq n']\!] \circ \pi_l \\
\mathbf{S}[\![\Delta \vdash ct\ \mathbf{on}\ u(v) \leq n \times \lceil u(v) \rceil]\!] & = & (\lambda w.w\ on\ u(v)^\omega) \circ \mathbf{S}[\![\Delta \vdash ct \leq n]\!] \\
\mathbf{S}[\![\Delta \vdash ct \leq n]\!] & = & \mathbf{S}[\![\Delta \vdash ct' \leq n]\!]
\end{array}$$

(b) - Interpretation of clock well-formedness

Figure 5.18: Clock polymorphism - typed semantics - interpretation of clock types

Following Coquand et al. [Coquand et al., 1989, 3.4], we extend the above construction to handle functors with additional parameters. Given two domains D_1, D_2 and a functor in $D_1 \times D_2 \to \mathbf{CPO^{EP}}$, we define the functor $\Pi^{D_1} F$ in $D_1 \to \mathbf{CPO^{EP}}$ as follows. Its action on objects is given by

$$(\Pi^{D_1} F)(x_1) = \Pi(\lambda x_2.F(x_1, x_2))$$

the λ-abstraction being understood here as simply notation. Its action on morphisms is

$$\begin{array}{lcl}
(\Pi^{D_1} F)(x_1 \leq_{D_1} x_1') & \in & (\Pi^{D_1} F)(x_1) \lhd (\Pi^{D_1} F)(x_1') \\
(\Pi^{D_1} F)(x_1 \leq_{D_1} x_1')^L(s)(x_2) & = & F(x_1 \leq_{D_1} x_1', x_2 \leq_{D_2} x_2)^L(s(x_2)) \\
(\Pi^{D_1} F)(x_1 \leq_{D_1} x_1')^R(t)(x_2) & = & F(x_1 \leq_{D_1} x_1', x_2 \leq_{D_2} x_2)^R(t(x_2))
\end{array}$$

with $(e,p)^L = e$ and $(e,p)^R = p$.

It is convenient for describing the semantics of types to define the pairing and exponential constructions on functors representing families of domains indexed by a domain. For F and G functors from the same domain to $\mathbf{CPO^{EP}}$, we define the two functors $F \times G$ and $F \Rightarrow_c G$. Their action on objects given "pointwise" by $(F \times G)(x) = F(x) \times G(x)$ and $(F \Rightarrow_c G)(x) = F(x) \Rightarrow_c G(x)$. Their action on morphisms is given by $(F \times G)(x) = F(x) \times G(x)$ and $(F \Rightarrow_c G)(x) = F(x) \Rightarrow_c G(x)$. The pairing and exponentiation of embedding-projection pairs has been defined in Section 3.3.1. Finally, given any domains D_1 and D_2, the constant functor $Const_{D_2} : D_1 \to \mathbf{CPO^{EP}}$ maps any element of D_1 to D_2 and any morphism $x \leq_{D_1} x'$ to the trivial embedding-projection pair (id_{D_2}, id_{D_2}).

Interpreting clock types In Chapter 3 we did interpret clock types as elements of **Ck**, since each clock type denoted a unique clock. Now that clock types contain free variables, we interpret them as functions from (the interpretation of) their free variables to the domain **Ck**. More precisely, we use once again the traditional solution of interpreting *derivations* rather

$$
\begin{array}{lcl}
\mathbf{S}[\![\Delta \vdash t \text{ type}]\!] & : & \mathbf{S}[\![\Delta]\!] \to \mathbf{CPO}^{\mathbf{EP}} \\
\mathbf{S}[\![\Delta \vdash dt :: ct \text{ type}]\!] & = & \mathit{CStream}^{\mathbf{S}[\![dt]\!]}_{\mathbf{S}[\![\Delta]\!]}(\mathbf{S}[\![\Delta \vdash ct \leq n]\!]) \\
\mathbf{S}[\![\Delta \vdash t_1 \otimes t_2 \text{ type}]\!] & = & \mathbf{S}[\![\Delta \vdash t_1 \text{ type}]\!] \times \mathbf{S}[\![\Delta \vdash t_2 \text{ type}]\!] \\
\mathbf{S}[\![\Delta \vdash t_1 \multimap t_2 \text{ type}]\!] & = & \mathbf{S}[\![\Delta \vdash t_1 \text{ type}]\!] \Rightarrow_c \mathbf{S}[\![\Delta \vdash t_2 \text{ type}]\!] \\
\mathbf{S}[\![\Delta \vdash \forall(\alpha \leq n).t \text{ type}]\!] & = & \Pi^{\mathbf{S}[\![\Delta]\!]}\mathbf{S}[\![\Delta, \alpha \leq n \vdash t \text{ type}]\!]
\end{array}
$$

(a) - Interpretation of well-formed types

$$
\begin{array}{lcl}
\mathbf{S}[\![\Delta \vdash \Gamma \text{ ctx}]\!] & : & \mathbf{S}[\![\Delta]\!] \to \mathbf{CPO}^{\mathbf{EP}} \\
\mathbf{S}[\![\Delta \vdash \Box \text{ ctx}]\!] & = & \mathit{Const}_{\{\perp\}} \\
\mathbf{S}[\![\Delta \vdash \Gamma, x : t \text{ ctx}]\!] & = & \mathbf{S}[\![\Delta \vdash \Gamma \text{ ctx}]\!] \times \mathbf{S}[\![\Delta \vdash t \text{ type}]\!]
\end{array}
$$

(b) - Interpretation of well-formed program contexts

$$
\begin{array}{lcl}
\mathbf{S}[\![\Delta \vdash \Gamma \multimap t \text{ wf}]\!] & : & \mathbf{S}[\![\Delta]\!] \to \mathbf{CPO}^{\mathbf{EP}} \\
\mathbf{S}[\![\Delta \vdash \Gamma \multimap t \text{ wf}]\!] & = & \mathbf{S}[\![\Delta \vdash \Gamma \text{ ctx}]\!] \Rightarrow_c \mathbf{S}[\![\Delta \vdash t \text{ type}]\!]
\end{array}
$$

(c) - Interpretation of the well-formedness judgment

Figure 5.19: Clock polymorphism - typed semantics - interpretation of types and contexts

than raw clock types. A clock context Δ is interpreted as a big tuple of clocks, and a well-formed clock $\Delta \vdash ct \leq n$ is interpreted as a continuous function form $\mathbf{S}[\![\Delta]\!]$ to $\mathbf{Ck}$. The corresponding definitions are given in Figure 5.18.

Interpreting types The interpretation of types is where the new typed semantics departs radically from the previous one. We interpret a derivation of the judgment $\Delta \vdash t$ type as a functor from $\mathbf{S}[\![\Delta]\!]$ to $\mathbf{CPO}^{\mathbf{EP}}$ using the previously introduced category-theoretical arsenal. The only remaining ingredient we need is the most specific to our case: it is the functor needed to interpret a type $dt :: ct$. Given any pair of clocks w, w', we know how to build the domains $\mathit{CStream}_w(D)$ and $\mathit{CStream}_{w'}(D)$. Let us observe that we can move between these two domains using the $\mathbf{repack}_{w'}$ and $\mathbf{repack}_w$ functions. A crucial observation is that when $w \sqsubseteq w'$, then these two functions form an embedding-projection pair. We can thus define the following functor.

$$
\begin{array}{lcl}
\mathit{CStream}^{D_2}_{D_1}(f \in D_1 \Rightarrow_c \mathbf{Ck}) & : & D_1 \to \mathbf{CPO}^{\mathbf{EP}} \\
(\mathit{CStream}^{D_2}_{D_1} f)(x) & = & \mathit{CStream}_{f(x)}(D_2) \\
(\mathit{CStream}^{D_2}_{D_1} f)(x \leq x') & = & (\mathbf{repack}_{f(x')}, \mathbf{repack}_{f(x)})
\end{array}
$$

This is a functor from the domain D_1 to the category of embedding-projection pairs. It is parametrized by a function f mapping an element of D_1 to a clock. In practice the domain D_1 will be the interpretation of a clock context Δ and the function f the interpretation of a judgment $\Delta \vdash ct \leq n$.

Figure 5.19 (a) gives the interpretation of types as functors. The functor defined above is used for the case of stream types. Pairing and exponentiation of functors handle the product and arrow cases. Polymorphic clock types are interpreted as sections using the parametrized domain of sections described earlier. Figure 5.19 (b) and (c) gives the simpler interpretation of the remaining well-formedness judgments. As usual contexts correspond to products and complete derivations $\Delta \vdash \Gamma \multimap t$ wf to functions.

Interpreting programs The interpretation of programs is a variation of that given in Chapter 3. The main idea is to interpret derivations of $\Delta;\Gamma \vdash e : t$ as sections of $\mathbf{S}[\![\Delta \vdash \Gamma \multimap t \text{ wf}]\!]$. Remember that the latter judgment can be derived from the former using Property 51. Note that programs are still interpret as elements of a domain. This leads to the type signature below.

$$\mathbf{S}[\![\Delta;\Gamma \vdash e : t]\!] \quad \in \quad \Pi\mathbf{S}[\![wf(\Delta;\Gamma \vdash e : t)]\!]$$

We do not give the precise definition of this interpretation but rather explain the general principles at work. From the semantics point of view, typing rules can be grouped into two distinct families.

The first group consists in typing rules where the clock context Δ does not change and is simply passed along. This covers all the preexisting rules from Chapter 3. To give the semantics of this group one builds the usual cartesian-closed combinators handling products and exponentials *fibrewise*. This means, informally, that they live in the fiber above a fixed $\delta \in \mathbf{S}[\![\Delta]\!]$. Let us briefly describe some operators found in our section construction kit. All the functors below are assumed to go from the same domain D to $\mathbf{CPO}^{\mathbf{EP}}$.

- Pairing: given functors F_1, F_2, G_1, G_2 and sections $s_1 \in \Pi(F_1 \Rightarrow_c G_1), s_2 \in \Pi(F_2 \Rightarrow_c G_2)$, the pairing combinator $(s_1 \times s_2)$ is a section in $\Pi((F_1 \times F_2) \Rightarrow_c (G_1 \times G_2))$. It is used for instance in the interpretation of rule PAIR.

- Composition: given functors F, G, H and sections $s_1 \in \Pi(G \Rightarrow_c H)$ and $s_2 \in \Pi(F \Rightarrow_c G)$, one may build $s_1 \circ s_2 \in \Pi(F \Rightarrow_c H)$. A basic building block, this combinator is used as plumbing in the interpretation of most rules.

- Duplication: given a functor F, one may build the *duplicating section* $Dupl_F$ belonging to $\Pi((F \times F) \Rightarrow_c F)$.[1] This combinator is used to duplicate values.

The definition of all the combinators above and others in the same group is simple since the index δ stays invariant. Thus, their action is given pointwise.

The second group is formed of the new rules CKGEN and CKINST which are related to the clock context or polymorphic clock types. The interpretation of these rules rely on two constructions. The first construction is a form of currying. Given two functors $F : D_1 \to \mathbf{CPO}^{\mathbf{EP}}$ and $G : D_1 \times D_2 \to \mathbf{CPO}^{\mathbf{EP}}$, the operation turns a section $s \in \Pi((F \circ Fst) \Rightarrow_c G)$ into

[1] This construction is often written Δ_F or simply Δ in category theory texts, but we wish to avoid confusion with clock contexts.

a section $Curry(s) \in \Pi(F \Rightarrow_c \Pi^{D_1} G)$. The second construction is, dually, a form of application. Given two functors $F : D_1 \to \mathbf{CPO}^{\mathbf{EP}}$ and $G : D_1 \times D_2 \to \mathbf{CPO}^{\mathbf{EP}}$ as well as a function $f \in D_1 \Rightarrow_c D_2$, the operation turns a section $s \in \Pi(F \Rightarrow_c \Pi^{D_1} G)$, into a section $App(s, f)$ belonging to $\Pi(F \Rightarrow_c \langle id_{D_1}, Const_{\mathbf{Ck}} \rangle)$.

This concludes the description of the typed semantics of the polymorphic clock language. There is nothing new here since the universal clock quantifier is basically a dependent product with its domain restricted to clocks. Interested readers looking for a complete description of the interpretation of dependent types in domains can find relevant details in Amadio and Curien [1998, Chapter 11]. More information on fibrations and their use in categorical logic can be found in the book of Jacobs [1999].

Remark 29. We have glossed over a lot of important details in the overview above. In particular, we have not explained how the substitution operation appearing in rule CkInst is to be handled. A formal handling of this question would probably benefit to the addition of explicit clock type substitutions in the language. An explicit substitution σ would be assigned a type $\vdash \sigma : \Delta_1 \Rightarrow \Delta_2$ and be interpreted as a function $\mathbf{S}[\![\sigma]\!] \in \mathbf{S}[\![\Delta_1]\!] \Rightarrow_c \mathbf{S}[\![\Delta_2]\!]$. Following an important idea of categorical logic, such a function induces a *base change* transporting data above $\mathbf{S}[\![\Delta_1]\!]$ into data above $\mathbf{S}[\![\Delta_2]\!]$. This operation can then be used to build the interpretation of $\Delta_2; \Gamma[\sigma] \vdash e : t[\sigma]$ from the one of $\Delta_1; \Gamma \vdash e : t$.

5.3.3 Compilation

In contrast with the typed semantics, the compilation of polymorphic clock types does not involve large changes. The idea is that the free clock type variables of an expression, described by a clock context, give rise to new inputs in its translation. Well-formed clock types are compiled to machines computing the successive value of the clock. Clock instantiation and generalization have natural implementations as machines.

Translating types In Chapter 4, a bound for any clock type could be computed by the $\lceil \_ \rceil$ function. Uses of this function permeate the translation since it is used in a very basic way to give the translation of the type $dt :: ct$. With polymorphic clock types, this function no longer exists: it is replaced with the explicit presence of well-formedness judgments $\Delta \vdash ct \leq n$. Thus, the compilation now acts on well-formed types and contexts, which ensures that we have access to the proper boundedness judgments.

Figure 5.20 gives the translation of clock contexts, well-formed types and well-formed contexts. A clock context is interpreted as a tuple of bounded integers. The interpretation of types is the same as in Chapter 4, with two exceptions. First, as explained above the bound for stream types $dt :: ct$ is given by the judgment expressing the well-formedness of ct in Δ. Second, polymorphic clock types have to be translated. Such a type of the form $\forall(\alpha \leq n).t$ is translated into a higher-order machine processing an input in $\mathtt{mint}\{n\}$ and returning an output in (the translation of) t. A program context Γ is as always a product. Finally, a well-typed context/type pair $\Delta \vdash \Gamma \multimap t$ wf is interpreted as a higher-order machine receiving (in curryfied form) two inputs corresponding to Δ and Γ and returning an output in t.

$$\boxed{(\!|\Delta|\!)}$$

$$\begin{array}{lcl}(\!|\square|\!) & = & \texttt{munit} \\ (\!|\Delta, \alpha \leq n|\!) & = & (\!|\Delta|\!) \boxtimes \texttt{mint}\{n\}\end{array}$$

(a) Translation of clock contexts

$$\boxed{\Delta \vdash t \text{ type}}$$

$$\begin{array}{lcl}(\!|\Delta \vdash dt :: ct \text{ type}|\!) & = & \texttt{unit} \multimap (\!|dt|\!)[n] \\ (\!|\Delta \vdash t_1 \otimes t_2 \text{ type}|\!) & = & (\!|\Delta \vdash t_1 \text{ type}|\!) \boxtimes (\!|\Delta \vdash t_2 \text{ type}|\!) \\ (\!|\Delta \vdash t_1 \multimap t_2 \text{ type}|\!) & = & (\!|\Delta \vdash t_1 \text{ type}|\!) \boxminus (\!|\Delta \vdash t_2 \text{ type}|\!) \\ (\!|\Delta \vdash \forall(\alpha \leq n).t \text{ type}|\!) & = & \texttt{mint}\{n\} \boxminus (\!|\Delta, \alpha \leq n \vdash t_1 \text{ type}|\!)\end{array}$$

(b) Translation of types

$$\boxed{\Delta \vdash \Gamma \text{ type}}$$

$$\begin{array}{lcl}(\!|\Delta \vdash \square \text{ ctx}|\!) & = & \texttt{munit} \\ (\!|\Delta \vdash \Gamma, x : t \text{ ctx}|\!) & = & (\!|\Delta \vdash \Gamma \text{ ctx}|\!) \boxtimes (\!|\Delta \vdash t \text{ type}|\!)\end{array}$$

(c) Translation of program contexts

$$(\!|\Delta \vdash \Gamma \multimap t \text{ wf}|\!) = (\!|\Delta|\!) \boxtimes (\!|\Delta \vdash \Gamma \text{ ctx}|\!) \boxminus (\!|\Delta \vdash t \text{ type}|\!)$$

Figure 5.20: Clock polymorphism - compilation - types

Translating clock types The fact that clock contexts are handled in an intuitionistic fashion in the type system comes from the fact that, morally, they only hold values and thus have no inputs. This means that clock contexts can be duplicated and erased. Figure 5.21 (a) and (b) give the corresponding machines.

The translation of a (well-formed) clock type $\Delta \vdash ct \leq n$ is a machine with inputs in Δ and an output in $\texttt{mint}\{n\}$. The translation of each rule is given in Figure 5.21 (c). Most cases are handled using machines in ways that we have already explained. The base machine **base** denotes a constant one. Weakening and variables are handled in the usual way, relying on the intuitionistic nature of the context Δ. Clock composition is, as before, implemented by the summation machine.

Translating auxiliary judgments Since we have changed the way we define the translation of types, by definition the interpretation of all judgments have to be modified. For auxiliary judgments the modifications are minimal. The only slightly significant changes happen in the places where $(\!|ct|\!)$ is replaced with $(\!|\Delta \vdash ct \leq n|\!)$. The design of the type system, together with the

$$
\begin{array}{lcl}
(\!|\Delta|\!)_E & \in & \mathcal{M}_{(\!|\Delta|\!)^*} \\
(\!|\square|\!)_E & = & \mathtt{mid}_{\mathtt{unit}} \\
(\!|\Delta, \alpha \le n|\!)_E & = & (\!|\Delta|\!)_E \parallel \mathtt{mforg}_{\mathtt{mint}\{n\}}
\end{array}
$$

(a) Clock context duplication

$$
\begin{array}{lcl}
(\!|\Delta \vdash ct \le n|\!) & \in & \mathcal{M}_{(\!|\Delta|\!) \boxminus (\!|\Delta|\!) \boxtimes (\!|\Delta|\!)} \\
(\!|\square|\!)_D & = & \mathtt{mid}_{\mathtt{unit}\times\mathtt{unit}\times\mathtt{unit}} \\
(\!|\Delta, \alpha \le n|\!)_D & = & \langle (\!|\Delta|\!), (\!|\Delta|\!), \mathtt{mint}\{n\}, \mathtt{mint}\{n\} \mapsto 0,2,1,3 \rangle_{ho} \\
 & & \boxdot\, ((\!|\Delta|\!)_D \boxtimes \mathtt{mdup}_{\mathtt{mint}\{n\}})
\end{array}
$$

(b) Clock context erasure

$$
\begin{array}{lcl}
(\!|\Delta \vdash ct \le n|\!) & \in & \mathcal{M}_{(\!|\Delta|\!) \boxminus \mathtt{mint}\{n\}} \\
(\!|\square \vdash \mathbf{base} \le 1|\!) & = & \mathtt{mid}_{\mathtt{unit}} \parallel \mathtt{mconst}_1 \\
(\!|\Delta, \alpha \le n \vdash \alpha \le n|\!) & = & (\!|\Delta|\!)_E \parallel \mathtt{mhoid}_{\mathtt{mint}\{n\}} \\
(\!|\Delta, \alpha \le n \vdash ct \le n'|\!) & = & (\!|\Delta \vdash ct \le n'|\!) \boxdot m_l \\
 & & \text{where } m_l \in \mathcal{M}_{(\!|\Delta, \alpha \le n|\!) \boxminus (\!|\Delta|\!)} \\
 & & \qquad m_l = \mathtt{mhoneutr}_{(\!|\Delta|\!)} \boxdot (\mathtt{mforg}_{\{n\}} \boxtimes \mathtt{mhoid}_{(\!|\Delta|\!)}) \\
 & & \qquad\quad \boxdot \langle (\!|\Delta|\!)^*, \mathtt{mint}\{n\}^*, (\!|\Delta|\!) \mapsto 1,0,2 \rangle_{ho} \\
(\!|\Delta \vdash ct \,\mathbf{on}\, p \le n \times \lceil p \rceil|\!) & = & \mathtt{mhosum}^n_{\lceil p \rceil} \boxdot m \boxdot (\!|\Delta \vdash ct \le n|\!) \\
 & & \text{where } m \in \mathcal{M}_{\mathtt{mint}\{n\} \boxminus \mathtt{mint}\{\lceil p \rceil\}[n]} \\
 & & \qquad m = \mathtt{mneutrinv}_{\mathtt{mint}\{\lceil p \rceil\}[n]^+} \\
 & & \qquad\quad \bullet\, \mathtt{mrepl}_n(\mathtt{mpw}_p(0)) \\
(\!|\Delta \vdash ct \le n|\!) & = & (\!|\Delta \vdash ct' \le n|\!)
\end{array}
$$

(c) Translation of clock type well-formedness

Figure 5.21: Clock polymorphism - compilation - clock types

$$
\begin{array}{lcl}
(\!|\Gamma \vdash \Gamma_1 \otimes \Gamma_2|\!) & \in & \mathcal{M}_{(\!|\Delta \vdash \Gamma\ \mathrm{ctx}|\!) \boxminus (\!|\Delta \vdash \Gamma_1\ \mathrm{ctx}|\!) \boxtimes (\!|\Delta \vdash \Gamma_2\ \mathrm{ctx}|\!)} \text{ assuming } \Delta \vdash \Gamma \text{ ctx} \\
(\!|\Delta \vdash t <:_k t'|\!) & \in & \mathcal{M}_{(\!|\Delta|\!) \boxtimes (\!|\Delta \vdash t\ \mathrm{type}|\!) \boxminus (\!|\Delta \vdash t'\ \mathrm{type}|\!)} \\
(\!|\Delta \vdash t \uparrow_{ct} t'|\!) & \in & \mathcal{M}_{(\!|\Delta|\!) \boxtimes (\!|\square \vdash t\ \mathrm{type}|\!)[n] \boxminus (\!|\Delta \vdash t'\ \mathrm{type}|\!)} \text{ where } \Delta \vdash ct \le n \\
(\!|\Delta \vdash t \downarrow_{ct} t'|\!) & \in & \mathcal{M}_{(\!|\Delta|\!) \boxtimes (\!|\Delta \vdash t\ \mathrm{type}|\!) \boxminus (\!|\square \vdash t'\ \mathrm{type}|\!)[n]} \text{ where } \Delta \vdash ct \le n
\end{array}
$$

