Complexité Implicite de Lambda-Calculs Concurrents

Antoine Madet

Université Paris Diderot (Paris 7)
Laboratoire Preuves, Programmes, Systèmes
École Doctorale Sciences Mathématiques de Paris Centre

Thèse de doctorat
Spécialité informatique

Complexité Implicite de Lambda-Calculs Concurrents

Antoine Madet

Soutenue le 6 décembre 2012
devant le jury composé de:

M. Roberto Amadio	*Directeur*
M. Patrick Baillot	*Directeur*
M. Ugo Dal Lago	*Rapporteur*
M. Jean-Yves Marion	*Rapporteur*
M. Virgile Mogbil	*Examinateur*
M. François Pottier	*Examinateur*

Remerciements

Je voudrais d'abord remercier chaleureusement mes directeurs de thèse Roberto et Patrick de m'avoir encadré tout au long de ces trois années de recherche et, semble-t-il, guidé vers une bonne fin ! Je remercie Roberto pour sa présence constante et Patrick pour son soutien continu.

Je voudrais également remercier Ugo Dal Lago et Jean-Yves Marion de s'être intéressés à mon travail depuis mes premiers exposés et d'avoir relu mon manuscrit avec minutie. Merci également à Virgile Mogbil et François Pottier d'avoir accepté de participer à mon jury.

Merci à Aloïs pour sa collaboration enthousiaste et fructueuse.

Pour le très agréable cadre de travail qui y règne, je remercie tous les chercheurs du laboratoire PPS. Merci à Odile de toujours être à nos petits soins.

Pour le très détendu cadre de vie (et de travail) qui y règne, merci à mes cothésards du bureau 6C10, à Stéphane et ses mains pleines de fusain, à Thibaut pour son partage optimal de la science, à Fabien ou le poulet, à Jonas pour son humeur toujours oplax et à Gabriel pour son style par passage-de-science. Merci également aux anciens, aux nouveaux, à ceux d'en face: Mehdi, Alexis, Guillaume, Shain, Kuba, Johanna, et tous les autres... Un merci spécial à Beniamino pour les déambulations nocturnes dans les ruelles parisiennes.

Merci également à tous les aventuriers de l'Oregon (Matthias, Andrei, Clément, Aloïs) et aux musiciens geeks (Sam, David, Vincent, Damiano, Matthieu).

Enfin merci à tous les copains sans qui cette thèse aurait été moins drôle: Luis (jazz'n'tapas), Charlie (jazz'n'roll), Elodie (blonde'n'blonde), la bande à Mylène-Bexen-Edlira-Giulia. Les vieux copains de Bordeaux Ugo, Mikaël. Sans oublier le cool cat Gilles et tous les autres !

Merci beaucoup à Magali pour son univers pétillant.

Merci à John C., Sonny R. et Florent D. pour leurs musiques et le reste.

Paris, le 21 Novembre 2012.

À Papa, Maman, Léa et Théo.

Implicit Complexity in Concurrent Lambda-Calculi

Contents

Chapter 1

Introduction

As programs perform tasks, they consume resources of the computational systems on which they are executed. These resources are by definition of limited availability and typically include processor cycles, memory accesses, input/output operations, network accesses, etc... The efficiency of programs highly depends on the usage of these resources and therefore it is of chief importance to be able to control the resource consumption of programs.

The control of resource usages has also many applications in the field of *computer security*. For example, the increasing mobility of programs raises many situations where an untrusted code may overuse resources of a host computational system and provoke a denial of service. Also, the important development of computational systems with critical amount of resources like smart cards and embedded systems appeals for an analysis of the resource consumption of programs.

The resources of computational systems differ quite a lot from one architecture to another and the analysis of their usages requires an understanding of very specific low-level details. Rather than computational *systems*, it is often simpler to work at the more abstract level of computational *models*. In these models, programs are written in abstract languages and consume abstract resources that usually include *computational time*, a certain number of computation steps, and *computational space*, a certain amount of memory space. The efficiency of programs, which is also known as their *computational complexity*, is then evaluated by analyzing the amount of abstract resources that is consumed for a given set of input parameters. For example, it is with respect to abstract resources that a program is said to be computable in polynomial time.

Analyzing the resource usages of programs can be done in two different ways. One of them is to measure *dynamically* (at run-time) the consumption of resources and to abort execution when safety limits are overreached. The drawback of this approach is that dynamic measurements introduce overhead costs

which may be critical if few amounts of resources are available. More importantly, in critical systems it may simply not be acceptable to abort the computation. The alternative approach is to compute *statically* (at compile-time) the resource usages of programs, so that it can be decided before their actual executions if they are safe. Also, static analyzes do not introduce any runtime cost.

Programmers themselves have the ability to perform manual static resource analyzes of programs, up to some precision... Since they usually work with *high-level* programming languages that hide the machinery of *low-level* resources like processor cycles, programmers reason at the more abstract level of computational complexity. They assign a *cost* to each operation of the programming language, that is a number of computation steps and/or memory units, and then evaluate what is the cost of their program for given inputs. Of course, manual static analyzes do not scale to realistic programs. Moreover, even though static analyzes can be automatically computed, it is hard to infer precise costs. We identify mainly two causes to this difficulty:

(1) The high-level nature of mainstream programming languages do not allow to analyze concrete resource usages like processor time and memory accesses. Even though we can reason at the more abstract level of computational resources, they are not obviously related to the consumption of concrete resources. High-level programs are most of the time compiled into low-level ones whose concrete resource usages are easier to observe, but the numerous compilation steps obscure very much the relationship between high-level programs and their low-level counterparts.

(2) Even if we focus on the abstract level of computational resources, programming languages frequently offer various features like higher-order functions, imperative side effects, multi-threading, object creation, etc... that when used together complicate very much the computation process. In these cases, computational complexity is hard to determine.

In this thesis, we address item (2). We are interested in static methods to analyze and control the consumption of computational resources by programs. In particular, we would like to focus on programs that are *higher-order* and *concurrent*.

1.1 Higher-order concurrent programs

Higher-order concurrent programs are those written by a combination of these two features:

- *Higher-order functions*: functions are first-class values which can be passed as arguments to other functions and which can be returned as values.

- *Concurrency*: programs are composed of parallel threads that interact through a shared state by *e.g.* receiving/sending messages on channels or reading/writing references.

The notion of higher-order function is very much central in the family of functional languages such as Ocaml or Haskell. These languages are in fact based on a fundamental core which has been introduced by A. Church in the 1930s as the 'λ-calculus' [Chu33]. This minimal yet powerful language embodies the concept of *function abstraction*, as well as the concept of *data abstraction* when equipped with a suitable type system.

In these programming languages (Ocaml, Haskell), it is also possible to write concurrent programs and various interaction mechanisms (references, channels) are available. However, the λ-calculus is not well-suited for concurrent programs: there is no internal notion of state and the result of the computation is always deterministic. Here is a typical example of program written in OCaml that uses the above features.

```
# let l = [ref 2; ref 2; ref 2] in
  let update r = r := !r * 2 in
  let iter_update l = List.iter update l;;
# Thread.create iter_update l;;
# Thread.create iter_update l;;
```

We first create a list `l` of references containing the integer 2. We define a function `update` that multiplies the content of a reference, and we define a function `iter_update` that iterates the function `update` on a given list. Then, we create two threads that both apply the function `iter_update` on the list `l`. Thus, one possible program execution updates the content of each reference to 8:

```
# l;;
- : int ref list = [{contents = 8}; {contents = 8}; {contents = 8}]
```

Before analyzing the resource usages of a program, a question that we may ask is the following: does the program even consume a *finite* amount of computational resources? Obviously, the above program is terminating and thus consumes a finite amount of resources. However, in some cases the combination of imperative side effects and higher-order functions is known to produce diverging computations. Consider the following program that uses the so-called *Landin's trick*.

```
# let r = ref (fun x -> x) in
  r := fun x -> (!r)x;
  !r();;
...
```

We first initialize a location `r` with the identity function. Then we assign to `r` a function that, when given an argument `x`, applies the content of `r` to `x`.

Consequently, when we apply the content of `r` to the unit value `()`, the program keeps running forever.