Figure 5.22: Clock polymorphism - compilation - auxiliary judgments

$$
\begin{array}{lcl}
(\!|\Delta;\Gamma\vdash e:t|\!) & \in & \mathcal{M}_{(\!|wf(\Delta;\Gamma\vdash e:t)|\!)} \\
(\!|\Delta;\Gamma,x:t\vdash x:t|\!) & = & (\!|\Delta|\!)_E \parallel (\!|\vdash \Gamma \text{ value}|\!)_E \parallel \mathtt{mhoid}_{(\!|t|\!)} \\
 & \cdots & \\
(\!|\Delta;\Gamma\vdash e\,e':t'|\!) & = & \mathtt{mhoev}_{(\!|t|\!),(\!|t'|\!)} \boxdot ((\!|\Delta;\Gamma_1\vdash e:t\multimap t'|\!) \boxtimes (\!|\Delta;\Gamma_2\vdash e':t|\!)) \\
 & & \boxdot \langle (\!|\Delta|\!),(\!|\Delta|\!),(\!|\Gamma_1|\!),(\!|\Gamma_2|\!) \mapsto 0,2,1,3\rangle_{ho} \\
 & & \boxdot ((\!|\Delta|\!)_D \boxtimes (\!|\Gamma\vdash\Gamma_1\otimes\Gamma_2|\!)) \\
 & \cdots & \\
(\!|\Delta;\square\vdash s:dtof(s)::ct|\!) & = & m_2 \boxdot ((\!|\Delta\vdash ct\le n|\!) \boxtimes m_1) \boxdot (\!|\Delta|\!)_D \\
 & & \text{where } m_1 \in \mathcal{M}_{(\!|\Delta|\!)\boxminus(\!|dtof(s)|\!)} \\
 & & \qquad m_1 = (\!|\Delta|\!)_E \parallel \mathtt{mconst}_s \\
 & & \qquad m_2 \in \mathcal{M}_{\mathtt{mint}\{n\}\boxtimes(\!|dtof(s)|\!)\boxminus(\!|dtof(s)|\!)[n]} \\
 & & \qquad m_2 = (\mathtt{mforg}_{(\!|dtof(s)[n]|\!)^-} \parallel \mathtt{mstutt}^n_{(\!|dtof(s)|\!)^+}) \\
 & & \qquad\quad \bullet \langle \mathtt{mint}\{n\}^+,(\!|dtof(s)|\!)^+,(\!|dtof(s)[n]|\!)^- \mapsto 2,0,1\rangle \\
 & \cdots & \\
(\!|\Delta;\Gamma\vdash e:t'|\!) & = & (\!|\Delta\vdash t<:_k t'|\!) \boxdot (\mathtt{mhoid}_{(\!|\Delta|\!)} \boxtimes (\!|\Delta;\Gamma\vdash e:t|\!)) \\
 & & \boxdot ((\!|\Delta|\!)_D \boxtimes \mathtt{mhoid}_{(\!|\Gamma|\!)}) \\
(\!|\Delta;\Gamma\vdash e:t|\!) & = & (\!|\Delta\vdash t'\uparrow_{ct} t|\!) \boxdot (\mathtt{mhoid}_{(\!|\Delta|\!)} \boxtimes (m \boxdot ((\!|\Delta\vdash ct\le n|\!) \boxtimes \mathtt{mhoid}_{(\!|\Gamma'|\!)}))) \\
 & & \boxdot ((\!|\Delta|\!)_D \boxtimes (\!|\Delta\vdash\Gamma\downarrow_{ct}\Gamma'|\!)) \boxdot ((\!|\Delta|\!)_D \boxtimes \mathtt{mhoid}_{(\!|\Gamma|\!)}) \\
 & & \text{where } m \in \mathcal{M}_{\mathtt{mint}\{n\}\boxtimes(\!|\Gamma'|\!)[n]\boxminus(\!|t'|\!)[n]} \\
 & & \qquad m = \mathtt{mhorepl}^n_{(\!|\Gamma'|\!),(\!|t'|\!)}((\!|\square;\Gamma'\vdash e:t'|\!) \boxdot \mathtt{mhoneutr}_{(\!|\Gamma'|\!)}) \\
(\!|\Delta;\Gamma\vdash e:\forall(\alpha\le n).t|\!) & = & (\!|\Delta,\alpha\le n;\Gamma\vdash e:t|\!) \boxdot \langle (\!|\Delta|\!),\mathtt{mint}\{n\},(\!|\Gamma|\!) \mapsto 0,2,1\rangle_{ho} \\
(\!|\Delta;\Gamma\vdash e:t[\alpha/ct]|\!) & = & (\!|\Delta;\Gamma\vdash e:\forall(\alpha\le n).t|\!) \boxdot \langle \mathtt{mint}\{n\},(\!|\Gamma|\!),(\!|\Delta|\!) \mapsto 1,2,0\rangle_{ho} \\
 & & \boxdot ((\!|\Delta\vdash ct\le n|\!) \boxtimes \mathtt{mhoid}_{(\!|\Delta|\!)\boxtimes(\!|\Gamma|\!)}) \boxdot ((\!|\Delta|\!)_D \boxtimes \mathtt{mhoid}_{(\!|\Gamma|\!)})
\end{array}
$$

Figure 5.23: Clock polymorphism - compilation - expressions

syntactic properties given at the end of Section 5.3.1, ensure that such a derivation is always available. Figure 5.22 gives the type of the new translation functions.

Translating expressions Figure 5.23 gives the translation of the judgment $\Delta;\Gamma\vdash e:t$. The generated higher-order machine has $(\!|\Delta|\!)$ and $(\!|\Delta\vdash\Gamma\text{ ctx}|\!)$ as inputs and $(\!|\Delta\vdash t\text{ ctx}|\!)$ as output. This is actually well-defined because of Property 51. The modifications are straightforward as they mostly consist in routing the values of free clock variables into premises and auxiliary judgments, duplicating or erasing them as needed. The only new rules, clock generalization and instantiation, are respectively compiled to a form of currying and a form of application. More precisely, the CkGen rule is a form of currying akin to the Fun rule, but this time acting on the program context, while CkInst is a form of application akin to App.

5.3.4 Discussion

Unit delays A core operator of Lustre and Lucid Synchrone is the *initialized delay* operator $\texttt{fby}_s$, s being the initial content of the delay. The untyped semantics of this operator simply adds s in front of its input stream, and is thus just a stream constructor. This is reminiscent of the *registers* present in synchronous circuits. Using polymorphic clock types we can express its usual typing rule as found in Lustre or Lucid Synchrone. In these languages the clock of its input is the same as the clock of its output.

$$\begin{array}{rcl} \texttt{fby}_n & : & \forall(\alpha \leq 1).\mathbf{int} :: \alpha \multimap \mathbf{int} :: \alpha \\ \texttt{fby}_n & = & \texttt{fun x.merge}\ 1(0)\ n\ \texttt{x} \end{array}$$

The typing derivation of this program with the type above requires the application of the subsumption rule on x. In other words, there is a buffer that transforms the clock type α of x into α **on** $0(1)$. Thus, when endowed with their usual clock type, delay operators correspond to *one-place buffers*. Note that this type is not as good as one could hope, since

$$\texttt{fby}_n \quad : \quad \forall(\alpha \leq 1).\mathbf{int} :: \alpha\ \mathbf{on}\ 0(1) \multimap \mathbf{int} :: \alpha$$

is also valid but strictly more general. The traditional typing rule of delays is particularly problematic when one wishes to handle causality through clock types since it adds artificial dependencies to what was originally the only buffering operator of synchronous language. We will discuss this matter and related ones in Section 6.2.

Local time scales As explained before, clock polymorphism offers a new, distinct dimension of reuse compared to local time scales. Local time scales are restrictive because they affect the free variables of an expression—in other words, because they only apply to closed programs. The CKINST rule suffers from no such restriction. Clock polymorphism makes it possible to give a piece of code a generic clock type which can then be specialized later, in a part of the program that is not known yet. Thus, clock polymorphism is more modular.

Remark 30. A related point is the fact that in the updated RESCALE rule the premise typing the expression inside a local time scale require that the clock context be empty. This is because free clock variables are conceptually inputs to the expression but cannot be scattered.

On the other hand, local time scales are sometimes less limiting than polymorphism. We illustrate this point through the example of the program p defined below. This program computes the classic stream $0.1.2\ldots$ of natural numbers as the fixpoint of function f.

$$\begin{array}{rcl} p : ct & \stackrel{\text{def}}{=} & \texttt{fix}\ f \\ f : ct_1 \multimap ct_2 & \stackrel{\text{def}}{=} & \texttt{fun}\ x.\texttt{merge}\ 1(0)\ 0\ (1+x) \end{array} \qquad \begin{array}{ll} \text{(A)} & ct_1 <:_0 ct_2\ \mathbf{on}\ 0(1) \\ \text{(B)} & ct_2 <:_1 ct_1 \end{array}$$

The clock types appearing in the type of f must obey the constraints given above. Constraint (A) comes from the typing rule of merge, combined with the possible use of adaptability/subtyping, while constraint (B) comes from the typing rule of fixpoints.

$$\dfrac{\dfrac{\dots}{ct \equiv ct\ \mathbf{on}\ (1)} \qquad \dfrac{}{ct\ \mathbf{on}\ 0(1) \equiv ct\ \mathbf{on}\ 0(1)} \qquad \dfrac{\dfrac{\dots}{ct \equiv ct} \qquad (1) <:_1 0(1)}{\square \vdash ct\ \mathbf{on}\ (1) <:_{1+1-1} ct\ \mathbf{on}\ 0(1)}}{\square \vdash ct <:_1 ct\ \mathbf{on}\ 0(1)}$$

(a)

$$\dfrac{\dfrac{\dots}{\alpha \equiv \alpha\ \mathbf{on}\ (1)} \qquad \dfrac{}{\alpha\ \mathbf{on}\ 0(1) \equiv \alpha\ \mathbf{on}\ 0(1)} \qquad \dfrac{\dfrac{}{\alpha \equiv \alpha} \qquad (1) <:_1 0(1)}{\alpha \leq 1 \vdash \alpha\ \mathbf{on}\ (1) <:_{1+1-1} \alpha\ \mathbf{on}\ 0(1)}}{\alpha \leq 1 \vdash \alpha <:_1 \alpha\ \mathbf{on}\ 0(1)}$$

(b)

$$\dfrac{\dfrac{\dots}{\square \vdash \square \downarrow_\alpha \square} \qquad \dfrac{\dots}{\square; \square \vdash p : \mathbf{base}} \qquad \dfrac{\dots}{\square \vdash \mathbf{base} \uparrow_{ct} ct} \qquad \dfrac{\dots}{\square \vdash ct \leq n_c}}{\square; \square \vdash p : ct}$$

(c)

$$\dfrac{\dfrac{\dfrac{\dots}{\alpha \leq n_d \vdash \square \downarrow_\alpha \square} \qquad \dfrac{\dots}{\square; \square \vdash p : \mathbf{base}} \qquad \dfrac{\dots}{\alpha \leq n_d \vdash \mathbf{base} \uparrow_\alpha \alpha} \qquad \dfrac{}{\alpha \leq n_d \vdash \alpha \leq n_d}}{\alpha \leq n_d; \square \vdash p : \alpha}}{\square; \square \vdash p : \forall(\alpha \leq n_d).\alpha}$$

(d)

Figure 5.24: Clock polymorphism - example derivations

Let us first discuss the clock type assignments for p and f that do not involve the use of the RESCALE rule. In this case $ct = ct_2$ and the problem boils down to the search for clock types satisfying constraints (A) and (B).

The immediate, boring solution is to take $ct = ct_2 = \mathbf{base}$ and $ct_1 = \mathbf{base}\ \mathbf{on}\ 0(1)$. In the world of binary clocks this is by definition the "best" solution since it has the highest throughput and no latency. More generally, fix any *binary* closed clock type $ct = ct_2$ and take $ct_1 = ct\ \mathbf{on}\ 0(1)$. This gives a solution to the system above since we have $\square \vdash ct <:_1 ct\ \mathbf{on}\ 0(1)$ by the derivation (a) of Figure 5.24. The fact that ct is binary and thus bounded by 1 matters in the application of rule ADAPTON, where this bound is highlighted in red.

To go one step further we can use polymorphic clock types to generalize the previous reasoning, replacing the arbitrary binary clock ct with a variable α bounded by 1. This would give to p the polymorphic clock type $\forall(\alpha \leq 1).\alpha$. The corresponding derivation of $\alpha <:_1 \alpha\ \mathbf{on}\ 0(1)$, assuming $\alpha \leq 1$, is given in Figure 5.24 (b).

Unfortunately, as soon as a clock w features an integer larger than one, $w <:_1 w\ on\ 0(1)^\omega$ cannot hold. This implies that the program p cannot be assigned a clock type bounded by 2 or more by solving the constraint system above. Yet, local time scales can be used to circumvent this limitation. Using the RESCALE rule, one may assign any closed integer clock type ct

to p: first solve the constraint system with ct_2 = **base** as before, and then introduce a local time scale driven by ct. This is shown in Figure 5.24 (c).

This solution can be further enriched by combining local time scales with bounded quantification, giving p a polymorphic type $\forall(\alpha \leq n_d).\alpha$, for a fixed n_d. The corresponding derivation is shown in Figure 5.24 (d). Such a type may later be instantiated modularly. This shows that local time scales and clock polymorphism can be combined to achieve new type assignments.

Unbounded polymorphism The latest example may look a bit unsatisfying. While in the derivation of Figure 5.24 (b) the clock type variable α must be bounded to ensure that the fixpoint is productive, this is not the case in derivation Figure 5.24 (d): the bound n appearing in the later is arbitrary.

To express this problem in a more formal way, let us remark that derivations (a), (c) and (d) are in fact meta-derivations, since they feature a meta-variables ct and n. Thus these really represent sets of concrete derivations. The crucial property is that derivation (b) *internalizes* the meta-derivation (a) in the sense that any concrete instance of (a) can be expressed inside the type system by instantiating α with ct. In contrast, there are concrete instances of (c) that cannot be expressed by instantiating α with ct, namely those where $n_c > n_d$. In other words, (d) does not internalize (c).

It is clear that our type language is too weak to be able to internalize (c) completely, as the bound n_c on ct may be an arbitrary integer, and is thus itself unbounded. The concrete effect of this limitation is that the program p has received a type which forbids legitimate uses. More precisely, this is a case where inlining p in its caller may turn an ill-typed program into a well-typed one.

The solution to this problem should be clear: introduce *unbounded* quantification. In fact, a simple way to do so is to see the unbounded case as a special form of the bounded one, by allowing "infinite" integers ∞ to appear as bounds. Looking back at the type system, the only place where boundedness really matters is rule ADAPTON in Figure 5.16, where the bound n refines the delay k between p_1 and p_2 as $k+1-n$. Taking $n = \infty$ gives $k+1-\infty = 0$, which is indeed sound. With such "infinite" bounds, one may modify the derivation (d) to give p the type $\forall(\alpha \leq \infty).\alpha$ which now internalizes all the concrete instances of (c).

This shows that unbounded polymorphism is a sound addition to the type system that actually increases its expressiveness. The trouble is that it breaks the finite-state nature of our compilation scheme as one no longer has a static bound on the lists appearing in the machine implementing p, or on the number of steps p performs. The requirement for type-driven modular code generation here conflicts with the expressiveness of the type system. We will discuss this point in Chapter 6.

5.4 Dependent Clock Types

The last extension is the most invasive one. Up to now we have focused on languages in which (closed) clocks types denote ultimately periodic clocks. A concrete manifestation of this fact is that programs in μAS cannot exhibit data-dependent control: the convergence of the

output streams of a program only depends on the convergence of its input streams, never on their values. One may think of such programs as exhibiting a form of *uniform continuity*.

In this section we lift this restriction by considering arbitrary *dependent clock types*, as found for instance in Lucid Synchrone, in addition to the usual ultimately periodic words. This leads to a more complex system where expressions may appear in clock types. Informally, such a system makes it possible to pass, compute, or return clock types from one part of a program to another. Streams can now be computed on clocks that depend on arbitrary conditions, including inputs. As usual, the price to pay for this expressiveness is the relatively obtuse nature of the type system, which has very limited knowledge of clock types involving expressions. In particular, clock type equivalence boils down to syntactic equality of expressions in general.

The rest of this section presents this system as an extension of the polymorphic one given in the previous section. This is done in order to keep the amount of modifications as low as possible. We avoid describing its typed semantics or compilation to state machines since it is a variation on the ideas outlined in the previous section. Instead, we spend some time proving syntactic properties of the type system showing that types are well-formedness and that typing derivations respect lexical scope. We finish with a discussion of the design of the system and features that would be required in a more practical language.

5.4.1 Syntax and Untyped Semantics

The syntax of clock types in Section 5.3 had compound clock types of the form ct **on** p. Dependent clock types generalize this to ct **on** e, where e is an arbitrary expression denoting a stream of integers. As expected, this modification has a deep impact on the syntax of types and typing judgments. It also leads to changes in the syntax of expressions. They are summed up in Figure 5.25. The free variables and free type variables of clock types, types, and contexts are defined in the usual manner.

Expressions A first, minimal change is the displacement of ultimately periodic words from clock types to expressions. This makes the new clock types strictly more expressive than the old ones. Since constant streams are a special case of periodic words, we remove them for the grammar. Their new typing rule will be a strict generalization of the CONST one given in the previous section. Another related modification is the removal of the fixed periodic word p from the syntax of the stream merging and sampling operators. Using dependent types and local time scales the typing rules for these operators can be internalized. However, to be able to do so for stream fusion we need a binary negation operator `not` on streams. It is assumed to belong to the set of operators *op*.

The last change to the syntax of expressions is a purely technical one. We add explicit abstraction and application of clock type variables. These constructs explicitly mark where generalization and instantiation of polymorphic clock types occur. Their presence in the syntax makes sure that the free variables of an expression include the free variables of clock types it contains. This is necessary for the type system to be sound.

e	::=	v	Variable
	|	$\texttt{fun}\,(v : t).e$	Function (type parameter added)
	|	$e\ e$	Application
	|	(e, e)	Pair constructor
	|	$\texttt{let}\,(v, v) = e\ \texttt{in}\ e$	Pair destructor
	|	$\texttt{fix}\,e$	Recursive definition
	|	p	Ultimately periodic stream (added)
	|	op	Lifted stream operator
	|	$\texttt{not}$	Negation operator (added)
	|	$\texttt{merge}$	Stream merging (word parameter removed)
	|	$\texttt{when}$	Stream sampling (word parameter removed)
	|	$\uparrow_{ct} e$	Explicit local time scale (added)
	|	$\texttt{Fun}\,\alpha.e$	Explicit polymorphic abstraction (added)
	|	$e\ ct$	Explicit polymorphic application (added)
ct	::=	**base**	Base clock type
	|	α	Clock type variable
	|	ct **on** e	Compound clock type
t	::=	$\{n\} :: ct$	Bounded clocked stream type
	|	$t_1 \otimes t_2$	Product
	|	$t_1 \multimap t_2$	Function
	|	$\forall(\alpha \leq n).t$	Polymorphic clock type
	|	$\Pi(x : \{n\} :: ct).t$	Dependent product
	|	$\Sigma(x : \{n\} :: ct).t$	Dependent sum

Figure 5.25: Dependent clock types - modified syntax

Types We replace $dt :: ct$ with $\{n\} :: ct$, the type of streams of numbers belonging to $[0, n]$. It is necessary to have such bounds now that expressions of stream type appear in clock types, which are necessarily bounded. As expected, we also add dependent products (functions) and sums (pairs) to types. They make it possible to abstract over stream expressions appearing in types and thus offer new forms of modularity.

Untyped semantics The semantics of the new and modified constructs is unremarkable and can be deduced from the previous chapters. Abstraction and application of polymorphic clock types are interpreted as the identity function since they are transparent in the untyped world.

$$\boxed{ct_1 \equiv ct_2}$$

$$\frac{}{ct \equiv ct} \qquad \frac{ct_1 \equiv ct_2 \qquad ct_2 \equiv ct_3}{ct_1 \equiv ct_3} \qquad \frac{ct_2 \equiv ct_1}{ct_1 \equiv ct_2} \qquad \frac{ct_1 \equiv ct_2}{ct_1 \textbf{ on } e \equiv ct_2 \textbf{ on } e} \qquad \frac{ct_1 \equiv ct_2}{ct_1 \textbf{ on } p_1 \textbf{ on } p_2 \equiv ct_2 \textbf{ on } (p_1 \; on \; p_2)}$$

$$\frac{ct_1 \equiv ct_2}{ct_1 \equiv ct_2 \textbf{ on } (1)} \qquad \frac{ct_1 \equiv ct_2 \qquad \lceil p \rceil \leq 1}{ct_1 \textbf{ on } (not \; p) \equiv ct_2 \textbf{ on } \overline{p}}$$

Figure 5.26: Dependent clock types - type system - modified clock-type equivalence judgment

$$\boxed{\Gamma \vdash \Gamma_1 \otimes \Gamma_2}$$

$$\textsc{SepEmpty}\ \frac{}{\square \vdash \square \otimes \square} \qquad \textsc{SepContract}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \qquad \vdash t \text{ value}}{\Gamma, x : t \vdash \Gamma_1, x : t \otimes \Gamma_2, x : t}$$

$$\textsc{SepLeft}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \qquad x \notin dom(\Gamma_2) \qquad FV(t) \subseteq dom(\Gamma_1)}{\Gamma, x : t \vdash \Gamma_1, x : t \otimes \Gamma_2} \qquad \textsc{SepRight}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \qquad x \notin dom(\Gamma_1) \qquad FV(t) \subseteq dom(\Gamma_2)}{\Gamma, x : t \vdash \Gamma_1 \otimes \Gamma_2, x : t}$$

Figure 5.27: Dependent clock types - type system - modified value and splitting judgments

5.4.2 Type System

The extended type system is based on the polymorphic one presented in the previous section. This latter system was in fact designed so that the modifications necessary to handle dependent clocks are actually minimal. One of the main difference is that nearly all judgments are now mutually recursive; for instance, the clock-type well-formedness judgment depends on the expression typing judgment, and vice versa. We must also pay special attention to scoping restrictions as well as complications arising from the conjoint presence of linearity and dependent types. In particular, we force expressions appearing in clock types to be evaluated in intuitionistic contexts.

Clock type equivalence Figure 5.26 gives the new rules for the clock type equivalence judgment. These are mostly similar to the ones given in Figure 5.13, the only change being that words have been replaced with expressions in the congruence rule. Note that we still allow "algebraic" reasoning on ultimately periodic words. Without this rule, clock type equivalence would boil down to plain syntactic equality. The last rule is also new, and assumes that the operator `not` can be simplified when it applies to a binary word.

Remark 31. As usual in dependent type systems, syntactic equality of expressions should be tested modulo α-conversion. We do not reach this level of details here.

Intuitionistic contexts, values, splitting As explained above, to simplify the system we will actually restrict the expressions appearing in clocks to be valid in intuitionistic contexts. What

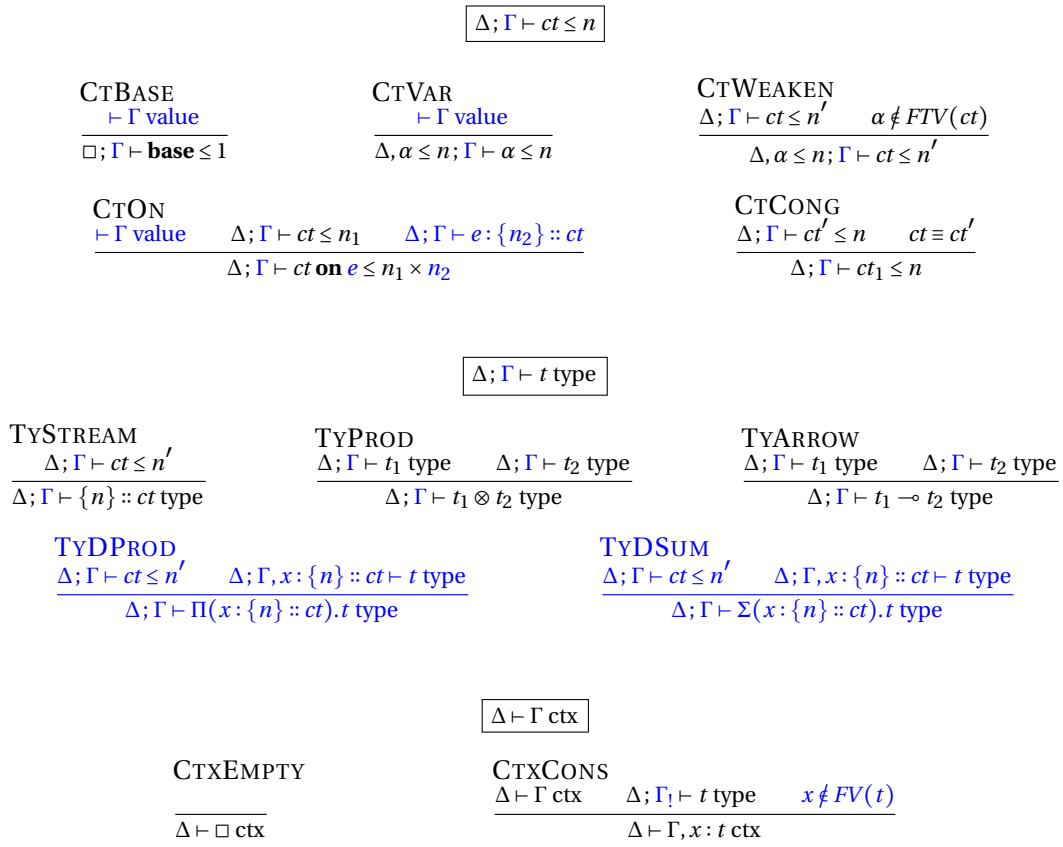


Figure 5.28: Dependent clock types - type system - modified well-formedness judgments

is an intuitionistic context? It is a context which can freely be duplicated, or in other words that is a value. The type value judgment itself is not modified; dependent products and sums are not values. This leads to the formal definition below.