Another question that we may ask is this one: does every execution of a given program consume the same amount of resources? Consider the following program.

```
# let r = ref (fun x -> x) in
  let f g = r := g in
  let t1 = Thread.create f (fun x -> (!r)x) in
  let t2 = Thread.create f (fun x -> x);;
```

The order of execution of the two threads `t1` and `t2` is non-deterministic so that in some cases the following expression

```
# !r();;
```

diverges while in other cases it terminates.

The above program examples suggest that the static analysis of resource usages of concurrent programs written with higher-order functions is rather difficult.

1.2 Proofs as programs

Instead of trying to analyze any program that could possibly be written, another approach is to force the programmer to write programs that, by construction, use restricted amounts of resources. A well-known proposal in this direction is to express the constraints by means of a *type system*. More precisely, it consists in associating a type to each sub-expression of a program in order to limit the kind of values the sub-expression may produce during execution. The constraints are actually expressed as typing rules that specify how expressions of given types can be composed. The goal is to define suitable typing rules so that if a program is *well-typed* (*i.e.* it can be given a type by following the rules), then it uses a 'safe' amount of resources. Finally, the static analysis only consists in trying to infer the type of programs.

Several interpretations of 'safe amount' of resources are possible. The first degree of safety that we may ask for is the consumption of a finite amount of computational time, *i.e.* the termination property. The relationship between type systems and termination has been extensively studied through the well-known *Curry-Howard* correspondence which establishes a direct relation between intuitionistic proofs and typed λ-terms, as depicted in the following table.

Intuitionistic Logic	λ-calculus
Formula	Type
Proof	Typed λ-term
Proof reduction	Term reduction
Normalization	Termination

The fact that every intuitionistic proof can be reduced to a normal form (*i.e.* which cannot be reduced further), which is called the normalization property, corresponds to the fact that every typed λ-term terminates.

Unfortunately, the simply typed λ-calculus is a very restricted fragment of Ocaml. More generally, ML programs do not correspond to intuitionistic proofs because features such as recursion and references are not reflected by intuitionistic logic. Consequently, the ML type system does not ensure the termination of programs; it suffices to check that the above program examples are well-typed.

1.3 Implicit Computational Complexity

Type systems have also been used to guarantee properties stronger than termination. For example, by designing sufficiently constrained typing rules, well-typed programs can be proved to terminate in *e.g.* polynomial time or logarithmic space. More generally, numerous other approaches have been proposed to constrain the complexity of programs. In fact, the design of programming languages that use safe amounts of resources has been very much inspired by the research field of *Implicit Computational Complexity* (ICC). This research area aims at providing logical principles or language restrictions to characterize various complexity classes. Here, "implicit" means that the restrictions do not refer to any specific machine model or external measuring conditions. The first implicit characterizations of bounded complexity were given by D. Leivant [Lei91] and then by S. Bellantoni and S. Cook [BC92]. By imposing the principle of *data-ramification*, they are able to characterize functions computable in polynomial time. Following these seminal works, various other approaches have been proposed in the literature such as logical principles, rewriting techniques, semantic interpretations...

What interests us is that ICC has found a natural application in the design of programming languages that are endowed with static criteria ensuring bounds on the computational complexity of programs. In this respect, while static criteria should guarantee reasonable complexity bounds, they should also allow sufficient flexibility to the programmer. The programming flexibility of an ICC criteria can be determined by the number of "natural" algorithms and programming features (*e.g.* higher-order functions, imperative side effects, concurrency) that are supported. Most of the times, ICC criteria are *extensionally complete* for *e.g.* polynomial time in the sense that every *mathematical function* computable in polynomial time can be represented by a program which satisfies the criteria. However, they are not *intensionally complete* in the sense that not every polynomial time *program* satisfies the criteria. Therefore, an important line of research consists in improving the intensional expressivity of these ICC criteria.

1.4 Light Logics

One well-known instance of ICC which combines nicely with higher-order functional languages is the framework of *Light Logics* [Gir98] that originates from Linear Logic [Gir87].

Linear Logic

In Linear Logic, there is a distinction between formulae that are linear, *i.e.* that can be used exactly once, and formulae that can be used arbitrarily many times and that must be marked with a modality '!' named *bang*. The discovery of Linear Logic led to a refinement of the proof-as-program correspondence that includes an explicit treatment of the process of *data duplication*, as depicted in the following table.