Definition 21 (Largest intuitionistic subcontext)**.** *We say that* Γ' *is an* intuitionistic subcontext *of* Γ *when* $\Gamma \vdash \Gamma \otimes \Gamma'$*. The* largest *intuitionistic subcontext* $\Gamma_!$ *of* Γ *is such that any other intuitionistic subcontext of* Γ *is also an intuitionistic subcontext of* $\Gamma_!$*. This subcontext* $\Gamma_!$ *always exists and is unique.*

It is not difficult to actually compute $\Gamma_!$ from Γ since the value judgment is decidable: simply remove all the bindings to types which are not values, or that are out of scope. To make our formulas easier to read, we often write $\Gamma_{1!}$ for $(\Gamma_1)_!$, and so on.

The context splitting judgment has to be adapted to handle the presence of free program variables in types. More precisely, we add a new premise to the SepLeft and SepRight rules in order to check that we do not create ill-formed contexts and respect lexical scope. This is shown in Figure 5.27.

Well-formedness Figure 5.28 gives the extended well-formedness judgments for clock types, types and contexts. Since clock types now incorporate expressions, the first two judgments depend on the program context Γ.

In the clock type boundedness judgment, the program context is passed along in all rules but CTON. The leaf rules CTBASE and CTVAR enforce that Γ is a value. Rule CTON is the only one that changes in an interesting way. It is now a generalization of the previous version where the expression e replaces the word p. Thus, one should be able to type e in Γ and obtain the type that was previously expected of p.

The type well-formedness judgments evolves in a similar fashion. Existing rules transmit the program context to their premises. A dependent product $\Pi(x : \{n\} :: ct).t$ is well-formed just if t is well-formed in the context enriched with $x : \{n\} :: ct$, and if ct itself is well-formed. The same is true of dependent sums.

Finally, as before a context is well-formed if all the types it contains are. The only remarkable point is that we check the well-formedness of each type in an intuitionistic context derived from Γ, rather than Γ itself.

Auxiliary judgments Figure 5.29 gives the modified auxiliary judgments. As before, the changes mostly consist in transmitting to premises the previously absent context Γ. This implicitly forces Γ to be a value context, since it must be duplicable. Note that the formulation of clock type equivalence as an untyped judgment makes it possible to keep the ADAPTSTREAM, UPSTREAM and DOWNSTREAM rules as it. In the latter two rules the clocks appearing inside the local time scale have to be well-formed in the empty program context, as expected.

Main judgment Figure 5.30 gives the judgment for well-typed expressions. We have given all the rules rather than just the modified ones, which are the majority anyway. The modifications add support for dependent types; this requires the reinforcement and addition of several premises related to scoping. We also need to pass intuitionistic parts of program contexts to the auxiliary judgments. Lastly, we showcase the power of dependent types by giving nicer, more uniform types to operators `when` and `merge`. Let us describe each rule.

- Rules VAR, PAIR and VAR are the same as in Figure 5.17.
- The rules LAMBDA and LETPAIR have additional scope restrictions which prevent the variables x and y added to the context to escape through the return type of the rule.
- The rules that rely on well-formedness judgments pass them an intuitionistic part of their program contexts. This is the case for SUB, RESCALE and CKINST for instance.
- Now that type abstraction and application operators have been added the rules CKGEN and CKINST become syntax-directed.
- The rules for ultimately periodic words p, operators *op*, `not`, `merge`, and `when` are expressed using polymorphic and dependent types. Note that this was not possible for stream merging and sampling before as one needed to abstract on an actual piece of

$$\boxed{\Delta; \Gamma \vdash t_1 <:_k t_2}$$

ADAPTSTREAM
$$\frac{ct_1 \equiv ct\ \mathbf{on}\ p_1 \qquad ct_2 \equiv ct\ \mathbf{on}\ p_2 \qquad \Delta; \Gamma \vdash ct \le n \qquad p_1 <:_k p_2}{\Delta; \Gamma \vdash \{n\} :: ct_1 <:_{n+1-k} \{n\} :: ct_2}$$

ADAPTPROD
$$\frac{\vdash \Gamma \text{ value} \qquad \Delta; \Gamma \vdash t_1 <:_k t_1' \qquad \Delta \vdash \Gamma <:_{t_2} k t_2'}{\Delta; \Gamma \vdash t_1 \otimes t_2 <:_k t_1' \otimes t_2'}$$

ADAPTARROW
$$\frac{\vdash \Gamma \text{ value} \qquad \Delta; \Gamma \vdash t_1 <:_k t_1' \qquad \Delta; \Gamma \vdash t_2 <:_k t_2'}{\Delta; \Gamma \vdash t_1 \multimap t_2 <:_n t_1' \multimap t_2'}$$

$$\boxed{\Delta; \Gamma; \Gamma' \vdash ct_1 \uparrow_{ct} ct_2}$$

$$\frac{\square; \Gamma' \vdash ct_1 \le n_1 \qquad ct_2 \equiv ct_1[\mathbf{base}/ct] \qquad \Delta; \Gamma \vdash ct_2 \le n_2}{\Delta; \Gamma; \Gamma' \vdash ct_1 \uparrow_{ct} ct_2}$$

$$\boxed{\Delta; \Gamma; \Gamma' \vdash t_1 \uparrow_{ct} t_2} \text{ and } \boxed{\Delta; \Gamma; \Gamma' \vdash t_1 \downarrow_{ct} t_2}$$

UPSTREAM
$$\frac{\Delta; \Gamma; \Gamma' \vdash ct_1 \uparrow_{ct} ct_2}{\Delta; \Gamma; \Gamma' \vdash \{n\} :: ct_1 \uparrow_{ct} \{n\} :: ct_2}$$

DOWNSTREAM
$$\frac{\Delta; \Gamma; \Gamma' \vdash ct_2 \uparrow_{ct} ct_1}{\Delta; \Gamma; \Gamma' \vdash dt :: ct_1 \downarrow_{ct} dt :: ct_2}$$

UPPROD
$$\frac{\vdash \Gamma \text{ value} \qquad \vdash \Gamma' \text{ value} \qquad \Delta; \Gamma; \Gamma' \vdash t_1 \uparrow_{ct} t_1' \qquad \Delta; \Gamma; \Gamma' \vdash t_2 \uparrow_{ct} t_2'}{\Delta; \Gamma; \Gamma' \vdash t_1 \otimes t_2 \uparrow_{ct} t_1' \otimes t_2'}$$

...

UPDPROD
$$\frac{\vdash \Gamma \text{ value} \qquad \vdash \Gamma' \text{ value} \qquad \Delta; \Gamma; \Gamma' \vdash \{n\} :: ct_1 \uparrow_{ct} \{n\} :: ct_2 \qquad \Delta; \Gamma, x : \{n\} :: ct_1; \Gamma', x : \{n\} :: ct_2 \vdash t_1 \uparrow_{ct} t_2}{\vdash \Gamma \text{ value} \qquad \vdash \Gamma' \text{ value} \qquad \Delta; \Gamma; \Gamma' \vdash \Pi(x : \{n\} :: ct_1).t_1 \uparrow_{ct} \Pi(x : \{n\} :: ct_2).t_2}$$

UPDSUM
$$\frac{\Delta; \Gamma; \Gamma' \vdash \{n\} :: ct_1 \uparrow_{ct} \{n\} :: ct_2 \qquad \Delta; \Gamma, x : \{n\} :: ct_1; \Gamma', x : \{n\} :: ct_2 \vdash t_1 \uparrow_{ct} t_2}{\Delta; \Gamma; \Gamma' \vdash \Sigma(x : \{n\} :: ct_1).t_1 \uparrow_{ct} \Sigma(x : \{n\} :: ct_2).t_2}$$

$$\boxed{\Delta \vdash \Gamma \downarrow_{ct} \Gamma'}$$

DOWNCTXEMPTY
$$\frac{}{\Delta \vdash \square \downarrow_{ct} \square}$$

DOWNCTXCONS
$$\frac{\Delta \vdash \Gamma \downarrow_{ct} \Gamma' \qquad \Delta; \Gamma; \Gamma' \vdash t \downarrow_{ct} t'}{\Delta \vdash \Gamma, x : t \downarrow_{ct} \Gamma', x : t'}$$

DOWNCTXWEAKEN
$$\frac{\Delta \vdash \Gamma \downarrow_{ct} \Gamma' \qquad \vdash t \text{ value} \qquad x \notin dom(\Gamma')}{\Delta \vdash \Gamma, x : t \downarrow_{ct} \Gamma'}$$

Figure 5.29: Dependent clock types - type system - modified auxiliary judgments

$$\boxed{\Delta;\Gamma \vdash e : t}$$

$$\textsc{Var}\ \frac{\vdash \Gamma\ \text{value}}{\Delta;\Gamma, x:t \vdash x:t} \qquad \textsc{Weaken}\ \frac{\Delta;\Gamma \vdash e:t' \quad \vdash t\ \text{value} \quad x \notin FV(e) \cup FV(t')}{\Delta;\Gamma, x:t \vdash e:t'}$$

$$\textsc{Fun}\ \frac{\Delta;\Gamma_! \vdash t\ \text{type} \quad \Delta;\Gamma, x:t \vdash e:t' \quad x \notin FV(t) \cup FV(t')}{\Delta;\Gamma \vdash \mathtt{fun}\,(x:t).e : t \multimap t'} \qquad \textsc{App}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \quad \Delta;\Gamma_1 \vdash e : t \multimap t' \quad \Delta;\Gamma_2 \vdash e' : t}{\Delta;\Gamma \vdash e\ e' : t'}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \quad \Delta;\Gamma_1 \vdash e_1 : t_1 \quad \Delta;\Gamma_2 \vdash e_2 : t_2}{\Delta;\Gamma \vdash (e_1, e_2) : t_1 \otimes t_2}$$

$$\textsc{LetPair}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \quad \Delta;\Gamma_1 \vdash e : t_1 \otimes t_2 \quad \Delta \vdash \Gamma_2, x:t_1, y:t_2\ \text{ctx} \quad \Delta;\Gamma_2, x:t_1, y:t_2 \vdash e' : t \quad x, y \notin FV(t)}{\Delta;\Gamma \vdash \mathtt{let}\,(x,y) = e\ \mathtt{in}\ e' : t}$$

$$\textsc{Fix}\ \frac{\Delta;\Gamma \vdash e : t \multimap t' \quad \Delta;\Gamma_! \vdash t' <:_1 t \quad \vdash t'\ \text{value}}{\Delta;\Gamma \vdash \mathtt{fix}\ e : t'} \qquad \textsc{Pword}\ \frac{}{\Box;\Box \vdash p : \forall(\alpha \le n).\{\lceil p \rceil\} :: \alpha}$$

$$\textsc{Not}\ \frac{}{\Box;\Box \vdash not : \forall(\alpha \le n).\{1\} :: \alpha \multimap \{1\} :: \alpha} \qquad \textsc{Op}\ \frac{}{\Box;\Box \vdash op : \forall(\alpha \le n).\{dtof(op)\} :: \alpha \otimes \{dtof(op)\} :: \alpha \multimap \{dtof(op)\} :: \alpha}$$

$$\textsc{Merge}\ \frac{}{\Box;\Box \vdash \mathtt{merge} : \forall(\alpha \le n).\Pi(c:\{1\} :: \alpha).\{n'\} :: \alpha\ \mathbf{on}\ c \otimes \{n'\} :: \alpha\ \mathbf{on}\ (\mathtt{not}\ c) \multimap \{n'\} :: \alpha}$$

$$\textsc{When}\ \frac{}{\Box;\Box \vdash \mathtt{when} : \forall(\alpha \le n).\Pi(c:\{1\} :: \alpha).\{n'\} :: \alpha \multimap \{n'\} :: \alpha\ \mathbf{on}\ c} \qquad \textsc{Sub}\ \frac{\Delta;\Gamma \vdash e : t \quad \Delta;\Gamma_! \vdash t <:_k t'}{\Delta;\Gamma \vdash e : t'}$$

$$\textsc{Rescale}\ \frac{\Delta;\Gamma_! \vdash ct \le n \quad \Delta \vdash \Gamma \downarrow_{ct} \Gamma' \quad \Box;\Gamma' \vdash e : t' \quad \Box;\Gamma;\Gamma' \vdash t' \uparrow_{ct} t}{\Delta;\Gamma \vdash \uparrow_{ct} e : t} \qquad \textsc{CkGen}\ \frac{\Delta, \alpha \le n;\Gamma \vdash e : t \quad \alpha \notin FTV(\Gamma)}{\Delta;\Gamma \vdash \mathtt{Fun}\ \alpha.e : \forall(\alpha \le n).t}$$

$$\textsc{CkInst}\ \frac{\Delta;\Gamma \vdash e : \forall(\alpha \le n).t \quad \Delta;\Gamma_! \vdash ct \le n}{\Delta;\Gamma \vdash e\ ct : t[\alpha/ct]} \qquad \textsc{DFun}\ \frac{\Delta;\Gamma_! \vdash ct \le n \quad x \notin FV(ct) \quad \Delta;\Gamma, x:\{n\} :: ct \vdash e : t}{\Delta;\Gamma \vdash \mathtt{fun}\,(x:\{n\} :: ct).e : \Pi(x:\{n\} :: ct).t}$$

$$\textsc{DApp}\ \frac{\Delta;\Gamma_! \vdash e' : \{n\} :: ct \quad \Delta;\Gamma \vdash e : \Pi(x:\{n\} :: ct).t}{\Delta;\Gamma \vdash e\ e' : t[x/e']} \qquad \textsc{DPair}\ \frac{\Delta;\Gamma_! \vdash e_1 : \{n\} :: ct \quad \Delta;\Gamma \vdash e_2 : t[x/e_1]}{\Delta;\Gamma \vdash (e_1, e_2) : \Sigma(x:\{n\} :: ct).t}$$

$$\textsc{LetDPair}\ \frac{\Gamma \vdash \Gamma_1 \otimes \Gamma_2 \quad \Delta;\Gamma_1 \vdash e : \Sigma(z:\{n\} :: ct).t \quad \Delta \vdash \Gamma_2, x:\{n\} :: ct, y:t[z/x]\ \text{ctx} \quad \Delta;\Gamma_2, x:\{n\} :: ct, y:t[z/x] \vdash e' : t \quad x, y \notin FV(t)}{\Delta;\Gamma \vdash \mathtt{let}\,(x,y) = e\ \mathtt{in}\ e' : t}$$

Figure 5.30: Dependent clock types - type system - modified main judgment

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\texttt{fun}\,(x:t).e) &= (FV(e) \setminus \{x\}) \cup FV(t) \\
FV(e_1\ e_2) \;=\; FV(e_1, e_2) &= FV(e_1) \cup FV(e_2) \\
FV(\texttt{let}\,(x,y) = e\ \texttt{in}\ e') &= FV(e) \cup (FV(e') \setminus \{x,y\}) \\
FV(\texttt{fix}\,e) \;=\; FV(\texttt{Fun}\,\alpha.e) &= FV(e) \\
FV(p) \;=\; FV(op) \;=\; FV(\texttt{merge}) \;=\; FV(\texttt{when}) &= \varnothing \\
FV(\uparrow_{ct} e) \;=\; FV(e\ ct) &= FV(ct) \cup FV(e) \\
\\
FV(\mathbf{base}) \;=\; FV(\alpha) &= \varnothing \\
FV(ct\ \mathbf{on}\ e) &= FV(ct) \cup FV(e) \\
\\
FV(\{n\} :: ct) &= FV(ct) \\
FV(t_1 \otimes t_2) \;=\; FV(t_1 \multimap t_2) &= FV(t_1) \cup FV(t_2) \\
FV(\Pi(x:\{n\} :: ct).t) \;=\; FV(\Sigma(x:\{n\} :: ct).t) &= FV(ct) \cup (FV(t) \setminus \{x\})
\end{aligned}
$$

Figure 5.31: Dependent clock types - free variables in expressions, clock types, and types

program, unlike in polymorphic clocks. We will go back to this point at the end of the section. The OP rule assumes that the function $dtof(op)$ gives the type of the arguments for the operator op.

- Finally, we have actual dependent products and sums. They are endowed with two rules each, and each of these rule is an adaptation of the corresponding non-dependent rule; their name reflects this fact, with DFUN corresponding to FUN, and so on. The rules themselves are the usual ones found in dependent type theories, except that they express how dependence is restricted to intuitionistic expressions. This shows in the premise handling the argument in DAPP or the first component of the pair in DPAIR.

This concludes the description of the type system, which is the most complex part of this extension. We do not discuss its typed semantics, which can be constructed using the same techniques as in Section 5.3, at the price of much sweat. The compilation to machines is also straightforward: dependent products and sums are translated to ordinary higher-order machine types, which makes their compilation basically the same as their non-dependent counterparts. We will rather spend some time on some meta-theoretical results to prove that the typing rules enjoy good properties. This is not completely obvious given the amount of details in the handling of contexts and binders.

5.4.3 Syntactic results

Our main goal in the rest of this section is to prove two modest theorems that will increase our confidence in the fact that the type system is actually well-behaved. The first theorem expresses that the type system only assigns well-formed types. The second theorem expresses

that it respects lexical scope. Proving the latter result is actually difficult, since we have defined neither the typed nor untyped semantics precisely. We content ourselves with a partial result showing that, informally, two contexts equivalent *up to lexical scope* are basically equivalent from the point of view of typing derivations. Most intermediate proofs are straightforward inductions and thus we omit them.

Free variables We begin with some technical properties relating the free variables of clock types, types, and expressions with typing contexts appearing in the derivations of the well-formedness or typing judgments.

Property 52. *Two equivalent clock types have the same free variables.*

$$ct \equiv ct' \quad \Rightarrow \quad FV(ct) = FV(ct')$$

Property 53. *Two adaptable types have the same free variables.*

$$\Delta; \Gamma \vdash t <:_k t' \quad \Rightarrow \quad FV(t) = FV(t')$$

Property 54. *If t' is the image of the type t in a local time scale driven by ct—that is t' gathers to t or t scatters to t'—then the free variables of t are exactly those of t' and ct combined.*

$$\begin{array}{lcl} \Delta; \Gamma; \Gamma' \vdash t \uparrow_{ct} t' & \Rightarrow & FV(t') = FV(t) \cup FV(ct) \\ \Delta; \Gamma; \Gamma' \vdash t \downarrow_{ct} t' & \Rightarrow & FV(t) = FV(t') \cup FV(ct) \end{array}$$

Property 55. *The free variables of a well-formed clock type are included into the domain of its typing context, and similarly for types.*

$$\begin{array}{lcl} \Delta; \Gamma \vdash ct \leq n & \Rightarrow & FV(ct) \subseteq dom(\Gamma) \\ \Delta; \Gamma \vdash t \ type & \Rightarrow & FV(t) \subseteq dom(\Gamma) \end{array}$$

Property 56. *The free variables of both a well-typed expression and its type are included in the domain of its typing context.*

$$\Delta; \Gamma \vdash e : t \quad \Rightarrow \quad FV(e) \cup FV(t) \subseteq dom(\Gamma)$$

Note that since the well-formedness and typing judgments are mutually inductive, Properties 55 and 56 have to be proved together by mutual induction. We have stated the results separately only for clarity and easier reference.

Weakening The next properties deal with weakening. The first property shows that weakening is admissible in the well-formedness judgments for clock types and types.

Property 57. *Let t an arbitrary* value *type. Then the following properties hold.*

$$\left\{ \begin{array}{l} \Delta; \Gamma \vdash ct \leq n \\ x \notin FV(ct) \end{array} \right. \Rightarrow \Delta; \Gamma, x : t \vdash ct \leq n \qquad \left\{ \begin{array}{l} \Delta; \Gamma \vdash t' \ type \\ x \notin FV(t') \end{array} \right. \Rightarrow \Delta; \Gamma, x : t \vdash t' \ type$$

The second property shows that from any derivation done in a context $\Gamma, x : t$ with x not free in the underlying clock type, type, or expression, one can extract a derivation in Γ. This is in some sense the converse of weakening.

Property 58. *Let t be a type. Then the following properties hold.*

$$\begin{array}{lll} \left\{\begin{array}{l}\Delta;\Gamma,x:t\vdash ct\leq n\\ x\notin FV(ct)\end{array}\right. & \left\{\begin{array}{l}\Delta;\Gamma,x:t\vdash t' \ type\\ x\notin FV(t')\end{array}\right. & \left\{\begin{array}{l}\Delta;\Gamma,x:t\vdash e:t'\\ x\notin FV(e)\end{array}\right.\\ \Rightarrow\quad \Delta;\Gamma\vdash ct\leq n & \Rightarrow\quad \Delta;\Gamma\vdash t' \ type & \Rightarrow\quad \Delta;\Gamma\vdash e:t' \end{array}$$

Well-formedness We are nearly ready to state and prove the first theorem, showing that the typing judgment builds well-formed types out of well-formed contexts. Before that we need to show a last series of technical properties handling auxiliary judgments.

Property 59. *Sub-contexts of a well-formed context are well-formed.*

$$\left\{\begin{array}{l}\Delta\vdash\Gamma \ ctx\\ \Gamma\vdash\Gamma_1\otimes\Gamma_2\end{array}\right. \quad\Rightarrow\quad \Delta\vdash\Gamma_i \ ctx \text{ for all } i\in\{1,2\}$$

Conversely, a type well-formed in a sub-context is well-formed in the original context.

$$\left\{\begin{array}{l}\Delta;\Gamma_i\vdash t \ type \text{ for any } i\in\{1,2\}\\ \Gamma\vdash\Gamma_1\otimes\Gamma_2\end{array}\right. \quad\Rightarrow\quad \Delta;\Gamma\vdash t \ type$$

Property 60. *Adaptable types are well-formed.*

$$\left\{\begin{array}{l}\Delta;\Gamma\vdash t<:_k t'\\ \Delta;\Gamma\vdash t \ type\end{array}\right. \quad\Rightarrow\quad \Delta;\Gamma\vdash t' \ type$$

Property 61. *Gathered and scattered types are well-formed.*

$$\left\{\begin{array}{l}\Delta;\Gamma;\Gamma'\vdash t\uparrow_{ct} t'\\ \square;\Gamma'\vdash t \ type\end{array}\right. \quad\Rightarrow\quad \Delta;\Gamma\vdash t' \ type \qquad \left\{\begin{array}{l}\Delta;\Gamma;\Gamma'\vdash t\downarrow_{ct} t'\\ \Delta;\Gamma\vdash t \ type\end{array}\right. \quad\Rightarrow\quad \square;\Gamma'\vdash t' \ type$$

Additionally, context Scattering preserves well-formedness.

$$\left\{\begin{array}{l}\Delta\vdash\Gamma\downarrow_{ct}\Gamma'\\ \Delta\vdash\Gamma \ ctx\end{array}\right. \quad\Rightarrow\quad \square\vdash\Gamma' \ ctx$$

Property 62. *Substituting well-formed types for variables preserves well-formedness.*

$$\left\{\begin{array}{l}\Delta;\Gamma,x:\{n\}::ct\vdash t' \ type\\ \Delta;\Gamma_!\vdash e:\{n\}::ct\end{array}\right. \quad\Rightarrow\quad \Delta;\Gamma\vdash t'[x/e] \ type$$

Property 63. *Any type appearing in front of a context is well-formed in this same context.*

$$\Delta\vdash\Gamma,x:t \ ctx \quad\Rightarrow\quad \Delta;(\Gamma,x:t)_!\vdash t \ type$$

This last technical property below is actually important. It depends crucially on the fact that x cannot appear in t: observe for instance that in $x : \{1\} :: \textbf{base}, x : \{1\} :: \textbf{base on } x$ it does not hold. We can now prove the expected theorem.

Theorem 9. *Typing derivations build well-formed types out of well-formed contexts.*

$$\left.\begin{array}{r} \Delta \vdash \Gamma \ ctx \\ \Delta ; \Gamma \vdash e : t \end{array}\right\} \quad \Rightarrow \quad \Delta ; \Gamma_! \vdash t \ type$$

Proof. The proof proceeds by induction on the typing derivation. We will not delve into this rather technical detail, but rather explain informally the most interesting cases. Other cases can be proved using similar ideas, sometimes at the cost of new technical lemmas.