Linear Logic	Linear λ-calculus
Linear formula	Linear type
Modal formula	Modal type
Proof	Typed linear λ-term
Proof reduction: - Consumption of formulae - Duplication of formulae	Term reduction: - Consumption of data - Duplication of data
Normalization	Termination

Proofs of Linear Logic now correspond to a linear λ-calculus which is a λ-calculus with a '!' constructor to mark duplicable data. The point is that the distinction between linear and modal formulae splits proof reductions into two kinds: those that *consume* linear formulae and those that *duplicate* modal formulae. At the level of terms, this corresponds to distinguishing reduction steps which are linear (functions use their arguments exactly once) and reductions steps which may duplicate/erase data.

Light Logics

Light Logics refine further Linear Logic by imposing restrictions on the bang modality so that the duplication process is restricted. To see why the duplication of data impacts on computational complexity, consider the following program.

```
# let f l = l @ l in
  f(f(f[1]));;
- : int list = [1; 1; 1; 1; 1; 1; 1; 1]
```

We define a function `f` that takes a list `l` as argument and appends `l` to itself. Then we iterate 3 times the function `l` on a list of one element so that we obtain

a list of 8 elements. It is straightforward to see that if we iterate n times the function `f`, then we obtain a list of length 2^n. Here, the exponential growth of the size of the program is due to the fact that the function `f` uses its argument twice.

Light Logic can be seen as a way to bound the complexity of the normalization/termination procedure. This is illustrated in the following table.

Light Logic	Light λ-calculus
Linear formula	Linear type
Weak modal formula	Weak modal type
Proof	Typed light λ-term
Proof reduction:	Term reduction:
- Consumption of formulae	- Consumption of data
- Weak duplication of formulae	- Weak duplication of data
Bounded complexity of normalization	Bounded complexity of termination

In Light Logics, the modality is *weaker* than in Linear Logic in the sense that it does not allow to duplicate formulae in an unrestricted way. At the term level, this amounts to weakening the duplicating power of programs. These restrictions allow to show that the normalization of proofs is of bounded complexity and consequently that typed light λ-terms terminate by consuming bounded amounts of computational resources.

To be precise, Intuitionistic Logic and Linear Logic already induce bounds on the complexity of proof/term reduction. These bounds are actually not useful from the point of view of efficiency and Light Logics generally address complexity classes corresponding to *feasible* computation such as polynomial time.

In Light Logics, formulae are decorated with special modalities and each formula can be assigned a *depth* which is the number of modalities in which it is enclosed. The interesting point is that the complexity properties of Light Logics only rely on the notion of depth. The depth of a formula, thus the depth of a type, can be somehow reflected at the level of programs by modal constructors which give a depth to each sub-term of a program. Therefore, the complexity of programs can be controlled by a *depth system* which constrains how programs of given depth can be composed. The use of types can then be seen as a way to guarantee additional safety properties like ensuring that values are used in a meaningful way (*i.e.* the *progress* property).

Panorama

To summarize, Light Logics are logical and implicit characterizations of complexity classes, which have found a nice application in the design of type sys-

tems to control the computational complexity of functional programs. Here is an overview of the main Light Logics and the related type systems.

The first light logic called *Light Linear Logic* (**LLL**) was initially proposed by Girard [Gir98] as a logical system corresponding to polynomial time: by imposing suitable restrictions on the depth of occurrences, every proof of **LLL** can be normalized in polynomial time and every polynomial time function can be represented by a proof of **LLL**. Later, A. Asperti observed [Asp98] that it is possible to simplify **LLL** into an affine variant, that is where the discarding of formulae is unrestricted, and which is called *Light Affine Logic* (**LAL**). As a result of the proof-as-program correspondence, a light logic gives rise to a 'light λ-calculus' whose terms can be evaluated in the same amount of time as the cut-elimination procedure of the logic. For instance, K. Terui introduced the *Light Affine λ-calculus* [Ter07] as the programming counterpart of **LAL**. Every program of this calculus terminates in polynomial time and every polynomial time function can be represented by a term of this calculus.