- Case VAR: this is exactly Property 63.
- Case WEAKEN: we know that $\Delta \vdash \Gamma$ ctx and thus $\Delta ; \Gamma \vdash t'$ type by induction. Since t is a value and $x \notin FV(t)$, we conclude $\Delta ; \Gamma, x : t \vdash t'$ type by Property 57.
- Case FUN: The fact that t is well-formed in $\Gamma_!$ and $x \notin FV(t)$ gives $\Delta \vdash \Gamma, x : t$ ctx. From this and the induction hypothesis we obtain $\Delta ; \Gamma_!, x : t \vdash t'$ type. Since $x \notin FV(t')$, it follows from Property 63 and Property 58 that t is well-formed in $\Gamma_!$.
- Case APP: from the first part Property 59 we deduce $\Delta \vdash \Gamma_{1!}$ ctx and $\Delta \vdash \Gamma_{2!}$ ctx. Thus by induction we obtain $\Delta ; \Gamma_{1!} \vdash t_1$ type and $\Delta ; \Gamma_{2!} \vdash t_2$ type. We conclude by the second part of Property 59.
- Case SUB: combine Property 60, the induction hypothesis, and Property 59.
- Case RESCALE: combine Property 61 and the induction hypothesis.
- Case DFUN: the first premises of the rule give the arguments needed to show that $\Gamma, x : \{n\} :: ct$ is well-formed in Δ. From this we obtain $\Delta \vdash \Gamma_!, x : \{n\} :: ct$ type t by induction, from which we conclude $\Delta \vdash \Gamma_!$ type $\Pi(x : \{n\} :: ct).t$ using rule TYDPROD.
- Case DAPP: this is exactly Property 62. □

Lexical context equivalence The second theorem is supposed to show that the type system respects lexical scope. As usual, since we interpret typing derivations with explicit weakening and separation this is not completely immediate, in particular now that types include expressions and thus variables. In Chapter 3 we did prove this result by showing that the typed semantics refines the untyped semantics (Theorem 6), which respects lexical scope by construction. However we cannot follow this route again since we have not described this semantics for the language given in this section.

To be able to state a version of this theorem, we introduce a notion of *lexical context equivalence*, or simply context equivalence. Informally, two contexts Γ^1 and Γ^2 are lexically equivalent up to a finite set of variables S, written $\Gamma^1 \sim_S \Gamma^2$, when they have the same rightmost

$$\textsc{Empty}\ \frac{}{\square \sim_S \square} \qquad \textsc{JunkL}\ \frac{\Gamma \sim_S \Gamma' \qquad x \notin S \qquad \vdash t \text{ value}}{\Gamma, x : t \sim_S \Gamma'} \qquad \textsc{JunkR}\ \frac{\Gamma \sim_S \Gamma' \qquad x \notin S \qquad \vdash t \text{ value}}{\Gamma \sim_S \Gamma', x : t} \qquad \textsc{Eq}\ \frac{\Gamma \sim_S \Gamma' \qquad FV(t) \subseteq S}{\Gamma, x : t \sim_{S \cup \{x\}} \Gamma', x : t}$$

Figure 5.32: Dependent clock types - lexical context equivalence

bindings for all variables in S. The precise definition of this judgment is given in Figure 5.32. Two empty contexts are equivalent up to any set. The JUNKL and JUNKR rules show that variables not belonging to S can be freely added to one context or the other. The added variable should be bound to a value type since we want lexical equivalence to preserve provability of the well-typedness judgment. Finally, the EQ rule is the most important one: it makes it possible to actually prove that a variable x belongs to S in a non-empty context, as its most recent appearances in both Γ^1 and Γ^2 bind it to the same type t. Note that Γ^1 and Γ^2 must also be equivalent modulo the free variables of t.

The lexical context equivalence relation enjoys the technical properties below. All will be used to prove the second theorem.

Property 64. *Any two contexts are equivalent up to the empty set.*

Property 65. *Context equivalence up to S is an equivalence relation, for any fixed S.*

Property 66. *Suppose $\Gamma, x : t \sim_S \Gamma'$ or $\Gamma \sim_S \Gamma', x : t$. Then $\Gamma \sim_{S \setminus \{x\}} \Gamma'$.*

Property 67. *Suppose $\Gamma, x : t \sim_S \Gamma'$ or $\Gamma \sim_S \Gamma', x : t$ with $x \notin S$. Then $\Gamma \sim_S \Gamma'$.*

Property 68. *Two contexts equivalent up to $S_1 \cup S_2$ are equivalent up to S_1 and up to S_2.*

Note that the converse property is not true. For a counterexample take $\Gamma = x : t, y : t'$ and $\Gamma' = y : t', x : t$ with $S_1 = \{x\}$ and $S_2 = \{y\}$.

Lexical scope As before, we must prove a host of technical properties showing that the auxiliary judgments of the type system play well with context equivalence.

Property 69. *Any context equivalent to a value context is itself a value context.*

$$\left.\begin{array}{r} \vdash \Gamma^1 \textit{ value} \\ \Gamma^1 \sim_S \Gamma^2 \end{array}\right\} \Rightarrow \vdash \Gamma^2 \textit{ value}$$

Property 70. *A derivation of the separation judgment in a certain context can be transformed into a derivation in any context equivalent to the first one.*

$$\left.\begin{array}{r} \Gamma^1 \vdash \Gamma^1_1 \otimes \Gamma^1_2 \\ \Gamma^1 \sim_S \Gamma^2 \end{array}\right\} \Rightarrow \exists \Gamma^2_1, \Gamma^2_2. \left\{\begin{array}{l} \Gamma^2 \vdash \Gamma^2_1 \otimes \Gamma^2_2 \\ \Gamma^1_i \sim_S \Gamma^2_i \textit{ for all } i \in \{1,2\} \end{array}\right.$$

Conversely, if two pairs of subcontexts are equivalent up to S_1 and S_2, their parent contexts are equivalent up to $S_1 \cup S_2$.

$$\left.\begin{array}{r} \Gamma^i \vdash \Gamma^i_1 \otimes \Gamma^i_2 \text{ for all } i \in \{1,2\} \\ \Gamma^1_i \sim_{S_i} \Gamma^2_i \text{ for all } i \in \{1,2\} \end{array}\right\} \Rightarrow \Gamma^1 \sim_{S_1 \cup S_2} \Gamma^2$$

Property 71. *The largest intuitionistic subcontexts of two contexts equivalent up to some S are themselves also equivalent up to S.*

$$\Gamma^1 \sim_S \Gamma^2 \Rightarrow \Gamma^1_! \sim_S \Gamma^2_!$$

Property 72. *A derivation of the clock-type well-formedness judgment for ct in a certain context can be transformed into a derivation in any context equivalent to the first one up to the free variables of ct. The bound is conserved.*

$$\left.\begin{array}{r} \Delta; \Gamma^1 \vdash ct \leq n \\ \Gamma^1 \sim_{FV(ct)} \Gamma^2 \end{array}\right\} \Rightarrow \Delta; \Gamma^2 \vdash ct \leq n$$

Property 73. *A derivation of the type well-formedness judgment for t in a certain context can be transformed into a derivation in any context equivalent to the first one up to the free variables of t.*

$$\left.\begin{array}{r} \Delta; \Gamma^1 \vdash t \text{ type} \\ \Gamma^1 \sim_{FV(t)} \Gamma^2 \end{array}\right\} \Rightarrow \Delta; \Gamma^2 \vdash t \text{ type}$$

Property 74. *A derivation of the adaptability judgment from t to t' in a certain context can be transformed into a derivation in any context equivalent to the first one up to the free variables of t (or t' since they are the same).*

$$\left.\begin{array}{r} \Delta; \Gamma^1 \vdash t <:_k t' \\ \Gamma^1 \sim_{FV(t)} \Gamma^2 \end{array}\right\} \Rightarrow \Delta; \Gamma^2 \vdash t <:_k t'$$

Property 75. *A derivation of the scattering judgment from t to t' in certain pair of contexts can be transformed into a derivation into any pair of contexts equivalent to the first one up to the free variables of t for the first component and t' for the second. Similarly for gathering.*

$$\left.\begin{array}{r} \Delta; \Gamma_1; \Gamma'_1 \vdash t \downarrow_{ct} t' \\ \Gamma_1 \sim_{FV(t)} \Gamma_2 \\ \Gamma'_1 \sim_{FV(t')} \Gamma'_2 \end{array}\right\} \Rightarrow \Delta; \Gamma_2; \Gamma'_2 \vdash t \downarrow_{ct} t' \qquad \left.\begin{array}{r} \Delta; \Gamma_1; \Gamma'_1 \vdash t \uparrow_{ct} t' \\ \Gamma_1 \sim_{FV(t')} \Gamma_2 \\ \Gamma'_1 \sim_{FV(t)} \Gamma'_2 \end{array}\right\} \Rightarrow \Delta; \Gamma_2; \Gamma'_2 \vdash t \uparrow_{ct} t'$$

A similar property also holds for scattered contexts. For any clock type ct and finite set of variables S, the following property holds.

$$\left.\begin{array}{r} \Delta \vdash \Gamma_1 \downarrow_{ct} \Gamma'_1 \\ \Gamma_1 \sim_S \Gamma_2 \end{array}\right\} \Rightarrow \exists \Gamma'_2 . \left\{\begin{array}{l} \Delta; \Gamma_2; \Gamma'_2 \vdash t \downarrow_{ct} t' \\ \Gamma'_1 \sim_S \Gamma'_2 \end{array}\right.$$

The above properties express that auxiliary judgments respect the context equivalence relation in some sense. Thanks to them we can finally prove the second theorem of this section, which is the analogue of these properties for the main typing judgment.

Theorem 10. *Given a typing derivation for $e : t$ in a context Γ^1 and a context Γ^2 equivalent to Γ^1 up to the free variables of e and t, one may build a typing derivation for $e : t$ in Γ^2.*

$$\left.\begin{array}{r} \Delta; \Gamma^1 \vdash e : t \\ \Gamma_1 \sim_{FV(e) \cup FV(t)} \Gamma^2 \end{array}\right\} \Rightarrow \quad \Delta; \Gamma^2 \vdash e : t$$

Proof. We proceed by induction over the derivation.

- Case VAR: we have $\Gamma^1, x : t \sim_{\{x\} \cup FV(t)} \Gamma^2$. We proceed by induction over this equivalence.
  - Case EMPTY: absurd.
  - Case JUNKL: absurd since $x \in \{x\}$.
  - Case JUNKR: we have $\Gamma^2 = \Gamma^{2\prime}, y : t'$ with $x \neq y$, t' a value type and $y \notin FV(t)$, together with $\Gamma^1, x : t \sim_{\{x\} \cup FV(t)} \Gamma^{2\prime}$. From the latter property and the (inner) induction hypothesis we obtain a derivation of $\Delta; \Gamma^{2\prime} \vdash x : t$. Since $y \notin FV(x) \cup FV(t)$ and t' is a value type, we may apply rule WEAKEN to derive $\Delta; \Gamma^2 \vdash x : t$.
  - Case EQ: we have $\Gamma^2 = \Gamma^{2\prime}, x : t$ with $\Gamma^1 \sim_{FV(t)} \Gamma^{2\prime}$. By Property 69 we know that $\Gamma^{2\prime}$ is a value context, and can thus derive $\Delta; \Gamma^{2\prime}, x : t \vdash x : t$ through the VAR rule.
- Case WEAKEN: we have $\Gamma^1, x : t \sim_{FV(e) \cup FV(t')} \Gamma^2$. From the premise $x \notin FV(e) \cup FV(t')$ and Property 67 we know that $\Gamma^1 \sim_{FV(e) \cup FV(t')} \Gamma^2$. The induction hypothesis thus gives a derivation of $\Delta; \Gamma^2 \vdash e : t$. This is enough to conclude.
- Case LAMBDA: we have $\Gamma^1 \sim_{(FV(e) \setminus \{x\}) \cup FV(t) \cup FV(t')} \Gamma^2$. Using the context equivalence rule EQ we deduce $\Gamma^1, x : t \sim_{FV(e) \cup FV(t')} \Gamma^2, x : t$. The induction hypothesis thus gives a derivation of $\Delta; \Gamma^2, x : t \vdash e : t'$. We also have a derivation of $\Delta; \Gamma^2 \vdash t$ type by Property 73. From this we can reapply LAMBDA to derive $\Delta; \Gamma^2 \vdash \texttt{fun}\,(x : t).\, e : t \multimap t'$.
- Case APP: straightforward application of Property 70 and Property 68 combined with the induction hypothesis.
- Case PAIR: similar to APP.
- Case LETPAIR: similar to a combination of FUN and APP.
- Case FIX: we have $\Gamma^1 \sim_{FV(e) \cup FV(t) \cup FV(t')} \Gamma^2$. By Property 71 and Property 68, $\Gamma_{1!} \sim_{FV(t')} \Gamma_{2!}$. We derive $\Delta; \Gamma_{2!} \vdash t$ type by Property 73. On the other hand the induction hypothesis gives $\Delta; \Gamma_2 \vdash e : t$. From this we can conclude using rule FIX.
- Case PWORD, NOT, OP, MERGE, WHEN: immediate.
- Case SUB: we have $\Gamma^1 \sim_{FV(e) \cup FV(t')} \Gamma^2$. By Property 53 we have $FV(t) = FV(t')$ and we thus apply the induction hypothesis to derive $\Delta; \Gamma^2 \vdash e : t$. By Property 74 we derive $\Delta; \Gamma^2 \vdash t <:_k t'$. We can thus derive $\Delta; \Gamma^2 \vdash e : t'$ using rule SUB.

- Case RESCALE: we have $\Gamma^1 \sim_{FV(e)\cup FV(ct)\cup FV(t)} \Gamma^2$. By Property 75 there exists a context $\Gamma^{2\prime}$ such that $\Delta \vdash \Gamma^2 \downarrow_{ct} \Gamma^{2\prime}$ with $\Gamma^{1\prime} \sim_{FV(e)\cup FV(t)} \Gamma^{2\prime}$. By Property 54 we know that $FV(t') \subseteq FV(t)$, we have $\Gamma^{1\prime} \sim_{FV(e)\cup FV(t')} \Gamma^{2\prime}$ and can thus obtain a derivation of $\square; \Gamma^{2\prime} \vdash e : t'$ by induction. Thus by Property 75 again there exists a derivation of $\Delta; \Gamma^2; \Gamma^{2\prime} \vdash t' \uparrow_{ct} t$, from which we derive $\Delta; \Gamma^2 \vdash e : t$ by RESCALE.

- Case CKGEN: immediate by induction.

- Case CKINST: straightforward application of Property 71 and the induction hypothesis.

- Case DFUN: similar to FUN.

- Case DAPP: similar to APP.

- Case DPAIR: similar to DFUN and PAIR.

- Case LETDPAIR: similar to DAPP and LETPAIR. □

As for the first theorem, strictly speaking all the previous properties as well as the theorem must be proved simultaneously by mutual induction. This concludes this preliminary study of the meta-theoretic properties of dependent clock types.

Remark 32. Observe how in the proof of Theorem 10 the derivation we build only differs from the original one by the occurrences of the weakening rule. We could make this notion formal as another equivalence relation, this time between derivations of the main typing judgment. Two derivations would be equivalent if their conclusion holds in equivalent contexts and they only differ by the amount and positions of the weakening rule. Armed with such a definition we could state and prove a result showing that any two derivations in equivalent contexts are themselves equivalent. This only holds because the system of Figure 5.30 is syntax-directed.

5.4.4 Discussion

We finish this section with a high-level discussion of dependent clock types and their advantages as well as limitations. First we give some examples of applications and the added expressiveness of the system compared to the previous ones in this thesis. Then we briefly compare polymorphic and dependent clock types, showing how they are complementary features. We finish with a comparison of our system with the one present in Lucid Synchrone.

Usage As alluded to in the beginning of this section, dependent clock types makes it possible to express data-dependent sampling and merging conditions. In particular, dependent products makes it possible to write functions whose output streams have clocks depending on an input. This makes them strictly more expressive than the systems featuring only ultimately periodic words and polymorphic clock types. The example below shows a simple situation exploiting this newfound expressiveness.

We start with a function that does not involve dependent clock types but is polymorphic. This function *csum* given below computes the cumulative sum of its input, a stream of 8-bit integers. To make its definition more readable we omit clock type abstractions and applications and use constants rather than ultimately periodic words.

$$\begin{array}{lcl} \mathit{csum} & : & \forall(\alpha \leq 1).\{255\} :: \alpha \multimap \{255\} :: \alpha \\ \mathit{sum} & = & \texttt{fun}\,(\texttt{x} : \{255\} :: \alpha).\texttt{fix}\,(\texttt{fun}\,(\texttt{o} : \alpha\ \mathbf{on}\ 0(1)).\texttt{x} + \texttt{merge}\ 1(0)\ 0\ \texttt{o}) \end{array}$$

Using *sum* we now define a generic *sampled cumulative sum* function *scsum* that receives a stream of booleans and a stream of bytes, and produces a stream of bytes. It simply computes the cumulative sum of its second argument sampled by its first argument.

$$\begin{array}{lcl} \mathit{scsum} & : & \Pi(\texttt{c} : \{1\} :: \mathbf{base}).\{255\} :: \mathbf{base} \multimap \{255\} :: \mathbf{base}\ \mathbf{on}\ \texttt{c} \\ \mathit{scsum} & = & \texttt{fun}\,(\texttt{c} : \{1\} :: \mathbf{base})(\texttt{x} : \{1\} :: \mathbf{base}).\,\mathit{csum}\,(\texttt{when c x}) \end{array}$$

Dependent clock types can be combined with previously introduced features to old types in new ways. For instance, imagine that we want to compute the sampled cumulative sum of a stream with a fixed, ultimately periodic sampling pattern, with some change of scale applied. The function *oddsum* computes the cumulative sum of the elements of its input of odd rank. It introduces a local time scale to actually compute one element of output per time step, at the price of consuming its input twice as fast.

$$\begin{array}{lcl} \mathit{oddsum} & : & \{255\} :: \mathbf{base}\ \mathbf{on}\ (2) \multimap \{255\} :: \mathbf{base} \\ \mathit{oddsum} & = & \uparrow_{\mathbf{base}\ \mathbf{on}\ (2)}\ (f\ (0\ 1)) \end{array}$$

Polymorphism and dependence From the type system design point of view, polymorphic and dependent clocks are orthogonal: one may perfectly have one without the other. The system presented in this section offers both to show that it is not difficult to combine them.

The key difference between dependent products and clock quantification is that the latter only exists at the type level. Indeed, rule DFUN types lambda-abstractions appearing in the "raw" source code, and thus characterizes untyped behavior. In particular, dependent products may only be introduced when the program syntax contains a function. In contrast, rule CKGEN is syntax-directed only for technical reasons and its untyped semantics is the identity function. In other words, clock quantification can be introduced (and eliminated) at any point. The fact that polymorphism only makes sense at the typed level explains why clock type variables belong to their own namespace, disjoint from the one of program variables.

This difference between the two constructs should have an impact on the design of a practical languages based on the ideas of this thesis. While dependent types should be introduced by (dedicated?) binders written by the programmer, we believe that polymorphic clock types should probably be inferred by default, as in ML-like languages.

Comparison with Lucid Synchrone As alluded to in the beginning of this section, this is not the first proposal for a synchronous language with dependent clocks. Indeed, we are indebted to the work on Lucid Synchrone [Caspi and Pouzet, 1996]. The main difference between the

system presented in this section and the one proposed by Caspi and Pouzet—apart from the orthogonal fact that we handle integer clocks—is our close adherence to the usual metatheory of dependent types à la Martin-Löf, as well as the presence of linearity. In contrast, the solution proposed in Caspi and Pouzet [1996] and its variations implemented in the successive versions of the Lucid Synchrone compiler is more practical. In particular, the authors adapt an idea originally proposed by Laufer and Odersky for adding abstract data types to ML in order to recover type inference in a robust manner. Our proposal is a *calculus* rather than a *language*: it is regular and relatively simple, but the questions remains as to whether it is usable and implementable in practice.

Chapter 6

Perspectives

This last chapter concludes the thesis with a general discussion of the links between our work and the broader world of programming languages. Section 6.1 opens the chapter with a discussion of related works, focused mainly on the links with existing synchronous and functional languages. We briefly touch upon the connection with models for the cyclic scheduling of streaming systems such as the so-called Synchronous Dataflow Graphs. Section 6.2 describes what we believe are the most important questions this thesis has given rise to. Some of them are related to the practical issues a usable language based on integer clocks and higher-order linearity would have to face. Others are more theoretical and could lead to a better understanding of the intrinsic nature of clocks. The practical and theoretical problems are closely intertwined. Finally, Section 6.3 concludes this final chapter by a discussion of the lessons we have learned during the writing of this manuscript, as well as long-term goals for this line of research.

6.1 Related Work

6.1.1 Functional Synchronous Languages

The languages presented in this thesis all belong to the family of synchronous functional languages originating from Lustre [Caspi et al., 1987]. They share the common characteristics of being stream-oriented domain-specific languages compiled to finite-state machines. We give a short account of their history before explaining the benefits of our proposal over the state of the art.

Lustre Lustre [Caspi et al., 1987] borrows from the Lucid dataflow language of Ashcroft and Wadge [1977] the idea of streams processing. However, the original Lucid requires a complicated implementation strategy to deal with the fact that it manipulates infinite streams in an unrestricted way. This makes it unfit for critical systems. Lustre restricts Lucid to synchronous stream functions, where the elements of rank i in output streams depend *at most* on the elements of rank i in input streams. This is checked by what is to our knowledge the

first instance of a clock calculus, introduced in the same paper; clocks were later studied in depth by Caspi [1992]. The compiler also enforces that all definitions are productive (absence of deadlock) by verifying that every recursive stream definition is guarded by a constructor, which in Lustre corresponds to the delay operator.

Lustre is the ancestor of AcidS. It is a mature language usable in practice which offers useful and pragmatic features that our development lacks, such as a macro-like system for static programming (including static recursion) or integration with model-checking tools. Its static analyses are not expressed as type-like systems, and are inherently non-modular. The compilation of Lustre is traditionally viewed as a global process that compiles a complete program at once [Halbwachs et al., 1991]. In contrast we insist on separate compilation, modularity, and explain code generation in a type-directed manner.

Lucid Synchrone While Lustre introduced several new ideas and techniques, its expressiveness is relatively low compared to mainstream functional languages. To remediate this fact, Caspi and Pouzet [1996] introduce Lucid Synchrone, a synchronous functional language bringing Lustre closer to the ML tradition through the addition of higher-order functions and pattern matching. The language was then extended over the years with new high-level constructs such as hierarchical state machines [Colaço et al., 2005] and signals [Colaço et al., 2006]. Another specificity is the formulation of all static analyses needed for the compilation of synchronous programs as type systems, such as the clock calculus [Colaço and Pouzet, 2003], causality analysis [Cuoq and Pouzet, 2001], and initialization analysis [Colaço and Pouzet, 2002]. The latter checks that the unspecified elements added to streams by uninitialized delay operators do not influence the results of computations.

In contrast with Lucid Synchrone, AcidS is purely geared towards stream processing and lacks the constructs used for writing control-dominated code in Lucid Synchrone, such as hierarchical automata or signals. These constructs ultimately rely on the presence of a modular *reinitialization* operator, which we do not know how to handle; we discuss this operator in the next section. On the other hand, the type system of AcidS is arguably simpler than the conjunction of the clocking, initialization, and causality type systems of Lucid Synchrone. The simplicity of the whole thing makes it possible to give a full formal description of the compilation process, from the untyped semantics to code generation, in a handful of pages. To our knowledge no similar treatment for Lucid Synchrone has appeared.

Another difference lies in the handling of higher-order functions. In Lucid Synchrone unrestricted higher-order functions are available, in contrast with even the most expressive systems in Chapter 5. The price to pay is that Lucid Synchrone compiles to OCaml and relies on its dynamic memory management and garbage collection facilities. In our proposal linearity makes it possible to reconcile higher-order functions with static memory usage. A last difference pertaining to functions is that in Lucid Synchrone one may send a function over a signal, which in effect creates a pseudostream of functions. The streams of AcidS are restricted to scalars or first-order data types. Note however that in Lucid Synchrone the free variables of a function transmitted over a signal are forced to be constant.