Elementary Linear Logic (**ELL**) is perhaps the simplest light logic. It was originally sketched by Girard [Gir98] as a by-product of **LLL** that on the one hand has simpler constraints on the bang modality but on the other hand captures the larger complexity class of elementary time. We recall that a function is elementary if it is computable on a Turing machine in time bounded by a tower of exponentials of fixed height. Every proof of **ELL** can be normalized in elementary time and every elementary time function can be represented by a proof of **ELL**. Later on, Danos and Joinet extensively studied **ELL** [DJ03] as a programming language. It is also well-known that the affine variant of elementary linear logic, namely **EAL**, can be considered without breaking the elementary time bound.

Another well-known light logic of polynomial time is Y. Lafont's *Soft Linear Logic* (**SLL** [Laf04]). **SLL** refines the bang modality in a quite different way than **LLL** and therefore a polynomial time function is represented by a **SLL** proof that is quite different from the **LLL** one. The programming counterpart of **SLL** is the *Soft λ-calculus* that was developed by P. Baillot and V. Mogbil [BM04]. Again, the affine variant **SAL** can be safely considered.

Recently, Gaboardi *et al.* proposed a characterization of polynomial space [GMR12] by a λ-calculus extended with conditional instructions and using criteria coming from **SAL**.

Expressivity

Both **SAL** and **LAL** are *extensionally complete*: every polynomial time *function* can be represented by a program of the logic. However, they are not *intensionally complete*: not every polynomial time *algorithm* can be written in the language of the logic. An important line of research is about improving

the intensional expressivity of these light languages that do not allow to write programs in a natural way. We identify two main directions in the literature:

(1) Programming with modalities is a heavy syntactic burden that hides the operational meaning of programs. Some work has been carried out to remove bangs from the syntax of light languages while leaving them at the level of types. For example Baillot and Terui developed a type system called **DLAL** [BT09a] for the standard λ-calculus that guarantees that λ-terms terminate in polynomial time. In a similar spirit, M. Gaboardi and S. Ronchi Della Rocca developed a type system called **STA** [GR09] for the standard λ-calculus that is based on **SAL**, and Coppola *et al.* developed a type system called **ETAS** [CDLRDR08] for the call-by-value λ-calculus that is based en **EAL**.

(2) Light languages are usually variations of the λ-calculus that do not feature high-level programming constructs. Recently though, Baillot *et al.* derived from **LAL** a functional language with recursive definitions and pattern-matching [BGM10], thus providing a significant improvement over the expressivity of the usual light languages.

1.5 The challenge

The framework of Light Logic, which we have seen is deeply rooted in Linear Logic, allows to control the complexity of higher-order functional programs through the proof-as-program correspondence. In this thesis, we would like to employ Light Logics to control the complexity of higher-order *concurrent* programs.

Recently, Light Logics have been applied to a model of concurrency based on process calculi. The first attempt is from Dal Lago *et al.* who designed a *soft higher-order π-calculus* [LMS10] where the length of interactions are polynomially bounded. Also, Dal Lago and Di Giamberardino built a system of *soft session types* [LG11b] where the interaction of a session is again polynomially bounded. However, process calculi cannot be considered as high-level programming languages and do not directly embody a notion of data representation, which the λ-calculus is more adapted for. In fact, these works may be seen as part as a larger project [LHMV12] which is to analyze the complexity of *interactions* between concurrent and distributed systems rather than analyzing the termination time of a specific program.

A work which seems closer in terms of objective is the recent framework of *complexity information flow* of J-Y. Marion [Mar11], which is built on the concepts of data-ramification and secure information flow to control the complexity of sequential imperative programs. More recently, Marion and Pechoux [MP12] proposed an extension of this framework to concurrency that ensures that every well-typed and terminating program with a fixed number of threads can be

executed in polynomial time. The type system captures interesting algorithms and seems to be the first characterization of polynomial time by a concurrent language. However, this work only concerns first-order programs and does not fit into the proof-as-program correspondence. Also, as opposed to Light Logics, the complexity analysis is dependent of the termination analysis: the complexity bounds are only valid if programs terminate.

To the best of our knowledge, this thesis is the first effort to statically control the complexity of higher-order *and* concurrent programs. We aim at bringing Light Logics closer to programming by finding new static criteria that are sound for a call-by-value and non-deterministic evaluation strategy with imperative side effects.

The application of the concepts of Light Logics to higher-order concurrent programs is not without raising some crucial questions that we will have to address:

- What is the impact of side effects on the depth of values? Since the complexity bounds rely on the notion of depth, we may wonder if side effects can accommodate the complexity properties of Light Logics.
- Can we ensure termination? Light type systems which are built out of the proof-as-program correspondence ensure the termination (with complexity bounds) of programs. However, we have seen that usual type systems cannot guarantee termination when programs produce side effects (see Landin's trick). Thus it is not clear whether the complexity bounds can be ensured without assuming the termination of programs.
- Is the call-by-value strategy compatible with the complexity properties of Light Logics? The usual proof of complexity bounds in Light Logics often rely on a very specific reduction strategy which differs from call-by-value. However, imperative side effects only make sense in a call-by-value setting.

1.6 Contributions

In this thesis, we propose an extension of the framework of Light Logics to higher-order concurrent programs that provides new static criteria to bound the complexity of programs. Our developments will be presented gradually: first, we look at the termination property (and confluence); second, we consider termination in elementary time and third, we consider termination in polynomial time. The rest of this section details our contributions.

Concurrent λ-calculi

Firstly, the languages we study are concurrent λ-calculi which are based on a formalization of R. Amadio [Ama09]. In this calculi, the state of a program is abstracted into a finite set of *regions* and side effects are produced by read and

write operations on these regions. A region is an abstraction of a set of dynamically allocated values that allows to simulate various interaction mechanisms like references and channels. An operator of parallel composition permits to generate new threads that may interact concurrently.

Termination

Our first contributed language is an *affine-intuitionistic* concurrent λ-calculus in which we can distinguish between values that can be used at most once (*i.e.* affine) and values that can be duplicated (*i.e.* intuitionistic). This is made possible by the design of a type system inspired by Linear Logic which takes the duplication power of side effects into account. The difficulty of the contribution is to show the *subject reduction* (*i.e.* well-typing is preserved by reduction) of the type system in which the bang modality interacts with region types. The distinction between affine and intuitionistic values allows us to develop a proper discipline of *region usages* that can be used to ensure the confluence of programs. Moreover, we show that region usages smoothly combine with the discipline of *region stratification* which ensures the termination of program. The stratification of regions has been initially proposed by G. Boudol [Bou10] to ensure the termination of higher-order concurrent program in a purely intuitionistic setting.

The above contribution has been presented at the workshop LOLA'10 [ABM10].

Elementary time

Our second contributed language is an *elementary* concurrent λ-calculus. We provide an *elementary affine* type system inspired by **EAL** that ensures the termination of programs in elementary time under a call-by-value strategy, and their *progress* (*i.e.* they do not go wrong). In particular, the type system captures the iteration of functions producing side effects over inductive data structures.

The results are supported by the following essential contributions:

- The type system is actually built out of a more primitive *depth system* which only controls the depth of values. The functional core of the depth system is inspired by the Light Affine λ-calculus of K. Terui [Ter07] and the effectful side relies on a careful analysis of the impact of side effects on the depth of values.
- Programs well-formed in the depth system are shown to terminate in elementary time under a call-by-value strategy. The proof is based on an original combinatorial analysis of programs which does not assume termination (*i.e.* regions stratification is not necessary). Interestingly, in the

purely functional case, the combinatorial argument applies to *every* reduction strategy while previous proofs assume a specific reduction strategy.

The above contribution has been published in the proceedings of TLCA'11 [MA11].

Polynomial time

Our third contributed language is a *polynomial* concurrent λ-calculus. We provide a *light linear* type system inspired by **LLL** that ensures the termination of programs in polynomial time under a call-by-value strategy, and their *progress*. As in the elementary case, the type system captures the iteration of functions producing side effects, but it is of course less permissive.

Contrary to the elementary case, we found no combinatorial argument to bound polynomially the complexity of the call-by-value evaluation. Thus, we propose a method which follows Terui's work on the Light Affine λ-calculus [Ter07]:

1. We provide a light linear *depth system* which controls the depth of values during the reduction of programs. We are able to show that a very specific evaluation strategy terminates in polynomial time by a combinatorial argument which is simply extended from Terui's work.
2. Since the polynomial evaluation strategy is really different from call-by-value, we *try* to show that every call-by-value reduction sequence can be transformed into one of the *same length* that follows the previous evaluation strategy, which would entail that call-by-value is polynomial.