Finally, and as expressed in Chapter 4, it is possible to understand the clock-directed

compilation scheme of Biernacki et al. [2008] as the composition of a source-to-source pass, which introduces new binary local time scales as guards, and normal software code generation. This shows that local time scales, even when not explicitly exposed to the programmer, are actually useful to streamline the compilation process. Expressing the generation of guards as a source-to-source transformation might help with formal proofs of compiler correctness.

n-synchrony and Lucy-n Lucid Synchrone is a relatively large language. In its third incarnation [Pouzet, 2006], it is probably the richest synchronous functional language available. Yet, it shares with Lustre the preeminence of the delay operator, which is a one-place buffer (register) with a programmer-specified initial value.[1] This operator is trivial from the clock point of view: its typing rule asks for its inputs and outputs to have the same clock type. Thus, in Lustre and Lucid Synchrone, clock types enforce that no *implicit* buffering happen.

Of course, programmers may use the delay operator to implement more complex buffering behaviors by hand. In practice however, since delays and complex clock types do not mix well, streams computed according to complicated conditions are frequently filled with "junk" values so that their clock stays simple. The programmer then has to know the exact rank of actually valid elements and sample or buffer the resulting stream accordingly.

The work on n-synchrony [Cohen et al., 2006] arises from the observation that this programming style is too low-level for some programs. In particular, multimedia processing frequently involves periodic computations for which it is very error-prone to implement complex buffers by hand. In n-synchronous language clock types reveal quantitative information about clocks so that the compiler can actually decide whether it is possible to buffer a stream into another. Lucy-n [Mandel et al., 2010] implements the n-synchronous point of view in a simplified variant of Lucid Synchrone endowed with an explicit buffering construct. Its compiler implements sophisticated clock type inference algorithms [Cohen et al., 2008; Mandel and Plateau, 2012]. Another example of a language that fits within the n-synchronous framework is Prelude [Forget et al., 2008], which is from the clock typing point of view an interesting special case of Lucy-n.

Lucy-n is the closest relative to the languages present in this thesis and several key concepts of AcidS, such as clock adaptability, originate from its metatheory. We believe that the main conceptual contributions of Lucy-n are the following. First, it introduced the idea that clock types were not only a tool to reject programs but also made it possible to drive the code generation process, an idea that is instrumental in the design of integer clocks and local time scales. Second, it managed to separate the orthogonal concepts of *initialization* and *buffering* which were tied together by the delay operator in Lustre or Lucid Synchrone. This has the immediate benefit that the dedicated initialization analysis of Lucid Synchrone disappears, as it is now completely subsumed by clock typing. This separation also paves the way for the clock-based handling of causality enabled by local time scales.

From a technical point of view, as a language Lucy-n is in fact equivalent to the first-order and binary fragment of AcidS, with the node and clock polymorphism extensions of Chapter 5, and with an explicit Curry-style construct for buffering. It is important that while the presence of an explicit buffering construct is probably important in practice, it makes no difference

[1] The initial value may be the special value *nil* in the case of so-called uninitialized delays.

for the metatheoretical point of view. An important part of the work on Lucy-n has focused on the design of usable algorithms for clock type inference [Cohen et al., 2008; Mandel and Plateau, 2012; Plateau, 2010]. We have completely neglected this aspect up to now, and will discuss it at length in the next section. Also, code generation for Lucy-n, while supposedly a straightforward variation of the traditional clock-directed scheme [Biernacki et al., 2008], had never been actually fleshed out. Chapter 4 may now serve as a reference for this point.

6.1.2 Other Synchronous Languages

In this thesis we have only studied functional synchronous languages in the vein of Lustre. Let us still say a word of the synchronous languages that do not belong to this tradition.

Extrinsic and intrinsic synchrony The two other original synchronous languages are Esterel [Berry and Gonthier, 1992] and Signal [Le Guernic et al., 1991]. Esterel, in its first incarnation, is an imperative and concurrent language whose original strength is in the implementation of control-dominated code. Its most recent version, Esterel v7 [Esterel Technologies, 2005], is much more expressive. For example, it includes as a sublanguage the control-free fragment of Lustre.[2] We discuss its compilation to circuits, which has been thoroughly studied, in the next paragraph. Signal, while closer to Lustre than the original Esterel, is more expressive since Signal programs describe *relations* rather than *functions* on streams. The price to pay for this increased flexibility is that code generation becomes more difficult.

In our opinion, an important difference between Esterel and Signal on the one hand, and recent dialects of Lustre on the other, is that the former assumes an *intrinsic* notion of time, while in the latter it is an *extrinsic* notion.[3] In both Esterel and Signal, time is explicit in programs and dealt with directly by the programmer. For instance, in Esterel one may write "*pause*" statements and react to the absence of signals, and in Signal relations may make the values produced by a program depend on clocks. Contrast with Lucid Synchrone, Lucy-n, or AcidS, in which it is impossible for a program to observe the clock of a stream. In these three languages, the reaction at which a given computation occurs is *entirely* and *uniquely* determined by clock typing, and hence is external, or extrinsic. This is especially visible in a language such as AcidS, where distinct typing derivations for the same program may lead to very different temporal behaviors. Interestingly, the control-free fragment of Lustre admits both intrinsic and extrinsic interpretations.

Whether time is intrinsic or extrinsic has a deep impact on a synchronous language. From the theoretical side, languages with extrinsic time tend to take a time-free semantics as the reference one; this is the untyped semantics of Chapter 3 or the so-called "Kahn" semantics of Pouzet [2006] or Mandel et al. [2010]. From the practical side, extrinsic time makes clock inference much more feasible. In a language with clocks but intrinsic time such as Signal, assigning different clock types to the same program may lead to unrelated final results. Moreover,

[2] We call "control-free" the fragment of Lustre without stream sampling or merging.

[3] Strictly speaking, time in the original Lustre is also intrinsic because of the `current` operator. This construct has been removed from recent Lustre dialects such as SCADE6 or Heptagon.

being able to react instantaneously to the absence of a signal complicates the generation of efficient distributed code [Potop-Butucaru et al., 2006]. Yet, in some situations intrinsic time is more appropriate and natural; this is certainly the case when programming control-dominated synchronous circuits, for instance.

To conclude, we would like to emphasize that programs written in extrinsic synchronous languages can be thought of as *parametric* over the underlying time scale: the result of execution is essentially unaffected by changes in computation rates, as expressed formally by Theorem 6. This is why a construct such as local time scales could be introduced. Some authors assert that the ability to react to absence, which strongly weakens this form of parametricity, is part of the very essence of synchrony; witness for instance the following definition, extracted from Benveniste et al. [2000]. Emphasis is ours.

> There have been several attempts to characterize the essentials of the synchronous paradigm. With some experience, we feel that the following features are indeed essential and sufficient for characterizing this paradigm: 1/ Programs progress via an infinite sequence of reactions [...] 2/ Within a reaction, *decisions can be taken on the basis of the absence of some events* [...] 3/ Communication is performed via instantaneous broadcast.

If this is indeed the case, then we believe that extrinsic synchronous languages ought to be called *synchronized* rather than *synchronous* languages.

Synchronous languages and circuits The close match between the original synchronous languages and digital synchronous circuits makes the idea of using the former to program the latter very natural, and this thesis takes another step in this general direction. However, we stress that the present work only studies a relatively small part of the problem, centered around an abstract view of circuits as stream processors. In particular, we have neglected interesting and difficult problems that arise in a practical language for circuits, such as language design, optimization, or verification questions. For example, AcidS currently lacks some high-level linguistic constructs found in other synchronous languages, such as hierarchical state machines, preemption, or reinitialization. We rather focus on the exploration of space/time trade-offs at the language level, which has perhaps been less explored in synchronous languages, as remarked by Berry [2007, Section 5].

Let us briefly mention related work addressing the design and implementation of synchronous languages dedicated to circuits. On the Lustre side, researchers have mostly focused on the generation of efficient software code, with the exception of Rocheteau [1992], as already discussed in Chapter 4. In contrast, the use of Esterel in hardware design has been thoroughly investigated, including specific language extensions (e.g., [Berry and Sentovich, 2001]), optimization of the generated circuits (e.g., [Sentovich et al., 1996]), and formal verification. The seventh version of the language [Esterel Technologies, 2005] combines these works and others to offer a rich circuit design environment which is able to describe both data paths and control logic. Berry et al. [2003] demonstrate its use for the design of a system mixing hardware and

software. The current reference for the compilation of Esterel is the already mentioned book by Potop-Butucaru et al. [2007].

Local time scales in other languages Benveniste et al. [1992] propose a denotational semantics based on streams and clocks for Signal programs. While their definition of clocks is restricted to binary ones, since they are defined as strictly-increasing index functions, they introduce a *multiplexing* operation that is similar to the post-composition with an integer clock. The multiplexing of a clock by an integer signal creates a new clock which intuitively goes "faster" than the original one. The theory is complicated by the fact that integer signals are not clocks, and that clocks are defined on varying time bases, which the multiplexing operator changes. Allowing arbitrary integers in clocks as in this thesis makes the theory more uniform, as everything can be expressed using clock composition. In particular, it appears that Theorem 2 in [Benveniste et al., 1992, Section 4.3.3] boils down to the fact that any clock can be expressed uniquely as the composition of a strictly positive and binary one. Additionally, as far as we know multiplexing was not conceived as an operation to be performed on programs in addition to clocks. To the best of our knowledge, it was never actually put into practice as a language feature, and the connections with loop generation and space-time trade-offs were not made.

Another related line of thought is the introduction of *time refinement* in the imperative synchronous language Quartz [Gemünde et al., 2013] and in the functional reactive language ReactiveML [Mandel et al., 2013]. The latter work was a direct inspiration for local time scales. ReactiveML [Mandel and Pouzet, 2005] is a higher-order language mixing ML-style programming with the reactive model of Boussinot and De Simone [1996]. In broad strokes, the reactive model of Boussinot is a variation on Esterel where instantaneous reaction to absence is forbidden. In such a language all programs are causal by construction. This makes it easier to add dynamic process creation and termination. Time refinement was added to ReactiveML by Mandel et al. [2013] and provides a clock domain construct that is very much like our local time scale. A clock domain is a piece of code that goes faster than the outside world. In accordance with the philosophy of the language, clock domains are completely dynamic. The amount of local steps a domain performs for a given global step is determined by the code running *inside* the domain, in contrast with local time scales. No attempt is made to statically schedule programs or to check that they can be implemented within bounded memory. This means in particular that clock domains scales are not guaranteed to cooperate. The proposed type system still enforces two safety properties. First, signals defined locally to a domain cannot escape to the outside world. Second, the code running inside a domain may not react instantaneously to signals defined in the slower global time scale. This latter issue is not a problem in our setting.

6.1.3 Dataflow Languages

All synchronous languages involve a notion of discrete time but differ in whether they are oriented towards stream processing. Let us now consider the other side of the spectrum, where

programming is based on streams or similar lazy data structures, but where the notion of time plays no important role, the so-called *dataflow* languages.

StreamIt We have already evoked Lucid [Ashcroft and Wadge, 1977], one of the original dataflow languages, and its influence on Lustre and its descendants—including AcidS. A more recent dataflow language is StreamIt [Thies et al., 2002]. While its authors conceived Lucid as a general purpose programming language and were intent on replacing conventional imperative programming [Wadge and Ashcroft, 1985], StreamIt is fully devoted to the programming of high-performance streaming applications such as multimedia processing. The idea is that a domain-specific language makes programming easier but still exposes more information to the compiler than traditionally available. This information should enable optimizations that are in practice out of reach of compilers for low-level languages such as C, such as whole-program automatic parallelization.

StreamIt is a two-level language. The top-level part of StreamIt is a declarative description of the structure of a program, which is an assemblage of processes called *actors*. Technically, StreamIt program gives rise to a *Synchronous Dataflow Graph* (SDF) of Lee and Messerschmitt [1987], or more precisely to a *Cyclo-Static Dataflow Graph* (CSDF) of Bilsen et al. [1996]. We will discuss these models in a moment, for now let us only say that (C)SDF graphs are special cases of Kahn process networks where communication information is statically known. The low-level part of StreamIt describes the internals of each actor. An actor is defined by a piece of first-order imperative code for its transition function together with the description of its private, internal state. The imperative code may feature loops and arrays. The transition function is annotated with periodic production and consumption information for all its inputs and outputs. Such a program is analyzed by the StreamIt compiler, scheduled, and translated into sequential or parallel code.

StreamIt and AcidS both describe computation rhythms of as periodic words, and use this information to generate code. The StreamIt authors have focused on the optimization and scheduling problems posed by its compilation [Thies, 2009; Gordon, 2010], which are invisible to the programmer. In contrast, we have focused on the orthogonal matter of devising a precise language for schedules in the guise of integer clocks and local time scales. This language should serve as an interface between the programmer and the compiler, which is necessary in a modular compilation setting. Another difference is that our language is safe: well-typed programs cannot encounter runtime errors. The StreamIt compiler does not attempt to check that the imperative code of an actor respects its declared production and consumption rates. Finally, one could say that StreamIt is in a sense lower-level: many benchmarks in the StreamIt distribution have actors with non-trivial transition functions that, in a sense, have been scheduled manually by the programmer. As a consequence, idiomatic StreamIt programs tend to directly expose less dependencies to the compiler than AcidS programs.

Remark 33. The work of Thies et al. on the phased scheduling of StreamIt programs [Thies et al., 2002] looks for schedules in which some activations have been regrouped in order to improve code size while minimizing buffering requirements. The resulting schedules bear a certain resemblance to ultimately periodic integer clocks and generate nested loops.

Model	Authors	Communication patterns
Synchronous Dataflow (SDF)	Lee and Messerschmitt [1987]	Constant words
Cyclo-Static Dataflow (CSDF)	Bilsen et al. [1996]	Strictly periodic words
Thresholded Cyclo-Static Dataflow (TCSDF)	Bodin [2013]	Ultimately periodic words
Scenario-Aware Dataflow (SADF)	Theelen et al. [2006]	Markov chains

Table 6.1: Some scheduled dataflow models

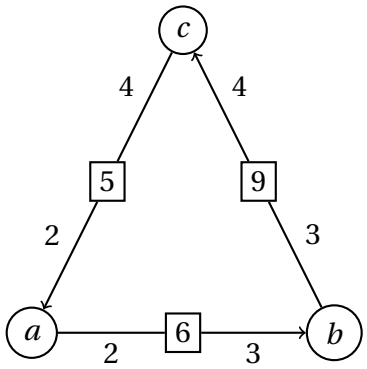
Polyhedral languages Like StreamIt, those languages focus on the generation of efficient code from high-level descriptions of dataflow-like programs. While languages such as Alpha [Le Verge et al., 1991], ArrayOL [Demeure et al., 1995] and CRP [Feautrier, 2006] do not manipulate streams, their compilation focuses on the problem of turning a program computing lazily over an unbounded (or very large) piece of data into a sequential or parallel piece of code that can be efficiently implemented on a finite computer or circuit. These languages are based on the polyhedral model [Feautrier, 1992a,b] which provides an expressive multidimensional setting for scheduling and code generation [Bastoul, 2004]. Like StreamIt, these languages focus on the search for good schedules rather than on the study of such schedules as objects exposed to the programmer and integrated into the language. Most of them also reject separate compilation, with the notable exception of CRP.

Other languages Other dataflow languages include the Ptolemy framework, developed by Lee and his students in Berkeley [Eker et al., 2003], and the Orcc [Yviquel et al., 2013] implementation of CAL [Eker and Janneck, 2003]. They are more expressive than both AcidS and the aforementioned scheduled dataflow languages. For instance, RVC-CAL is able to express non-determinism. The price to pay is that, in general, programs written in these languages have to be interpreted, making them unfit for critical systems or circuits. Static scheduling is viewed as a best-effort optimization that is not exposed to the programmer, in contrast with the clock types of synchronous languages.

6.1.4 Models for Streaming Systems

Schedulable dataflow StreamIt is partly founded on (C)SDF graphs. Such graphs belong to a family of formalisms used for the analysis of streaming systems. This area is generally considered to spark from the Synchronous Dataflow Graphs of Lee and Messerschmitt [1987], but its roots go back at least to the Computation Graphs of Karp and Miller [1966] and to the Even Graphs of Commoner et al. [1971], the latter being a subclass of Petri Nets. We will call such models *schedulable dataflow* for the lack of a standard name.

All these works represent a static Kahn network as a finite graph where nodes correspond to processes and edges to channels. The graph is supplemented with quantitative information including at least the initial occupancy of each channel and, for each actor, the amount of data consumed and produced by its firing. The exact format of this information depends on the formalism used; we recall some of them in Table 6.1, in order of increasing expressiveness.



(a) - An example SDF graph

$$\begin{cases} w_a \; on \; (2)^\omega & <:_1 & w_b \; on \; (3)^\omega \; on \; 0^6(1)^\omega \\ w_b \; on \; (3)^\omega & <:_1 & w_c \; on \; (4)^\omega \; on \; 0^9(1)^\omega \\ w_c \; on \; (4)^\omega & <:_1 & w_a \; on \; (2)^\omega \; on \; 0^5(1)^\omega \end{cases}$$

(b) - Inequations obeyed by its schedules

$w_a = (1)^\omega, w_b = 1^2(1^2\ 0)^\omega, w_c = 1^3(1^2\ 0^2)^\omega$

(c) - A valid solution; describes a schedule

Figure 6.1: Encoding SDF graphs as inequations on clocks

The literature has mostly focused on performance and scheduling analyses enabled by the availability of such static information.

An essential difference between schedulable dataflow models and the synchronous programming world is a difference of objectives. From the point of view of schedulable dataflow, the interest of clock types is the close integration of scheduling inside the programming language. This integration provided by the type-based approach has several benefits. First, it exposes schedules to the programmer via a precise language. Second, clock types are modular in the sense that they express the precise interface between a subprogram and its calling context. Third, one may use them to explain the generation of statically-scheduled code in a precise manner, as in Chapter 4. Furthermore, separate compilation is built-in, alleviating the need for ad hoc extensions such as in Tripakis et al. [2013].

The problem of scheduling a (C)SDF graph is equivalent to solving a system of inequations over ultimately periodic integer words. Figure 6.1 gives an example of this correspondence for SDF graphs. The left side of the figure describes the graph. Round nodes correspond to actors and square nodes to initialized buffers. The number in a square node gives the amount of initialization data and the numbers on edges model data consumed and produced by one firing of the corresponding actor. The right upper side of the figure gives the system of inequations on clocks. Each actor x gives rise to an unknown clock w_x which models its schedule. Each channel where a producer x writes p values, a consumer y reads c values and initialized with n initial values gives rise to an inequation $w_x \; on \; (p)^\omega <:_1 w_y \; on \; (c)^\omega \; on \; 0^n(1)^\omega$. The use of 1-adaptability is consistent with the usual semantics of schedulable dataflow graphs, where no instantaneous communication may occur. The lower right side gives a possible solution to the system. The fact that such a solution exists guarantees that the underlying SDF is deadlock-free, since all its words have non-zero rates. This encoding readily adapts to CSDF or Thresholded CSDF graphs.

This thesis proposes new operations that were, to our knowledge, never considered in schedulable dataflow models. Traditional scheduling techniques search for solutions within the set of binary words, where a process is only fired once per global time step. In fact, authors

with a background in Petri nets (e.g., Benabid-Najjar et al. [2012]) typically add additional *non-reentrance* constraints that forbid multiple firings of the same actor per time step, and thus reject integer clocks from the start. This comes from the fact that, in the SDF setting, it is generally assumed that actors fired at the same time step execute in a completely independent fashion. From this point of view, simultaneous firings of the same actor correspond to parallel execution, which is problematic if the actor is stateful. Concerning local time scales, rescaling by a constant strictly positive clock corresponds to *unfolding*, which has been used by several authors [Parhi and Messerschmitt, 1991; Chao and Sha, 1997]. More general cases have not been studied, as far as we know. This might come from the fact that they become needed when one wants to compose separate programs, which has traditionally not been a focus in the schedulable dataflow community.

An important criticism of our work is that its algorithmic side is nearly non-existent compared to the wealth of proposals available for schedulable dataflow graphs and related formalisms. Powerful optimization and analysis techniques for (C)SDF graphs abound, including for instance scheduling techniques that balance throughput and buffer sizes [Bodin et al., 2013]. Such a technique would correspond to a clock type inference algorithm in our setting; we will discuss this question in Section 6.2. We believe that most of the literature on schedulable dataflow could be reused and studied from the clock typing point of view, and therefore that the two approaches are complementary rather than in competition.

Modeling languages Beyond schedulable dataflow lies the question of documenting, modeling, and analyzing arbitrary systems that involve timing and causality issues. This problem is very general, and thus a large number of approaches have been proposed. Let us only mention the *Clock Constraint Specification Language* (CCSL) of André and Mallet [Mallet, 2008; André, 2009]. CCSL provides a rich language of operators and relations to describe timing and causality constraints which are partly inspired from the original work on n-synchrony [Cohen et al., 2006], including ultimately periodic binary clocks. It has recently been used to analyze and schedule stream-processing systems [Mallet et al., 2010; Yin et al., 2013]. Like schedulable dataflow models, such works are generally more concerned with modeling and analysis than with programming and code generation.

6.1.5 Functional Programming

Most of the languages we have discussed up to this point are first-order, with the exception of Lucid Synchrone and ReactiveML. We now turn to work proposed by the programming language community at large, and in particular to functional programming languages. Since the literature is huge, we focus on the work related to the design and implementation of reactive systems and digital circuits.

Functional Reactive Programming The ability to program with streams and other infinite data structures is intrinsic to higher-order lazy functional languages, which should therefore be relevant programming reactive systems. Indeed, in an influential paper Elliott and Hudak

[1997] propose Fran, a Haskell library dedicated to *Functional Reactive Programming* (FRP). The lazy nature of Haskell makes describing time-varying behaviors simple, and they are easily composed through dedicated combinators. As a library, Fran integrates well with the rest of the language. Unfortunately, the expressiveness of Haskell is both a blessing and a curse: a well-known problem of traditional FRP is that it makes it too easy to leak memory by retaining history from the past indefinitely. Various solutions to this problem have been proposed, some abandoning the first-class nature of time-varying values [Wan et al., 2002; Nilsson et al., 2002], while more recent ones use dedicated type systems inspired from modal logic to keep the amount of retained history in check [Jeffrey, 2012; Jeltsch, 2012; Krishnaswami, 2013].

A difference between traditional FRP and synchronous languages is that the latter seek to enforce that programs have finite state, and if it is the case to generate such an implementation. This motivates the use of type systems which by their very nature reject some good programs, favoring safety over expressiveness. With the more recent FRP systems that rely on dedicated type systems, this distinction is less clear cut. In particular, some aspects of the type system proposed by Krishnaswami [2013], such as its fixpoint rule inherited from modal logic, looks similar to ours. The proposed implementation technique is also very reminiscent of synchronous programming. Yet, the idea that a program may receive distinct types that drive the code generation process is absent from this line of work, which also does not accept stream functions whose untyped semantics is not length-preserving. On the other hand, the language of Krishnaswami handles general user-declared time-varying data types, including but not limited to streams.

Functional programming for circuits Predating FRP by more than a decade, another line of work concerns the use of functional programming languages to describe and reason about digital circuits. Gammie [2013] surveys this area.

An important class of languages dedicated to circuits are domain-specific ones embedded into a functional host language as libraries. The idea is that functional programs using such libraries are actually circuit generators which, when run, produce a complete netlist. This netlist can then be passed to the usual synthesis flow. The Lava language [Bjesse et al., 1998], already mentioned in Chapter 3, is the poster-child for this approach. Programming circuit generators in a high-level language is convenient, making it easy, for example, to program a whole family of circuits parametrized over its number of inputs as a single Haskell function. Similarly, recursive circuits are simply recursive Haskell functions. All these features are not available in the languages proposed in this thesis. Another difference is that Lava is able to describe layout constraints, which are integral to a design targeting traditional VLSI, but do not accept the complicated stream functions accepted in our system. Thus, we would argue that Lava sits at a a lower-level of abstraction than AcidS where the programmer directly programs a circuit. Sheeran [2005] recounts the history of languages in the style of Lava, including predecessors and successors.

The Ruby language of Jones and Sheeran [1991] is a higher-order relational language tailored to the description of regular circuits. As for Signal, Ruby programs denote relations rather than functions, which makes the language very expressive, but non-executable in

general. Ruby is particularly apt at *refinement*, transforming abstract, naive programs into more concrete ones until an executable and efficient form is reached. Sheeran [1988] investigates the use for refinement for implementing the classic circuit transformations known as retiming and *slowdown*. Slowdown consists in adding a number of registers everywhere to a circuit, which in effect makes it compute on several distinct interleaved streams. Thus, conceptually, this transformation turns a single circuit into several slower ones. Slowdown is different from rescaling and buffering; its integration in our setting should be possible, and exhibit interactions between clocks and linearity.

Lava and its descendants, as well as Ruby, describe circuits where timing behavior is fixed by the designer, while in AcidS this is decided by clock types. This makes our work closer to the so-called High-Level Synthesis work, which seeks to compile timing-independent descriptions to digital hardware. While most of the works in this area apply to imperative languages such as C or C++, some authors have considered functional languages closer to our work. A first example is the *Statically Allocated Functional Language* (SAFL) of Mycroft and Sharp [2000]. SAFL is a first-order call-by-value language in which recursion may only occur in tail position. There is no restriction on linearity but the compiler has to count its number of calls in the whole program, breaking separate compilation. Mycroft and Sharp investigate program transformations resulting in space-time trade-offs similar to the ones that can be achieved using local time scales and integer clocks. The fact that our transformations are characterized through a type system has the advantage of reducing the correctness of each transformation to the soundness of the type system. Another example which was a direct inspiration for our handling of higher-order functions is the Geometry of Synthesis of Ghica and his collaborators [Ghica, 2007; Ghica and Smith, 2010, 2011; Ghica et al., 2011]. We have already discussed it in detail at the end of Chapter 4.

Productivity in Type Theory The type systems proposed in this thesis rule out programs that do not define infinite streams. The same need arises in type theories with coinductive objects, where partial definitions need to be rejected to preserve logical consistency. Following Nakano [2000], various authors (e.g., [Appel et al., 2007; Krishnaswami and Benton, 2011; Birkedal et al., 2012; Atkey and McBride, 2013; Bizjak et al., 2016]) have built type-based productivity checkers into programming languages or proof assistants. The basic idea dates back to the Gödel-Löb logic of provability. Predicates are enriched with a "later" modality $\triangleright$ such that the formula $\triangleright P$ denotes the truth of P at the next time step. Productive recursive definitions are then allowed by the presence of an axiom $\mathbf{fix} : (\triangleright P \Rightarrow P) \Rightarrow P$, which is very similar to our FIX rule. This axiom is sometimes called the Löb or Gödel-Löb rule. We call *modal type theories* the type theories that are based on a "later"-like modality and the Löb rule.

In contrast with the clock type systems exposed in this thesis, modal type theories aim at generality and relative simplicity. For instance, they allow arbitrary coinductive types, such as infinite trees. In contrast, clock type systems only handle stream transformers. On the other hand, clock type systems describe fine-grained aspects of operational behavior that are not captured by current modal type theories. This is what enables the use of clock for generating finite-state code. Also, modal type theories have a more Church-style outlook than us: types

come first, and typing is generally not regarded as a refinement process that brings additional information about untyped programs.

6.1.6 Circuit Design

Latency-Insensitive Design In general, inserting a register in a synchronous digital circuit completely changes its final result. This makes it difficult to tweak the number and position of registers in a circuit to improve its performance, or to compose two circuits that have been designed separately. Carloni et al. [2001] proposed to work around this limitation by making circuits tolerate the insertion of arbitrary delays on datapaths. Such circuits are called *Latency-Insensitive Designs* (LIDs). Under mild hypotheses, a digital synchronous circuit can be enclosed within control logic that makes it latency-insensitive. LIDs can be composed using dynamic (e.g., Carloni et al. [2001]; Cortadella et al. [2006]; Cao et al. [2015]) or static (e.g., Boucaron et al. [2007]; Carmona et al. [2011]) scheduling. Compared to ordinary circuits, latency-insensitive ones can be optimized and analyzed in novel ways [Bufistov et al., 2008; Oms et al., 2010].

The goals and methods of this line of work are very close to n-synchrony. Indeed, we like to think that Latency-Insensitive Design is the study of efficient implementation and analysis techniques for Kahn Process Networks implemented in hardware. In fact, Lucy-n has been used to model statically-scheduled latency-insensitive designs [Mandel et al., 2011], and de Simone and his students have used ultimately periodic clocks [Boucaron et al., 2007; Millo and De Simone, 2012] to study LIDs. Like Lucy-n, well-typed μAS programs give rise to statically-scheduled circuits that are latency-insensitive by construction. Moreover, one can see this thesis as extending the idea of modular circuit design to circuits that are not only insensitive to variations in *time* but also to variations in *space*. From a more technical point of view, LIDs also ask the question of mixing dynamic and static scheduling. We will discuss this point in more details in Section 6.2.

High-Level Synthesis Let us finish with a quick word of traditional High-Level Synthesis. Coussy et al. [2009] provide an introduction. The generation of an efficient digital circuit from a piece of C code, for instance, is a difficult problem which requires powerful static analyses, language restrictions, and aggressive optimizers. High-Level Synthesizers typically work as black boxes whose inner working is difficult to understand by the programmer. The remarks we made for schedulable dataflow models apply: clock types and integer clocks offer a precise and clean language shared between the programmer and the compiler. They also express what it means for a schedule to be valid for a given program, and the corresponding code generation scheme, from the heuristics searching for such a schedule. A variant of AcidS might serve as a good intermediate language in a High-Level Synthesis tool flow, but this idea remains to be investigated.

6.2 Future Work

In this section we discuss future research directions, improvements, and issues we have not tackled yet. Some of these questions are practical and need to be addressed before a realistic implementation of AcidS is possible, while others are more theoretical in nature; some are halfway between theory and practice. We begin with the more practical side before moving on to theoretical matters.

6.2.1 Practical Aspects

The languages presented in Chapter 3 and the extensions in Chapter 5 are in a sense ready to be implemented. One can simply propose a Church-style syntax for the language akin to the one in Section 5.4, with explicit type abstractions and applications, local time scales, as well as Lucy-n-style buffers for introducing adaptability constraints. This makes type-checking simple. One may then compile the resulting fully typed digital circuits using the code generation scheme of Chapter 4.

Unfortunately, this approach does not result in a usable programming language. First, the amount of code needed to type a program is potentially very large and difficult to find. This mandates some amount of type inference. Second, the expressiveness of the language is questionable. Third, in practice one may sometimes want to generate software code rather than circuits, an issue we have not discussed yet. We now discuss each of these questions in turn, including complete or partial solutions to some of them.

Type Inference

Up to now we have not discussed at all the issue of type inference. In the setting of Chapter 3, a type inference procedure can be understood formally as a computable partial function taking a program e and a context Γ and returning a type t and a derivation of $\Gamma \vdash e : t$. This may involve finding the clock types of polymorphic operators such as constants, introducing adaptability constraints through the SUB rule, and inferring local time scales together with their driving clock types. While we do not have a complete solution to this problem, the design space is partly understood.

In the case of ultimately periodic binary clocks and in the absence of local time scales, type inference has been thoroughly investigated by Plateau and collaborators [Cohen et al., 2008; Mandel and Plateau, 2012; Plateau, 2010]. Since the type system reasons up to equivalence of clock types, type inference involves algebraic manipulations of ultimately periodic words. A raw source program gives rise to a set of adaptability and equality constraints on types, which is then reduced to a set of equations and inequations over ultimately periodic words. The real difficulty lies in the resolution of such a system; various trade-offs between precision and efficiency have been explored. Any solution to the constraints on ultimately periodic word immediately gives rise to a solution at the level of types. It is then simple to elaborate the program into an explicitly typed form.

$$
\begin{array}{lcll}
S & ::= & \exists B^*.C^* & \text{Constraint system} \\
B & ::= & c \leq \overline{n} & \text{Unknown (with } \overline{n} \in \mathbb{N} \cup \{\infty\}\text{)} \\
C & ::= & s <:_n s & \text{Adaptability constraint} \\
 & | & s = s & \text{Equality constraint} \\
s & ::= & c \,\mathbf{on}\, p & \text{Unknown composed with a constant} \\
 & | & p & \text{Constant word}
\end{array}
$$

(a) Syntax of constraints over ultimately periodic words

$$
\exists(c_x \leq \infty, c_y \leq \infty).\begin{cases} c_x \,\mathbf{on}\,(1\,0) & = & c_x \,\mathbf{on}\,(0\,1) \\ c_x \,\mathbf{on}\,(0\,1\,0) & <:_1 & c_y \,\mathbf{on}\,(1\,0\,0) \end{cases} \qquad \begin{cases} c_x & = & (2) \\ c_y & = & 0(1\,0\,5) \end{cases}
$$

(b) An example of system and one of its possible solutions

Figure 6.2: Systems of ultimately periodic words

This approach can be extended to infer type instantiations and subtyping (adaptability) in the polymorphic system of Section 5.3. As before a source program gives rise to a canonical set of constraints on words, which may now contain arbitrary natural numbers. The precise syntax of a system of constraints S is given in Figure 6.2 (a). A system is formed of a finite list of constraints C. A constraint is either a k-adaptability constraint, coming from implicit applications of the subtyping rule or fixpoints, or an equality constraint. Constraints involve either an unknown c composed with a constant word p on the right, or simply a constant p. As usual, constraints without unknowns can simply be removed once their validity has been verified. A simple example of constraint system is given in Figure 6.2 (b), together with one possible solution.

We have experimented with techniques for solving such systems, including a preliminary implementation. Our initial approach is to extend to the general integer case the *concrete resolution* algorithm of Mandel and Plateau [2012]. Briefly, this algorithm reduces the system of constraints to an Integer Linear Programming (ILP) problem whose variables are the positions of 1 in unknowns words. The inequalities in the integer linear program encode the adaptability, equality and boundedness constraints from the word system, but also enforce that the integer variables actually describe well-formed words. The generation of inequalities has to be slightly modified to allow general integer words; for instance, the original technique enforces that the positions of successive ones are strictly increasing. This is no longer necessary with integer clocks, where several ones may be placed at the same position, indicating that an integer larger than one occurs there. One may encode various optimization goals as the objective function of the resulting linear program, as expressed in Mandel and Plateau [2012, Section 5.3].

While this algorithm is practical for small programs, it leads to linear programs exponential in the size of the original system. Since Integer Linear Programming is NP-hard, this is prohibitively expensive for larger programs. The *abstract resolution* algorithm of Cohen et al. [2008] is much more efficient. It replaces each word with its *envelope*, which is a pair of lines bounding the cumulative sum of the clock from below and above. This leads to an abstracted

system of constraints whose unknowns and constants are both envelopes, and can be solved by a linear program of proportional size. A solution to the abstract system consists in an envelope per unknown, and any word which belongs to this envelope is a solution of the original concrete system. This approach should readily adapt with integer clocks too, as remarked by Plateau [2010, Section 11.1.1] in the conclusion of her PhD thesis, but we have not developed nor implemented her insights yet.

Another approach to the resolution of ultimately periodic word would consist in adopting techniques from the schedulable dataflow community. As a first step in this direction, we are currently studying the scheduling techniques proposed by Bodin et al. [2013] for cyclo-static dataflow graphs in an n-synchronous perspective. It seems that most of the development therein carries over to our setting, including their key notion of precedence constraints over activations. Many details remain to be ironed out however.

Finally, let us go back to the level of types and typing derivations. The reduction of a typing problem to a system of constraints following the syntax of Figure 6.2 (a) only handles subtyping and clock instantiations, which are not the only typing constructs that need to be handled. First, we need to decide when to introduce polymorphic clock quantification. A simple solution would be to adopt ML-style polymorphism, where types are restricted to rank one polymorphism and generalized only at let bindings. The inference of local time scales is a more complex question. It may be more reasonable not to infer them, at least in a first implementation. Finally, we expect the inference of bounded exponentials of Section 5.1 to be straightforward. Since the language does not contain recursive functions or usage polymorphism, we may simply count the number of occurrences of each identifier. Node declarations should pose no particular problem since they are not first-class objects and their typing is syntax-directed: type inference should reduce to type checking.

Expressiveness

In this thesis we have focused more on the formal study of our proposed extensions, showing in particular how they make clock typing more uniform and modular, rather than on programming concrete applications and programs. As such, it is probable that AcidS is not a very convenient *programming* language without further extensions, even taking into account the developments in Chapter 5. Even if the precise design of such extensions should go hand-in-hand with the development of realistic programs, we now discuss some broad ideas related to both types and terms.

A first remark is that the type system presented in Section 5.4 is relatively inflexible. While more expressive than both Lustre and the ultimately periodic part of Lucy-n taken separately, this system is basically unable to say anything about clock types containing expressions that are not words. The authors of Lucy-n have proposed to expose the envelopes of the previous subsection to the programmer, in addition to their role as technical devices in abstract resolution. This leads to a clock type language that is able to express non-strictly periodic rhythms yet restrictive enough for the compiler to do algebraic reasoning. We wish to extend this idea first by clarifying some semantics issues that appear when one adds envelopes in clock types, then by designing other quantitative abstractions as well as a way to combine them together.

Another important issue is the design of an actual set of useful constructions and operators on streams. We have deliberately left the set of operators *op* unspecified in earlier chapters, since they are orthogonal to the type soundness and compilation issues under study. In practice, a realistic language targeting circuits should have a rich library of arithmetic operators and types handling several integer widths. It is possible that to be really convenient a language featuring such expressive types should have dedicated subtyping rules injecting inhabitants of narrow integer types into wider ones.

A more interesting and specific issue is the question of array handling. Since the beginning of this thesis we have only considered arrays with suspicion: one of the original ideas behind integer clocks was to generate efficient array-processing software code without mentioning them in the source language. In particular, it was felt that the complexity of optimizing functional arrays into traditional imperative code (e.g., Gérard et al. [2012]) could be sidestepped using a clock-based scheme. It is true that, using the work presented in this thesis, a programmer may generate distinct array-processing code from the same program with the guarantee that they compute the same final result. However, the generated code will not necessarily be efficient, as we will discuss later in this section. More importantly, arrays have a striking feature compared to streams: they allow *random access* to their elements. This is probably more convenient in lots of code, compared to expressing complicated access patterns using stream sampling and merging.

To add arrays to AcidS in practice, one can introduce a type $dt[n]$ similar the one present in the language of machines to the grammars of data types dt, as well as array indexing. The indices can be bounded using the bounded integer type appearing in Section 5.4. Interestingly, arrays integrate well with integer clocks. Consider the following operators.

$$\begin{array}{rcl} vec_n & : & dt :: ct \,\mathbf{on}\, (n) \multimap dt[n] :: ct \\ str_n & : & dt[n] :: ct \multimap dt :: ct \,\mathbf{on}\, (n) \end{array}$$

The idea is that vec_n transforms a stream of, say, integers into a "shorter" stream of arrays of integers, and conversely for str_n. Integer clocks makes it easy to express the action of such operators on clock types, and the data representation of Chapter 4 makes their compilation almost trivial: they simply correspond to the gathering and scattering machines.

The question of the amount of inference required from the compiler is not yet decided, as shown in the previous paragraph. In any case, it is probably useful to have explicit buffer operators corresponding to a Church-style SUB rule, as in Lucy-n. Initialized buffers are also an interesting possibility, since they are very easy to add to the type system and probably leads to the generation of better code. We will soon come back to this later point.

A difficult point is the reinitialization operator introduced in Lucid Synchrone and later adopted in modern Lustre dialects such as Heptagon or SCADE6. Earlier we have mentioned that this operator is at the heart of important high-level constructs such as hierarchical state-machines. Unfortunately, in our opinion its semantics is unclear, which makes understanding its interaction with n-synchronous clock types difficult. For instance, assuming that f is an expression of type $(1\,0) \multimap (0\,1)$, how can we understand f `every` $(1\,0)$, which intuitively expresses that we reinitialize f at every even time steps? Should its type be $(1\,1\,0) \multimap (0\,0\,1)$, or possibly $(1) \multimap (0)$? Can we reinitialize things that are not functions? It is possible that a type

system such as the one proposed by Hamon and Pouzet [2000] for controlling reinitializations could be added to AcidS; but they do not give an untyped semantics for the operator, which makes it hard to really understand.

Code Generation

Generating correct software code While Chapter 4 gives a blueprint for writing a compiler from AcidS to a hardware description language, it does not completely explain compilation to software language such as C. The issue lies in the feedback machine used to compile higher-order functions. While this machine has a direct implementation as a circuit, its implementation as software is difficult. In a lazy language such as Haskell, it can be simulated using the usual recursive definition of a fixpoint from the host language. Moreover, this is a modular implementation in the sense that the combinator implementing the feedback machine $\texttt{mfb}^{mt_3}_{mt_1,mt_2}(m)$ is independent from the implementation of the wrapped machine m. This does not work in a call-by-value language such as C since their semantics preclude recursively-defined values.

One possible solution is to translate the final circuit into sequential C code using techniques like those traditionally found in Esterel compilers [Potop-Butucaru et al., 2007]. This breaks separate compilation. A more modular approach would be to avoid the Int() construction altogether and apply the usual closure-conversion transformation. Fortunately, the number of closures needed for each function is statically bounded because of the linear type system, and thus garbage collection is not needed. One may even allocate closures at compilation time, avoiding dynamic memory allocation.

Bypassing the Int() construction step reduces drastically the number of feedback machine in the output code. The only remaining feedback loops come from the compilation of the FIX rule and thus of source-level recursive definitions. Fortunately, the additional premises of the FIX rule makes their compilation easy. We will now explain it with an example. Assume that we have a library of ring buffers implementing at least the interface given in Figure 6.3 (a). The source code to be compiled is given in Figure 6.3 (b); the function f is abstract, and g is the prototypical fixpoint computation. Figure 6.3 (b) gives a simplified and idealized version of the C code generated for this fixpoint computation.

In this example each piece of code is compiled to three elements: a structure representing its internal state, a function allocating and initializing this internal state, and a transition function. The first part of the code gives the declarations of the corresponding objects for f. The second part gives the corresponding definitions for g. The internal state of g, `g_mem`, is composed of the state for the call to f, the buffer involved in the adaptability premise of the FIX rule, and an integer field holding the current value of the clock $0(1)^\omega$. The initialization function creates an instance of this record with an initially empty buffer and the field `w1p0` set to 1, the first value of $0(1)^\omega$. In accordance with the FIX rule, at every time step the buffer contains the latest output of f; the first call to `g_step` reads nothing from it, but we know from the type of f that at this point `f_step` does not depend on its input anyway.

```
struct buffer;
struct buffer *buffer_alloc(size_t n);
void buffer_write(struct buffer *, const int *src, size_t n);
void buffer_read(struct buffer *, int *dst, size_t n);
```

(a) - Buffer abstract data type

$$\begin{array}{lcl} f & : & \mathbf{int} :: 0(1) \multimap \mathbf{int} :: (1) \\ f & = & \ldots \end{array} \qquad \begin{array}{lcl} g & : & \mathbf{int} :: (1) \\ g & = & \mathtt{fix}\, f \end{array}$$

(b) - Example of fixpoint computation

```
// Declarations for f : int :: 0(1) -o int :: (1)

struct f_mem;
struct f_mem *f_alloc();
void f_step(struct f_mem *mem, int inp, int *out);

// Definitions for g : int :: (1)

struct g_mem             // Internal state of g
{
    struct f_mem *f_mem;    // Memory for the application of f
    struct buffer *fb_buff; // Buffer from the fixpoint rule
    int w0p1;               // Current value of 0(1)
};

struct g_mem *g_alloc() // State allocation function of g
{
    struct g_mem *mem = malloc(sizeof *mem);
    mem->f_mem = f_alloc();                  // Create the state of f
    mem->fb_buff = buffer_alloc(1);          // Create a 1-place buffer
    mem->w0p1 = 0;                           // Set first value of 0(1)
}

void g_step(struct g_mem *mem, int *out)  // Transition function of g
{
    int inp_ck, inp, out;
    inp_ck = mem->w0p1; mem->w0p1 = 1;       // Compute current value of 0(1)
    buffer_read(mem->fb_buff, &inp, inp_ck); // Read previous output
    f_step(mem->f_mem, inp, &out);           // Compute current output
    buffer_write(mem->fb_buff, &out, 1);     // Store current output
    return out;                              // Return current output
}
```

(c) - Generated C code for example (b)

Figure 6.3: Generating C code - buffers and fixpoints

```
node f() returns (x : int :: base;
                  y : int :: base)
let
  y = x + 1;
  x = 0 fby y;
tel
```

(a) - Heptagon/Lustre source code

```
struct f_mem {
    int x;
};

struct f_mem *f_alloc() {
    struct f_mem *mem = malloc(sizeof *mem);
    mem->x = 0;
    return mem;
}

void f_step(struct f_mem *mem,
            int *ox, int *oy)
{
    *ox = mem->x;
    *oy = *ox + 1;
    mem->x = *oy;
}
```

(b) - Generated C code

Figure 6.4: Generating C code - intra-step scheduling in Heptagon

$$
\begin{array}{lcl}
f & : & (1)\otimes(1) \\
f & = & \uparrow_{(2)} \mathtt{fix}\,(\mathtt{fun}\,(\mathtt{x}:(0\,1),\mathtt{y}:0(0\,1)).(\mathtt{merge}\,1(0)\,0\,\mathtt{y},1+\mathtt{x}):(1\,0)\otimes(0\,1))
\end{array}
$$

Figure 6.5: Generating C code - intra-step scheduling reflected in clock types

Generating efficient software code Generating good code will require specific optimizations in addition to those already discussed in Section 4.5. More specifically, our use of local time scales to avoid the traditional intra-step scheduling raises serious efficiency questions. Consider the Lustre node in Figure 6.4 (a). Its two outputs denote the streams of natural numbers and of strictly positive natural numbers, respectively. The C code generated by the Heptagon [Delaval et al., 2012] compiler looks basically like the one given in Figure 6.4 (b), up to uninteresting technical details. It produces one element per time step for each stream. Figure 6.5 gives a version of this program in μAS, using a small amount of syntactic sugar, type annotations, and an explicit local time scale.

Looking carefully at the clock types of x and y, we see that they reflect the intra-step scheduling performed by the Heptagon compiler. The goal is now to exploit this information to obtain code similar to the one in Figure 6.4 (b). To show that this is not completely trivial, Figure 6.6 features the code that could be generated by a naive AcidS compiler. Its state consists in two counters implementing ultimately periodic words, and two integers implementing the one-place buffers that correspond to the premise $\vdash (1\,0)\otimes(0\,1) <:_1 (0\,1)\otimes 0(0\,1)$ in the fixpoint rule. The body of the transition function consists in a counted loop implementing the local time scale driven by $(2)^\omega$. Its body can be decomposed into four main parts. The first three parts implement the fixpoint compilation scheme as in Figure 6.3. Lines 21-30 read the previous outputs of the fixpoint, computing the required clock $(0\,1)^\omega$ and $0(0\,1)^\omega$. Remark how we have assume that the compiler is smart enough to compute $(0\,1)^\omega$ using the loop index; this is not immediate either. Lines 31-41 implement the body of the fixpoint. Remember

```c
struct f_mem {
    int w1p0;  // 1(0)
    int w0p01; // 0(0 1)
    int b1;
    int b2;
};

struct f_mem *f_alloc()
{
    struct f_mem *mem = malloc(sizeof *mem);
    mem->w1p0 = 1;
    mem->w0p01 = 0;
    return mem;
}


void f_step(struct f_mem *mem, int *ox, int *oy)
{
    int x, y, nx, ny, y_ck;
    for (int i = 0; i < 2; ++i) {
        // Fixpoint: read previous output x on (0 1)
        if (i == 1) x = mem->b1;
        // Compute current value of 0(0 1)
        y_ck = (mem->w0p01 > 1) && (mem->w0p01 % 2 == 0);
        if (mem->w0p01 < 2) mem->w0p01++;
        // Fixpoint: read previous output y on 0(0 1)
        if (y_ck == 1) y = mem->b2;
        // Merge: driven by (1 0)
        if (i == 0) {
            if (mem->w1p0 == 1) {
                mem->w1p0 = 0;
                nx = 0;
            } else
                nx = y;
        }
        // Operator: driven by (0 1)
        if (i == 1) ny = x + 1;
        // Fixpoint: store current output x on (1 0)
        if (i == 0) mem->b1 = nx;
        // Fixpoint: store current output y on (0 1)
        if (i == 1) mem->b2 = ny;
        // Local time scale: gather x from (1 0) to (1)
        if (i == 0) *ox = nx;
        // Local time scale: gather y from (0 1) to (1)
        if (i == 1) *oy = ny;
    }
}
```

Figure 6.6: Generating C code - intra-step scheduling in AcidS - semi-naive

```
1  struct f_mem {
2      int w1p0;  // 1(0)
3      int b2;
4  };
5
6  struct f_mem *f_alloc()
7  {
8      struct f_mem *mem = malloc(sizeof *mem);
9      mem->w1p0 = 1;
10     return mem;
11 }
12
13
14 void f_step(struct f_mem *mem, int *ox, int *oy)
15 {
16     int x, y, nx, ny, y_ck, b1;
17
18     // First iteration
19     if (mem->w1p0 == 0) y = mem->b2;
20     if (mem->w1p0 == 1) {
21         mem->w1p0 = 0;
22         nx = 0;
23     } else
24         nx = y;
25     *ox = nx;
26     b1 = nx;
27
28     // Second iteration
29     x = b1;
30     *oy = x + 1;
31     mem->b2 = y;
32     *oy = y;
33 }
```

Figure 6.7: Generating C code - intra-step scheduling in AcidS - optimized

from Chapter 4 that compiling stream merging and pointwise operators generate new local time scales because of their implicit clock-polymorphism. This is what the tests on lines 32 and 40 correspond to. Lines 42-47 store the current outputs of the fixpoint into the buffer. Finally, lines 48-53 performs gathering, which is here particularly simple: one simply has to store each current output of the fixpoint into the corresponding output of `f` at the proper time.