This latter transformation is non-trivial because it requires to evaluate side effects in a very liberal order. On the other hand, we show that the arbitrary evaluation of side effects may trigger an exponential blowup of the size of the computation. Therefore, our contribution is to identify a proper evaluation strategy for which the transformation succeeds (*i.e.* returns sequences of the same length) and that preserves the call-by-value semantics of the program.

The above contribution has been published in the proceedings of PPDP'12 [Mad12].

Quantitative realizability

The above static criteria (depth systems, type systems) induce complexity bounds which are proved by combinatorial analyzes and syntactic transformations. Our last contribution is to provide an alternative semantic proof of these complexity bounds. More precisely, the framework of *quantitative realizability* has been proposed by Dal Lago and Hofmann [LH11] to give semantic proofs of complexity soundness of various Light Logics. We introduce an extension of quantitative realizability to higher-order *imperative* programs, focusing on a type system inspired by **LAL**. By proving that the type system is sound with respect to the

realizability model, we obtain that every typable program terminates in polynomial time under a call-by-value strategy. In particular, we do not need to simulate call-by-value reductions by any other reductions. Moreover, the proof method is parametric in the type system and could easily be adapted to the elementary case.

Our interpretation is based on bi-orthogonality (*à la Krivine* [Kri09]), following A. Brunel's extension of quantitative realizability to a *classical* setting [Bru12]. Realizability in the presence of imperative side effects is usually difficult as it raises circularity issues; here, our semantic interpretation is indexed by depth levels which allows to define an inductive interpretation of types. This draws some interesting connections with other realizability models based on step-indexing [AM01] and Nakano's modality [Nak00].

The important drawback of the method is that, at the moment, it does not scale to multi-threaded programs.

This last contribution is joint work with Aloïs Brunel and is to published in the proceedings of APLAS'12 [BM12].

1.7 Structure

This document is structured into three parts:

I - Termination and Confluence
The first part is preliminary to the analysis of the complexity of programs. In Chapter 2, we review a concurrent λ-calculus and the discipline of region stratification. In Chapter 3, we introduce an affine-intuitionistic concurrent λ-calculus and show how termination and confluence can be ensured by region stratification and region usages, respectively.

II - Combinatorial and Syntactic Analyzes
The second part introduces static criteria to control the complexity of programs and the complexity bounds are proved by combinatorial syntactical analyzes. The first Chapter 4 and Chapter 5 deal with elementary time and the last Chapter 6 and Chapter 7 deal with polynomial time. Each time we start by reviewing the purely functional case before moving to concurrency.

III - Quantitative Realizability *(joint work with Aloïs Brunel)*
The unique Chapter 8 introduces quantitative realizability for an imperative λ-calculus. As a case study, we focus on termination in polynomial time.

Part I

Termination and Confluence

Chapter 2

A concurrent λ-calculus

We present in this chapter an extension of the λ-calculus to concurrency which is based on R. Amadio's formalization [Ama09]. In this λ-calculus, that we call $\lambda^{\|}$, the state of a program is abstracted by a constant number of *regions* and side effects are produced by read and write operations on these regions. In fact, a region is an abstraction of a set of dynamically allocated values. We will see that working at the abstract level of regions allows to simulate various interaction mechanisms like imperative references and communication channels.

In this chapter, we also present a technique to establish the termination of higher-order concurrent programs. Indeed, the property of termination may be seen as preliminary to the property of termination in bounded time, the latter being central in this thesis. The Curry-Howard correspondence establishes a well known connection between the termination of purely functional programs and types. However, side effects are not taken into account by the usual type systems and they may make programs diverge. *Type and effect* systems have been introduced by J. Lucassen and D. Gifford [LG88] to approximate the way programs act on regions. Later, they have been used by M. Tofte and J-P. Talpin [TT97] to determine statically the management of memory. Recently, G. Boudol [Bou10] introduced a discipline of *region stratification* by means of a type and effect system, to show the termination of higher-order concurrent programs. This chapter presents the discipline of region stratification in a way that has been clarified and generalized by R. Amadio [Ama09].