The code from Figure 6.6 is unsatisfactory. It is true that optimizing C compilers may recover some amount of performance using traditional optimizations. For instance, in our precise example it is clearly helpful to unroll the loop and perform constant folding and other dataflow optimizations. However, this is only limited. First, on embedded platforms we may not have a good enough optimizing C compiler at our disposal. Second, some optimizations cannot be performed by the C compilers, barring whole-program compilation. One such optimizations appears in the C code of Figure 6.6: the clock $0(0\ 1)^\omega$ is actually redundant as it can here be expressed in terms of $0(1)^\omega$. To see why, consider the effect of unrolling the local time scale driven by (2). In the first iteration, we only see the elements of $0(0\ 1)^\omega$ of even rank, that is $0(0\ 1)^\omega$ *when* $(1\ 0)^\omega = 0(1)^\omega$. This clock is the element-wise negation of $1(0)^\omega$, which already appears as the argument of the stream merging operator. For the second iteration, we

have $0(0\ 1)^\omega$ *when* $(0\ 1)^\omega = (0)^\omega$. This equation implies that some conditions are always false in the second iteration, and the corresponding if statements can thus be removed.

Figure 6.7 gives a new version of the C code implementing Figure 6.5. We have unrolled the loop, removed statements which were only executed in one iteration from the other, optimized some variables away, and performed the clock simplification explained above. We have left some simple redundancies and useless variables to make the link with the previous code clearer; they would be removed by any copy propagation pass. One interesting point is that the buffer `b1` has disappeared. It was in fact always empty at the end of a call to `f_step`, as its only role is to transmit data from the first iteration to the second one. One may implement such behavior using a simple local variable. In contrast, the value stored into `b2` survives from one global time step to the next, and therefore this variable has to be part of the state record, as in the code generated by Heptagon (Figure 6.4 (b)).

Remark 34. We believe that his example sheds light on the notion of local variable in synchronous languages. The compilation of synchronous languages involves both transient storage, corresponding to wires (in circuits) or local variables (in software), and persistent storage, that is registers (in circuits) or state variables (in software). In existing languages, an expression is implemented through a register if and only if it is a delay in Lustre/Lucid Synchrone, or buffer in Lucy-n. In contrast, in AcidS a buffer can sometimes be implemented as a local variable, provided it is observed *at a sufficiently high level of granularity*. Local time scales makes it possible to control this granularity, hiding internal steps and thus transforming persistent storage into transient storage, as in our example. Thus the fact that a variable is local or not depends on the clock at which it can be observed rather than on a syntactic criterion.

One may argue that the code in Figure 6.7 is not yet as satisfying as the one in Figure 6.4 (b), even after copy propagation has been performed. Indeed, it features some unnecessary state and control related to the fact that, conceptually, the code generated by Heptagon has precomputed some results used in the first time step. The solution is probably to add an initialized buffering operator generalizing `fby` to the n-synchronous setting. Using such an operator it should be possible to generate code comparable to the one produced by existing compilers in the case of Lustre-like programs.

This discussion and illustration of the optimizations needed for sequential code generation shows that much work remains to be done. While we do not have a good grip on which optimizations are important and which are redundant with the ones provided by optimizing C compilers, we feel that the unrolling of local time scales is promising and should be investigated further. Its interaction with clock types should allow for further optimizations.

Modular compilation All along this thesis we have defended the dogma of modularity, understood in the sense of separate compilation. But separate compilation actually consists in two different properties, separate type checking and separate code generation. The first property is actually a general property of type systems: the type of a program is only determined by the type of its constituent, not by their bodies. In practice, this property is not true of languages with macro systems such as C++ templates or the static recursion of Lustre, where it is impossible to type-check programs without a first phase of global program rewriting. The

second property is more restrictive, as it means that changing the body of a subprogram does not cause the recompilation of the surrounding context as long as its type does not change.

Separate code generation is actually an expensive property when targeting circuits. This comes from the fact that circuits not only have finite state, but state whose size is statically bounded *syntactically*. Indeed, we have excluded several very useful type constructors from the type systems of Chapter 5 because of our insistence on separate code generation. These type formers include unbounded clock polymorphism and unbounded exponentials, but also other forms of polymorphism such as the usual "data" parametric polymorphism. One cannot generate a unique piece of circuit for a program with such a type, since they can be called from arbitrary many contexts needing different physical data representations.

In practice the value of such features may outweigh the benefits of completely separate code generation. The general solution is to introduce a dedicated linking process that would inline and specialize nodes with such types, making them disappear before code generation. Note that in contrast with the removal of higher-order features from a functional language, such transformations are type-directed local reductions that do not require a form of global rewriting. The handling of nodes of monomorphic or bounded type would still be modular.

Parallelism A last and more speculative subject is the generation of parallel software code from AcidS programs. At first sight it may seem trivial: any program, even an ill-typed one, has a natural interpretation as a Kahn network. In addition, the type system provides sufficient buffer sizes for well-typed programs. There is thus no technical difficulty running AcidS programs on top of any runtime implementing the Kahn model. The difficulty is rather in the generation of *good* parallel code. This question differs from the traditional issues of distribution (e.g., Girault [2005]) or real-time scheduling (e.g., Forget et al. [2008]) of Lustre programs. We do not wish to add explicit parallel programming constructs [Cohen et al., 2012] either. Lustre-like languages encourage very fine-grained code with lots of tightly-coupled dependencies. This is not good for parallelism, as even efficient runtime systems struggle when buried under huge numbers of very short tasks. Moreover, on modern platforms it is actually beneficial to communicate large chunks of data sparingly rather than frequently but by small amounts [Lê et al., 2013].

A potentially important feature of integer clocks is that they are actually capable of expressing trade-offs between computation and synchronization that matter in parallel programs. In fact, this was one of the original motivations for their introduction. As an example, a programmer can easily introduce buffering to turn a stream of clock $(1)^\omega$ into one of clock $(0\,0\,3)^\omega$ which may be processed faster in practice. Similarly, creating a local time scale driven by a fast clock makes it possible to increase the computation/synchronization ratio of a piece of code, while being sure that its functional semantics is preserved.

The addition of actual parallel programming features to AcidS raises a lot of questions, from a practical but also theoretical point of view. Can we design a language where parts of programs are dynamically-scheduled and others are sequential and statically-scheduled? How should the boundaries between such parts be delimited? Should the compiler, informed with clocking information, try to set this frontier, seen as an automatic parallelization problem?

$$\boxed{\Delta;\Gamma \vdash t \geq t'}$$

$$\frac{}{\Delta;\Gamma \vdash t \geq t} \qquad \frac{\Delta;\Gamma \vdash t_1 \geq t_2 \qquad \Delta;\Gamma \vdash t_2 \geq t_3}{\Delta;\Gamma \vdash t_1 \geq t_3}$$

$$\frac{\Delta;\Gamma \vdash t_1 \geq t_1' \qquad \Delta;\Gamma \vdash t_2 \geq t_2'}{\Delta;\Gamma \vdash t_1 \otimes t_2 \geq t_1' \otimes t_2'} \qquad \frac{\Delta;\Gamma \vdash t_1' \geq t_1 \qquad \Delta;\Gamma \vdash t_2 \geq t_2'}{\Delta;\Gamma \vdash t_1 \multimap t_2 \geq t_1' \multimap t_2'} \qquad \frac{\Delta,\alpha \leq n;\Gamma \vdash t \geq t'}{\Delta;\Gamma \vdash \forall(\alpha \leq n).t \geq \forall(\alpha \leq n).t'}$$

$$\frac{\Delta;\Gamma \vdash t <:_0 t'}{\Delta;\Gamma \vdash t \geq t'} \qquad \frac{\Delta \vdash ct \leq n \qquad \Delta \vdash \Gamma \downarrow_{ct} \Gamma' \qquad \Delta \vdash t \uparrow_{ct} t'}{\Delta;\Gamma \vdash t \geq t'} \qquad \frac{\Delta \vdash ct \leq n}{\Delta;\Gamma \vdash \forall(\alpha \leq n).t \geq t[\alpha/ct]}$$

Figure 6.8: Principality - type containment for polymorphic clock types

What should these boundaries look like as syntactic as well as semantic features? Could such a multi-mode language lead to a more elegant source-to-source formulation of compilation? These issues remain to be investigated, and a practical parallel code generator will have to wait for the availability of a good sequential code generator.

6.2.2 Theoretical Aspects

We finish this section with a discussion of the theoretical properties of the type system from the point of view of modularity, as well as improved formulations of the semantics of AcidS.

Clocks and Types

The design of a practical type system as well as type inference engine for AcidS raises the issue of the relationship between an untyped program and its set of possible types and typing derivations. This raises two different issues, one being syntactic and the other syntactic.

Principality A well-known property a type system may enjoy is the existence of *principal types*. A principal type is the most general type for a given program in a fixed typing environment in the sense that any other type is an *instance* of the principal one. The precise definition of the instance relation between types depends on the system under consideration; for more details and nuance we refer the reader to Wells [2002].[4] In the best case the principal type of a program can actually be computed, a famous example being the Damas-Milner type system used in ML. This tends to make type inference more robust and predictable.

To discuss the issue of principality in our systems, we first need to fix a reasonable instance relationship between clock types. We focus on the polymorphic type system of Section 5.3

[4]The paper of Wells is actually about the more general notion of *principal typings*, but principal types are also discussed at length.

since it strikes a good balance between simplicity and expressiveness. The *type containment* relation characterizes the clock transformations that can be applied to a typed program. It is expressed as a judgment holding in a fixed pair of typing contexts Δ and Γ; we write $\Delta;\Gamma \vdash t \geq t'$. A derivation of $\Delta;\Gamma \vdash t \geq t'$ encodes a way of turning any derivation of $\Delta;\Gamma \vdash e : t'$ into one of $\Delta;\Gamma \vdash e : t$. Thus, a type inference engine focused on modularity should maximize the inferred types, and principality can be understood as the existence a largest type for any fixed well-typed program.

The rules of the type containment judgment are given in Figure 6.8. Most of them are administrative, expressing that it is a preorder and a congruence with regard to type constructors. The interesting rules express that type containment includes adaptability, the creation of local time scales, and the instantiation of polymorphic types.

Remark 35. This definition of type containment describes type transformations related to clocks. It is also possible to define a containment relation characterizing transformations related to the system with bounded exponentials of Section 5.1. We have not studied the corresponding notion of principality.

The type systems proposed in this thesis do not have principal types. Let us begin with the monomorphic clock type system of μAS (Chapter 3). The type containment relation of Figure 6.8 adapts to this simpler system by removing clock contexts and rules related to quantifiers. The fact that this system does not have principal types can be seen by examining the program `fun (x,y).(x,y)`, introduced in the thesis of Gonthier [1988], which we will henceforth call *gon*. The problem with *gon* here is that while its two outputs are completely independent, any monomorphic type will add unwanted dependencies between them. For example, the type $(1)\otimes(1) \multimap (1)\otimes(1)$ makes the second output depend on the first output, and thus cannot be turned into the type $0(1)\otimes(1) \multimap 0(1)\otimes(1)$ which is also valid for *gon*. Bounded polymorphic clock types solve this problem by assigning to *gon* a type such as

$$\forall(\alpha_1 \leq n_1).\forall(\alpha_2 \leq n_2).\alpha_1 \otimes \alpha_2 \multimap \alpha_1 \otimes \alpha_2$$

with n_1 and n_2 fixed constants, which clearly reflects the independence of `x` and `y`. Unfortunately, the presence of bounds on polymorphic variables break principality again. In the type above the choice of n_1 and n_2 are arbitrary, and will thus forbid valid instantiations of the corresponding type variables. This defect also affects the dependent clock types of Section 5.4.

Unbounded polymorphism, as proposed in the conclusion of Section 5.3, is able to assign the type $\forall(\alpha_1 \leq \infty).\forall(\alpha_2 \leq \infty).\alpha_1 \otimes \alpha_2 \multimap \alpha_1 \otimes \alpha_2$ to the function *gon*. This type would actually be principal for such a system. One may wonder whether a system with unbounded polymorphism has principal types in general. Unfortunately this is not the case. We explain why using an example proposed by Raymond [1988] in his master thesis. Call *ray* the program `fun (x,y).(and(x,y),y)`, `and` being the usual boolean conjunction lifted elementwise. Its first output depends on both of its inputs while the second output only depends on the first input. Let us call ct_x, ct_y, ct_p, and ct_q the clock types of `x`, `y` and of the first and second outputs respectively. In a language such as μAS or Lucy-n, ct_x and ct_y must both be adaptable to ct_p, and thus synchronizable. It is also clear that ct_y must be synchronizable with ct_q. Since synchronizability is an equivalence relation, this implies that all four clock types must be

synchronizable. Since we want to fix buffer sizes locally, we must fix those synchronizable clock types at this point. There are lots of choices, including for instance $\forall(\alpha \leq \infty).\alpha \otimes \alpha \multimap \alpha \otimes \alpha$ and variations. Unfortunately, none of them are principal. In any case, the synchronizability constraints force all the clock types to have the same root, and thus express that the second output may depend on the first input, which is untrue. This is problematic, since there exists fixpoint computations that are rejected by one of these choices but accepted for the other.

This example can be understood as an intrinsic limitation of (n-)synchrony. Finite buffers introduce *back-pressure* dependencies: production depends on the availability of free space in the buffer, and thus on consumption. The shape and amount of back-pressure dependencies is determined by the size of the buffer rather than by the semantics of the untyped program. In the *ray* program, fixing buffer sizes introduce dependencies of the second output on the first input. In general, such dependencies need not exist, and thus no principal type exists.

This example illustrates a defect of our work: our type system mixes *finiteness* and *causality*, which are unrelated issues. We may abandon finiteness and thus synchronizability constraints, which leaves us only with the precedence relation. Clocks ordered by precedence form a lattice, in contrast with clocks ordered by adaptability. Thus, we could imagine having a least upper bound operator $\sqcup$ in clock types. In such a system, the function *ray* could receive the type $\forall(\alpha_1 \leq \infty).\forall(\alpha_2 \leq \infty).\alpha_1 \otimes \alpha_2 \multimap (\alpha_1 \sqcup \alpha_2) \otimes \alpha_2$ which captures exactly the dependencies between inputs and outputs. The interest of such a type is debatable: one may argue that it reveals too much information about the inner workings of the program. In any case, this type should be principal for *ray* in a system which does not deal with the finiteness of buffers.

To conclude this discussion, we believe that this example illustrates how in a language where causality and scheduling are captured inside types, the existence of principal types is intimately related to modular scheduling [Raymond, 1988; Pouzet and Raymond, 2010]. We feel that the nonexistence of principal types caused by finiteness constraints is a limitation of synchrony, and prevents truly modular scheduling in our setting. It should be noted, however, that our approach schedules more programs than traditional techniques [Pouzet and Raymond, 2010] can handle. Finally, and from a more philosophical point of view, this illustrates once again that finiteness is often at odds with true modularity.

Completeness Principality is a syntactic property of a type system. In our case, types are mostly approximations of clocks, which are semantic, non-computable objects. Clocks describe the growth of streams and make it possible to capture the semantics of stream functions, as explained in Chapter 2. One may wonder whether an analogue of the *best clock type* exists in the semantic setting: can we define a notion of *best clock* characterizing a stream function, and study its (in)existence? Gérard [2013] has studied this question in the third part of his thesis. We believe that his work could be revisited with integer clocks and local time scales in mind.

Semantics

Category theory In Chapter 3 and Chapter 4 we have given two separate interpretations of typing derivations. The first one was the synchronous denotational semantics defined in terms of domains. The second one was the compilation to higher-order macro-machines. We

strongly suspect that these two translations are actually the same interpretation in disguise and should thus be unified. Indeed, each translation can for the most part be explained as the interpretation of a program in a symmetric monoidal closed category (SMCC) where, for instance, value types correspond to comonoid objects. In the case of the synchronous semantics, this construction is somewhat obscured by the fact that the monoidal category of domains is actually cartesian. The categorical viewpoint would also justify retrospectively the organization of Chapter 4, decomposing the compilation process factors as the construction of a closed category of macro-machines from the non-closed monoidal category of machines followed by the generic interpretation of μAS into an SMCC with adequate structure.

The difficulty in giving a convincing categorical semantics for our language is one of axiomatization. We do not yet know what should be the structure required of a SMCC to interpret clock-related constructions, and in particular local time scales. An interesting lead is to remark that rescaling is fundamentally a substitution operation of a special kind occurring in types. This suggests interpreting it using fibrational techniques, as for ordinary parametric polymorphism, with the RESCALE rule corresponding to a *change of base* operation.

Type theory Another kind of improvement for the mathematical development present in this thesis would be to express them in a computer proof assistant based on dependent types. Most chapters of this thesis are already written in an informal type-theoretical framework, and we believe that moving from informal to formal would not be a very large step. Concretely, we contemplate the formalization of μAS and its derivatives in the style of Chapman [2009]. This would enable longer-term projects such as the development of a certified compiler.

Beyond streams

A last, more speculative research direction consists in seeking connections with other fields of programming language theory. We are particularly interested in investigating the relationship between synchronous compilation and the strictness analysis and deforestation optimization of lazy functional programs. The link between the two appears to be folklore in some parts of the synchronous and functional programming communities. For instance, one may read the following paragraph in Caspi and Pouzet [1996].

> Lazy evaluating this program is costly: intermediate lists are allocated and deallocated by the Garbage Collector during execution. On the contrary, the synchronous dataflow compilers translate it into a sequential program with bounded memory and response time. One could say that these compilers transform the call-by-need evaluation into a call-by-value one.

The connection between synchrony and the call-by-value evaluation strategy was also highlighted by Krishnaswami [2013] in more recent work. But, as far as we know, there is no systematic investigation of this link, which remains unclear at the moment. Another possibility is to understand synchronous compilation as a very specific kind of *incrementalization* of a stream function. Indeed, the resulting state machine is able to process streams chunk-by-chunk, while in the untyped semantics the whole output has to be recomputed each time

a new input is received. In any case, we believe that explaining synchronous compilation within a larger semantic framework may help with its extension to more expressive languages, including the ability to handle richer data types than streams.

6.3 Conclusion

In this thesis we have proposed a higher-order functional programming language called AcidS in which programs manipulate infinite streams of data. Each program has a natural interpretation as a Kahn process network, a well-known model of deterministic parallel computation. The language is endowed with a type system, called clock typing, which rejects programs that deadlock or cannot be executed within a finite amount of memory. This system works by establishing a notion of discrete time step through the whole program, following the tradition of synchronous languages. In a well-typed program each stream is paired with a clock, which is a mathematical object assigning to every element a time step at which it has to be computed. Well-typed programs are compiled to ordinary finite-state digital circuits. The role of typing information is not only to reject ill-behaved programs but also to drive code generation, influencing the shape of the final circuit in a precise way.

Several elements set our work apart from previous synchronous languages. First, in AcidS several elements of a stream may be computed during the same time step, which manifests by the fact that clocks are streams of integers rather than streams of booleans. The existing theory of clocks extends smoothly to this new setting. Second, the language offers a local time scale construct which hides from the outside some of the steps performed by a subprogram. Third, we enforce a linear discipline on higher-order functions, which must be used a statically bounded amount of times.

The combination of such features has a profound effect on the theory and practice of the type system. The linear nature of higher-order functions makes their modular compilation to circuits possible. Local time scales offer a new kind of modularity in synchronous functional languages; their ability to abstract precise timing information is instrumental in the integration of causality (deadlock-freedom) into types. Integer clocks give a high-level description of space/time trade-offs in the implementation of the same source program. In addition, having a unified type system that captures all the properties needed for the compilation to circuits makes the full formal description of the language tractable. This thesis provided a complete description of a synchronous functional language, its type system, and type-directed compilation function, including soundness proofs.

Appendix A

Index

Semantics

Judgments

Interpretations

Appendix B

Figures

Bibliography

Samson Abramsky and Achim Jung. Domain theory. In *Handbook of Logic in Computer Science*, pages 1–168. Clarendon Press, 1994. Cited on pages 20 and 25.

Rajeev Alur and Thomas A Henzinger. Reactive modules. *Formal Methods in System Design*, 15 (1):7–48, 1999. Cited on page 166.

R.M. Amadio and P.L. Curien. *Domains and Lambda-Calculi*. Cambridge University Press, 1998. Cited on pages 25, 83, and 199.

Charles André. Syntax and semantics of the clock constraint specification language (CCSL). Technical report, INRIA, 2009. Cited on page 232.

Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 2006. Cited on page 57.

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A Very Modal Model of a Modern, Major, General Type System. In *Symposium on Principles of Programming Languages (POPL'07)*. ACM, 2007. Cited on page 234.

Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977. Cited on pages 223 and 229.

Robert Atkey and Conor McBride. Productive Coprogramming with Guarded Recursion. In *International Conference on Functional Programming (ICFP 2013)*. ACM, 2013. Cited on page 234.

Steve Awodey. *Category Theory*. Oxford University Press, 2006. Cited on page 25.

Marc Bagnol and Adrien Guatto. Synchronous Machines: a Traced Category. Research report, INRIA, November 2012. URL `https://hal.inria.fr/hal-00748010`. Cited on page 165.

Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*. IEEE, 2004. Cited on page 230.

Abir Benabid-Najjar, Claire Hanen, Olivier Marchetti, and Alix Munier-Kordon. Periodic schedules for Unitary Timed Weighted Event Graphs. *Automatic Control, IEEE Transactions on*, 57 (5), 2012. Cited on page 232.

Nick Benton, Chung-Kil Hur, Andrew J. Kennedy, and Conor McBride. Strongly Typed Term Representations in Coq. *Journal of Automated Reasoning*, 49(2):141–159, 2012. Cited on page 110.

Albert Benveniste, Paul Le Guernic, Yves Sorel, and Michel Sorine. A denotational theory of synchronous reactive systems. *Inf. Comput.*, 99(2):192–230, 1992. Cited on pages 47 and 228.

Albert Benveniste, Benoıt Caillaud, and Paul Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, 163(1):125–171, 2000. Cited on page 227.

Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992. Cited on page 226.

Gérard Berry and Ellen Sentovich. Multiclock Esterel. In *IFIP Workshop on Correct Hardware Design and Verification Methods (CHARME'01)*, 2001. Cited on page 227.

Gérard Berry, Michael Kishinevsky, and Satnam Singh. System Level Design and Verification Using a Synchronous Language. In *International Conference on Computer-Aided Design (ICCAD'03)*, 2003. Cited on page 227.

Gérard Berry. Circuit Design and Verication with Esterel v7 and Esterel Studio. In *High Level Design Validation and Test Workshop (HLVDT 2007)*. IEEE, 2007. Cited on page 227.

Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *ACM Symposium on Languages, Compilers and Tools for Embedded Systems (LCTES'08)*, 2008. Cited on pages 46, 47, 110, 164, 225, and 226.

G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, Feb 1996. Cited on pages 229 and 230.

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), 2012. Cited on pages 107 and 234.

Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. Guarded Dependent Type Theory with Coinductive Types. In *Foundations of Software Science and Computation Structures (FoSSaCS'16)*. Springer, 2016. Cited on page 234.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In *International Conference on Functional Programming (ICFP'98)*. ACM, 1998. Cited on page 233.

Bruno Bodin. *Analyse d'applications flot de données pour la compilation multiprocesseur*. PhD thesis, Université Pierre et Marie Curie, 2013. Cited on page 230.

Bruno Bodin, Alix Munier Kordon, and Benoît Dupont de Dinechin. Periodic schedules for Cyclo-Static Dataflow. In *ESTImedia*, 2013. Cited on pages 232 and 238.

Julien Boucaron, Robert De Simone, and Jean-Vivien Millo. Formal methods for scheduling of latency-insensitive designs. *EURASIP journal on Embedded Systems*, 2007(1):8–8, 2007. Cited on page 235.

Frédéric Boussinot and Robert De Simone. The SL Synchronous Language. *Software Engineering, IEEE Transactions on*, 22(4):256–266, 1996. Cited on page 228.

Dmitry Bufistov, Jorge Júlvez, and Jordi Cortadella. Performance optimization of elastic systems using buffer resizing and buffer insertion. In *International Conference on Computer-Aided Design (ICCAD'08)*. IEEE, 2008. Cited on page 235.

B. Cao, K. A. Ross, M. A. Kim, and S. A. Edwards. Implementing latency-insensitive dataflow blocks. In *International Conference on Formal Methods and Models for System Design (MEMOCODE'15)*. ACM, 2015. Cited on page 235.

Luca P Carloni, Kenneth L McMillan, and Alberto L Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(9):1059–1076, 2001. Cited on page 235.

Josep Carmona, Jorge Júlvez, Jordi Cortadella, and Michael Kishinevsky. A scheduling strategy for synchronous elastic designs. *Fundamenta Informaticae*, 108(1-2):1–21, 2011. Cited on page 235.

Paul Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94(1):125–140, 1992. Cited on pages 47 and 224.

Paul Caspi and Nicolas Halbwachs. A functional model for describing and reasoning about time behaviour of computing systems. *Acta Informatica*, 22(6):595–627, 1986. Cited on pages 31 and 47.

Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *International Conference on Functional Programming (ICFP'96)*. ACM, 1996. Cited on pages 13, 17, 52, 164, 167, 188, 221, 222, 224, and 250.

Paul Caspi and Marc Pouzet. A co-iterative characterization of synchronous stream functions. *Electronic Notes in Theoretical Computer Science*, 11:1–21, 1998. Cited on page 165.

Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Symposium on Principles of Programming Languages (POPL'87)*, 1987. Cited on pages 13, 46, and 223.

Liang-Fang Chao and Edwin Hsing-Mean Sha. Scheduling data-flow graphs via retiming and unfolding. *Transactions on Parallel and Distributed Systems, IEEE Transactions on*, 8(12): 1259–1267, 1997. Cited on page 232.

James Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009. Cited on page 250.

Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. *N*-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. In *Symposium on Principles of Programming Languages (POPL'06)*, 2006. Cited on pages 14, 46, 48, 109, 225, and 232.

Albert Cohen, Louis Mandel, Florence Plateau, and Marc Pouzet. Abstraction of Clocks in Synchronous Data-flow Systems. In *The Sixth Asian Symposium on Programming Languages and Systems (APLAS 2008)*, 2008. Cited on pages 225, 226, 236, and 237.

Albert Cohen, Léonard Gérard, and Marc Pouzet. Programming parallelism with futures in lustre. In *International Conference on Embedded Software (EMSOFT'12)*. ACM, 2012. Cited on page 246.

Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. In *Synchronous Languages, Applications, and Programming*, volume 65. Electronic Notes in Theoretical Computer Science, 2002. Cited on page 224.

Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *International Conference on Embedded Software (EMSOFT'03)*. ACM, 2003. Cited on page 224.

Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005. Cited on page 224.

Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Mixing Signals and Modes in Synchronous Data-flow Systems. In *ACM International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October 2006. Cited on page 224.

Frederic Commoner, Anatol W. Holt, Shimon Even, and Amir Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5(5):511–523, 1971. Cited on page 230.

Thierry Coquand, Carl Gunter, and Glynn Winskel. Domain theoretic models of polymorphism. *Information and Computation*, 81(2):123–167, 1989. Cited on pages 195 and 196.

Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. Self: Specification and design of synchronous elastic circuits. In *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU'06)*. IEEE, 2006. Cited on page 235.

Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009. Cited on page 235.

Pascal Cuoq and Marc Pouzet. Modular Causality in a Synchronous Stream Language. In *European Symposium on Programming (ESOP'01)*. Springer, 2001. Cited on pages 13, 64, and 224.

Gwenaël Delaval, Léonard Gérard, Adrien Guatto, Hervé Marchand, Cédric Pasteur, Marc Pouzet, and Éric Rutten. The Heptagon synchronous language. `http://heptagon.gforge.inria.fr`, 2012. Cited on pages 188 and 242.

Alain Demeure, Anne Lafage, Emmanuel Boutillon, Didier Rozzonelli, Jean-Claude Dufourd, and Jean-Louis Marro. Array-OL: Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *15° Colloque sur le traitement du signal et des images*. Groupe d'Etudes du Traitement du Signal et des Images (GRESTI), 1995. Cited on page 230.

Thomas Ehrhard. A categorical semantics of constructions. In *Symposium on Logic in Computer Science (LICS'88)*. IEEE, 1988. Cited on page 195.

Johan Eker and Jorn W. Janneck. CAL language report: Specification of the CAL actor language, 2003. Cited on page 230.

Johan Eker, Jorn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. Taming Heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003. Cited on page 230.

Conal Elliott and Paul Hudak. Functional Reactive Animation. In *International Conference on Functional Programming (ICFP'97)*. ACM, 1997. Cited on page 232.

Esterel Technologies. The Esterel v7 Reference Manual, 2005. Version v7_30 - initial IEEE standardization proposal. Cited on pages 16, 110, 226, and 227.

Esterel Technologies. SCADE Suite. `http://www.esterel-technologies.com/products/scade-suite/`, 2015. Cited on pages 14 and 188.

Paul Feautrier. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International journal of parallel programming*, 21(5):313–347, 1992a. Cited on page 230.

Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International journal of parallel programming*, 21(6):389–420, 1992b. Cited on page 230.

Paul Feautrier. Scalable and structured scheduling. *International Journal of Parallel Programming*, 2006. Cited on page 230.

Matthias Felleisen. On the expressive power of programming languages. In *European Symposium on Programming (ESOP'90)*. Springer, 1990. Cited on page 175.

Julien Forget, Fréderic Boniol, Daniel Lesens, and Claire Pagetti. A Multi-Periodic Synchronous Data-Flow Language. In *Symposium on High-Assurance Systems Engineering (HASE'08)*. IEEE, 2008. Cited on pages 225 and 246.

Peter Gammie. Synchronous digital circuits as functional programs. *ACM Computing Surveys*, 46(2):21:1–21:27, November 2013. ISSN 0360-0300. doi: 10.1145/2543581.2543588. URL http://doi.acm.org/10.1145/2543581.2543588. Cited on page 233.

Mike Gemünde, Jens Brandt, and Klaus Schneider. Clock refinement in imperative synchronous languages. *EURASIP Journal on Embedded Systems*, 2013(1):1–21, 2013. Cited on pages 47 and 228.

Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. A Modular Memory Optimization for Synchronous Data-flow Languages: Application to Arrays in a Lustre Compiler. In *International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'12)*. ACM, 2012. Cited on pages 16 and 239.

Dan R. Ghica. Geometry of Synthesis: A Structured Approach to VLSI Design. In *Symposium on Principles of Programming Languages (POPL'07)*. ACM, 2007. Cited on pages 17, 166, and 234.

Dan R. Ghica and Mohamed N. Menaa. On the compositionality of round abstraction. In *CONCUR 2010-Concurrency Theory*, pages 417–431. Springer, 2010. Cited on page 166.

Dan R. Ghica and Alex Smith. Geometry of Synthesis II: From Games to Delay-Insensitive Circuits. In *Mathematical Foundations of Programming Semantics (MFPS'10)*. Elsevier, 2010. Cited on pages 166 and 234.

Dan R. Ghica and Alex Smith. Geometry of Synthesis III: Resource Management Through Type Inference. In *Symposium on Principles of Programming Languages (POPL'11)*. ACM, 2011. Cited on pages 166 and 234.

Dan R. Ghica, Alex Smith, and Satnam Singh. Geometry of Synthesis IV: Compiling Affine Recursion Into Static Hardware. In *International Conference on Functional Programming (ICFP'11)*. ACM, 2011. Cited on pages 166 and 234.

Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987. Cited on pages 58, 59, 111, and 154.

Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded Linear Logic: a Modular Approach to Polynomial-time Computability. *Theoretical Computer Science*, 97(1):1–66, 1992. Cited on page 167.

Alain Girault. A survey of automatic distribution method for synchronous programs. In *International Workshop on Synchronous Languages, Applications and Programs (SLAP'05)*, 2005. Cited on page 246.

Georges Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones application à Esterel*. PhD thesis, Université Paris-Sud, 1988. Cited on page 248.

Michael I. Gordon. *Compiler techniques for scalable performance of stream programs on multicore architectures*. PhD thesis, Massachusetts Institute of Technology, 2010. Cited on page 229.

Léonard Gérard. *Programmer le parallélisme avec des futures en Heptagon un langage synchrone flot de données et étude des réseaux de Kahn en vue d'une compilation synchrone*. PhD thesis, Université Paris-Sud, 2013. Cited on pages 47 and 249.

N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991. Cited on pages 164 and 224.

Nicolas Halbwachs. *Modelling and analysis of timed computer system behaviour*. Habilitation à Diriger des Recherches, Institut National Polytechnique de Grenoble ; Université Joseph-Fourier, 1984. Cited on page 31.

Nicolas Halbwachs. A Synchronous Language at Work: the Story of Lustre. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE'05)*. IEEE, 2005. Cited on pages 16, 47, and 110.

Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous Observers and the Verification of Reactive Systems. In *International Conference on Algebraic Methodology and Software Technology (AMAS'93)*. Springer, 1994. Cited on page 154.

Grégoire Hamon and Marc Pouzet. Modular Resetting of Synchronous Data-flow Programs. In *International Symposium on Principles and Practice of Declarative Programming (PPDP'00)*. ACM, 2000. Cited on page 240.

IEEE. VHDL Register Transfer Level (RTL) Synthesis. *IEEE Std 1076.6-1999*, 2000. Cited on page 162.

IEEE. Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006. Cited on page 162.

IEEE. VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 2009. Cited on page 162.

Bart Jacobs. *Categorical Logic and Type Theory*. Elsevier, 1999. Cited on page 199.

Alan Jeffrey. LTL types FRP: Linear-Time Temporal Logic Propositions as Types, Proofs as Functional Reactive Programs. In *Programming Languages meets Program Verification (PLPV'12)*. ACM, 2012. Cited on page 233.

Wolfgang Jeltsch. Towards a Common Categorical Semantics for Linear-Time Temporal Logic and Functional Reactive Programming. In *Mathematical Foundations of Programming Semantics (MFPS'XXVIII)*. Elsevier, 2012. Cited on page 233.

Geraint Jones and Mary Sheeran. Deriving bit-serial circuits in Ruby. In *VLSI'91*, 1991. Cited on page 233.

André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119:447–468, 4 1996. Cited on pages 17 and 165.

Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing Congress (IFIP'74)*. IFIP, 1974. Cited on pages 12, 19, and 46.

R. Karp and R. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966. doi: 10.1137/0114108. Cited on pages 47 and 230.

Neelakantan R Krishnaswami. Higher-Order Functional Reactive Programming without Spacetime Leaks. In *International Conference on Functional Programming (ICFP'13)*. ACM, 2013. Cited on pages 233 and 250.

Neelakantan R. Krishnaswami and Nick Benton. Ultrametric Semantics of Reactive Programs. In *Annual Symposium on Logic in Computer Science (LICS'11)*. IEEE, 2011. Cited on pages 107 and 234.

Peter J Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964. Cited on page 57.

Nhat Minh Lê, Adrien Guatto, Albert Cohen, and Antoniu Pop. Correct and Efficient Bounded FIFO Queues. In *Computer Architecture and High Performance Computing (SBAC-PAD'13)*. IEEE, 2013. Cited on page 246.

Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991. Cited on pages 19 and 226.

Hervé Le Verge, Christophe Mauras, and Patrice Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI signal processing systems for signal, image and video technology*, 3(3):173–182, 1991. Cited on page 230.

Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. Cited on pages 18, 47, 229, and 230.

Charles E. Leiserson and James B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 1991. Cited on page 16.

Frédéric Mallet. Clock Constraint Specification Language: Specifying Clock Constraints with UML/MARTE. *Innovations in Systems and Software Engineering*, 4(3):309–314, 2008. Cited on page 232.

Frédéric Mallet, Julien DeAntoni, Charles André, and Robert De Simone. The clock constraint specification language for building timed causality models. *Innovations in Systems and Software Engineering*, 6(1-2):99–106, 2010. Cited on page 232.

Louis Mandel and Florence Plateau. Scheduling and buffer sizing of n-synchronous systems: Typing of ultimately periodic clocks in Lucy-n. In *Mathematics of Program Construction (MPC'12)*. Springer, 2012. Cited on pages 225, 226, 236, and 237.

Louis Mandel and Marc Pouzet. ReactiveML: a reactive extension to ML. In *International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*. ACM, 2005. Cited on pages 110 and 228.

Louis Mandel, Florence Plateau, and Marc Pouzet. Lucy-n: a n-Synchronous Extension of Lustre. In *Mathematics of Program Construction (MPC'10)*. Springer, 2010. Cited on pages 14, 46, 109, 110, 188, 225, and 226.

Louis Mandel, Florence Plateau, and Marc Pouzet. Static Scheduling of Latency Insensitive Designs with Lucy-n. In *Formal Methods in Computer Aided Design (FMCAD'11)*, 2011. Cited on page 235.

Louis Mandel, Cédric Pasteur, and Marc Pouzet. Time refinement in a functional synchronous language. In *International Symposium on Principles and Practice of Declarative Programming (PPDP'13)*. ACM, 2013. Cited on pages 47 and 228.

Paul-André Melliès and Jérôme Vouillon. Recursive polymorphic types and parametricity in an operational framework. In *Symposium on Logic in Computer Science (LICS'05)*. IEEE, 2005. Cited on page 158.

Jean-Vivien Millo and Robert De Simone. Periodic scheduling of marked graphs using balanced binary words. *Theoretical Computer Science*, 458:113–130, 2012. Cited on page 235.

Lionel Morel. Efficient compilation of array iterators for Lustre. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'02)*. Elsevier, 2002. Cited on page 16.

Alan Mycroft and Richard Sharp. A Statically Allocated Parallel Functional Language. In *International Conference on Automata, Languages and Programming (ICALP'00)*. Springer, 2000. Cited on page 234.

Hiroshi Nakano. A Modality for Recursion. In *Symposium on Logic in Computer Science (LICS'00)*. IEEE, 2000. Cited on page 234.

Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *SIGPLAN Haskell Workshop (Haskell'02)*. ACM, 2002. Cited on page 233.

Marc Galceran Oms, Jordi Cortadella, and Michael Kishinevsky. Symbolic performance analysis of elastic systems. In *International Conference on Computer-Aided Design (ICCAD'10)*. IEEE, 2010. Cited on page 235.

Keshav K. Parhi and David G. Messerschmitt. Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding. *Computer, IEEE Transactions on*, 40(2):178–195, 1991. Cited on pages 16, 47, and 232.

Cédric Pasteur. *Raffinement temporel et exécution parallèle dans un langage synchrone fonctionnel*. PhD thesis, Université Pierre et Marie Curie, 2013. Cited on page 110.

Benjamin C Pierce. *Types and programming languages*. MIT press, 2002. Cited on page 188.

Andrew M. Pitts. Relational Properties of Domains. *Information and Computation*, 127(2):66 – 90, 1996. Cited on page 25.

Andrew M. Pitts. Parametric Polymorphism and Operational Equivalence. *Mathematical Structures in Computer Science*, 10(3):321–359, 2000. Cited on page 154.

Florence Plateau. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée*. PhD thesis, Université Paris-Sud, 2010. Cited on pages 20, 31, 32, 41, 47, 50, 109, 188, 226, 236, and 238.

Dumitru Potop-Butucaru, Benoît Caillaud, and Albert Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, 2006. Cited on page 227.

Dumitru Potop-Butucaru, Stephen A Edwards, and Gérard Berry. *Compiling Esterel*, volume 86. Springer Science & Business Media, 2007. Cited on pages 228 and 240.

Marc Pouzet. *Lucid Synchrone: un langage synchrone d'ordre supérieur*. Paris, France, 14 novembre 2002. Habilitation à diriger les recherches. Cited on page 53.

Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Cited on pages 46, 225, and 226.

Marc Pouzet and Pascal Raymond. Modular static scheduling of synchronous data-flow networks. *Design Automation for Embedded Systems*, 14(3):165–192, 2010. Cited on page 249.

Pascal Raymond. Compilation séparée de programmes lustre. Technical report, Projet SPECTRE, IMAG, july 1988. Cited on pages 248 and 249.

Pascal Raymond. *Compilation efficace d'un langage déclaratif synchrone : Le générateur de code Lustre-V3*. PhD thesis, Institut National Polytechnique de Grenoble, 1991. Cited on pages 110 and 164.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Reprinted from the proceedings of the 25th ACM National Conference*, pages 717–740. ACM, 1972. Cited on page 57.

John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1996. Cited on page 25.

John C. Reynolds. The Meaning of Types — From Intrinsic to Extrinsic Semantics. Technical report, University of Aarhus, Department of Computer Science, BRICS, 2000. Cited on page 110.

Frédéric Rocheteau. *Extension of the Lustre language and application to hardware design: the Lustre-v4 language and the Pollux system.* PhD thesis, Institut National Polytechnique de Grenoble, 1992. Cited on pages 164 and 227.

Frédéric Rocheteau and Nicolas Halbwachs. Implementing Reactive Programs on Circuits: a Hardware Implementation of LUSTRE. In *Real-Time: Theory in Practice (REX Workshop'92)*, 1992. Cited on page 164.

Dana S. Scott. A Type-theoretical Alternative to ISWIM, CUCH, OWHY. *unpublished paper from 1969 later published in Theoretical Computer Science*, 121(1-2):411–440, December 1969. ISSN 0304-3975. Cited on pages 20 and 25.

Ellen Sentovich, Horia Toma, and Gérard Berry. Latch optimization in circuits generated from high-level descriptions. In *International Conference on Computer Assisted Design (ICCAD'96)*, 1996. Cited on page 227.

Mary Sheeran. Slowdown and Retiming in Ruby. In *IFIP Workshop on The Fusion of Hardware Design and Verification*, 1988. Cited on page 234.

Mary Sheeran. Hardware Design and Functional Programming: a Perfect Match. *Journal of Universal Computer Science*, 2005. Cited on page 233.

Thomas Streicher. *Domain-Theoretic Foundations of Functional Programming.* World Scientific, 2006. Cited on pages 24 and 25.

Bart D Theelen, Marc CW Geilen, Twan Basten, Jeroen PM Voeten, Stefan Valentin Gheorghita, and Sander Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE'06)*. IEEE, 2006. Cited on page 230.

William Thies. *Language and Compiler Support for Stream Programs.* PhD thesis, Massachusetts Institute of Technology, 2009. Cited on page 229.

William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Compiler Construction*, pages 179–196. Springer, 2002. Cited on page 229.

Stavros Tripakis, Dai Bui, Marc Geilen, Bert Rodiers, and Edward A Lee. Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(3):83, 2013. Cited on page 231.

Jaap Van Oosten. *Realizability: an introduction to its categorical side*, volume 152. Elsevier, 2008. Cited on page 111.

Jean E. Vuillemin. On circuits and numbers. *Computers, IEEE Transactions on*, 43:868–879, 1994. Cited on page 160.

William W. Wadge and Edward A. Ashcroft. *LUCID, the Dataflow Programming Language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985. Cited on page 229.

Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Practical Aspects of Declarative Languages (PADL'02)*. Springer, 2002. Cited on page 233.

Joe B. Wells. The Essence of Principal Typings. In *International Conference on Automata, Languages and Programming (ICALP'02)*. Springer, 2002. Cited on page 247.

Glynn Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993. Cited on pages 25 and 28.

Ling Yin, Jing Liu, Zuohua Ding, Frédéric Mallet, and Robert De Simone. Schedulability Analysis with CCSL Specifications. In *Asia-Pacific Software Engineering Conference (APSEC'13)*. IEEE, 2013. Cited on page 232.

Herve Yviquel, Antoine Lorence, Khaled Jerbi, Gildas Cocherel, Alexandre Sanchez, and Mickael Raulet. Orcc: Multimedia Development Made Easy. In *International Conference on Multimedia (MM'13)*. ACM, 2013. Cited on page 230.

Résumé

Cette thèse traite de la conception et implémentation d'un langage de programmation pour les systèmes de traitement de flux en temps réel, comme l'encodage vidéo. Le modèle des réseaux de Kahn est bien adapté à ce domaine et y est couramment utilisé. Dans ce modèle, un programme consiste en un ensemble de processus parallèles communicant à travers des files mono-producteur, mono-consommateur. La force du modèle réside en son déterminisme.

Les langages synchrones fonctionnels comme Lustre sont dédiés aux systèmes embarqués critiques. Un programme Lustre définit un réseau de Kahn *synchrone* qui peut être exécuté avec des files bornées et sans blocage. Cette propriété est garantie par un système de types dédié, le *calcul d'horloge*, qui établit une échelle de temps globale à un programme. Cette échelle de temps globale est utilisée pour définir les *horloges*, séquences booléennes indiquant pour chaque file, et à chaque pas de temps, si un processus produit ou consomme une donnée. Cette information sert non seulement à assurer la synchronie mais également à générer du logiciel ou matériel à état fini.

Nous proposons et étudions les *horloges entières*, une généralisation des horloges booléennes autorisant des entiers naturels arbitrairement grands. Les horloges entières décrivent la production ou consommation de plusieurs valeurs depuis une même file au cours d'un instant. Nous les utilisons pour définir la construction d'*échelle de temps locale*, qui peut masquer des pas de temps cachés par un sous-programme au contexte englobant.

Ces principes sont intégrés à un calcul d'horloge pour un langage fonctionnel d'ordre supérieur. Nous étudions ses propriétés et prouvons en particulier que les programmes bien typés ne bloquent pas. Nous compilons les programmes typés vers des circuits numériques synchrones en adaptant le schéma de génération de code dirigé par les horloges de Lustre. L'information de typage contrôle certains compromis entre temps et espace dans les circuits générés.

Mots-clés

Langages de programmation fonctionnels; langages de programmation synchrones; systèmes de types; compilation; circuits numériques synchrones.

Abstract

This thesis addresses the design and implementation of a programming language for real-time streaming applications, such as video decoding. The model of Kahn process networks is a natural fit for this area and has been used extensively. In this model, a program consists in a set of parallel processes communicating via single reader, single writer queues. The strength of the model lies in its determinism.

Synchronous functional languages such as Lustre are dedicated to critical embedded systems. A Lustre program defines a *synchronous* Kahn process network, that is, which can be executed using finite queues and without deadlocks. This is enforced by a dedicated type system, the *clock calculus*, which establishes a global time scale throughout a program. The global time scale is used to define *clocks*: per-queue boolean sequences indicating, for each time step, whether a process produces or consumes a token in the queue. This information is used both for enforcing synchrony and for generating finite-state software or hardware.

We propose and study *integer clocks*, a generalization of boolean clocks featuring arbitrarily big natural numbers. Integer clocks model the production or consumption of several values from the same queue in the course of a time step. We then rely on integer clocks to define the *local time scale* construction, which may hide time steps performed by a sub-program from the surrounding context.

These principles are integrated into a clock calculus for a higher-order functional language. We study its properties, proving among other results that well-typed programs do not deadlock. We adjust the clock-directed code generation scheme of Lustre to generate finite-state digital synchronous circuits from typed programs. The typing information controls certain trade-offs between time and space in the generated circuits.

Keywords

Functional programming languages; synchronous programming languages; type systems; compilation; digital synchronous circuits.

Numéro national
de thèse : 70793