Outline This chapter is organized as follows. In Section 2.1 we present the syntax and the reduction of the concurrent λ-calculus $\lambda^{\|}$. The reduction rules are defined such that $\lambda^{\|}$ simulates a concurrent λ-calculus with references or channels. In Section 2.2 we present the type and effect system. Here, the effect of a program is an over approximation of the set of regions it may read or write. This type and effect system allows some form of circularity: a region r can

contain a value which has an effect on r itself, and this may lead to a diverging computation. In Section 2.3 we introduce the discipline of region stratification to prevent this kind of circularity. The intuitive idea is that a region may only produce side effects on regions which are smaller according to a well-founded order. Then we review the proof by reducibility candidates which proves that well-typed stratified programs terminate. Finally, in Section 2.4 we present two additional concurrent λ-calculi: one with dynamic allocation of references and one with dynamic allocation of channels, that are respectively called $\lambda^{\|\mathsf{Ref}}$ and $\lambda^{\|\mathsf{Chan}}$. We show that the abstract language with regions, namely $\lambda^{\|}$, simulates both $\lambda^{\|\mathsf{Ref}}$ and $\lambda^{\|\mathsf{Chan}}$. Since we can lift the stratification of regions to the languages with dynamic values, we are able to show the termination of concurrent programs with references and channels.

A summary of the presented calculi is illustrated in Figure 2.1. For any calculi

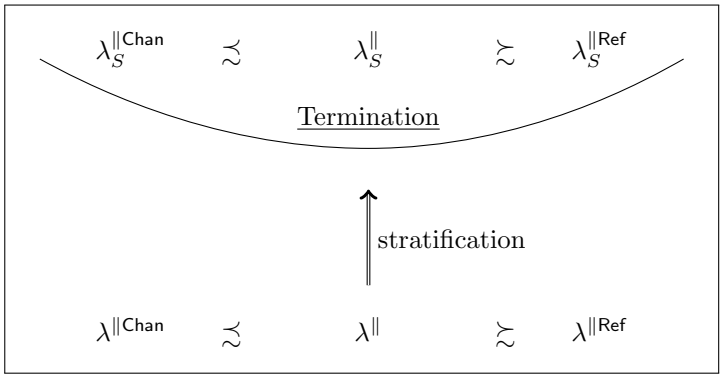


Figure 2.1: Stratification entails termination in every calculi

X, Y, the relation $X \succsim Y$ stands for X *simulates* Y, and X_S stands for the stratified version of X. We see that in any case, stratified or not, the calculus with region $\lambda^{\|}$ simulates both the one with references $\lambda^{\|\mathsf{Ref}}$ and the one with channels $\lambda^{\|\mathsf{Chan}}$. Stratification can be lifted to every calculus and entails termination.

2.1 Syntax and reduction

The concurrent λ-calculus $\lambda^{\|}$ is a call-by-value λ-calculus equipped with regions and parallel composition. We recall that a region is an abstraction of a set of dynamically generated values like references and channels. We regard $\lambda^{\|}$ as an abstract, highly non-deterministic language which, as we will see in Section 2.4, simulates more concrete languages like $\lambda^{\|\mathsf{Ref}}$ and $\lambda^{\|\mathsf{Chan}}$.

2.1.1 Syntax

The syntax of the language is presented in Figure 2.2. We have the usual set

-variables	$x, y, \ldots$		
-regions	$r, r', \ldots$		
-values	V	$::=$	$x \mid r \mid \star \mid \lambda x.M$
-terms	M	$::=$	$V \mid MM$
			$\mathsf{get}(r) \mid \mathsf{set}(r, V)$
			$(M \parallel M)$
-stores	S	$::=$	$r \Leftarrow V \mid (S \parallel S)$
-programs	P	$::=$	$M \mid S \mid (P \parallel P)$

Figure 2.2: Syntax of $\lambda^{\parallel}$

of variables $x, y, \ldots$ and we also have a set of regions $r, r', \ldots$ Values contain integers, variables, regions, the unit value $\star$ and λ-abstractions. Terms are made of values, applications, an operator $\mathsf{get}(r)$ to read a value from region, an operator $\mathsf{set}(r, V)$ to assign a value to a region and the parallel composition $(M \parallel N)$ to evaluate M and N concurrently. A store S is the composition of several assignments $r \Leftarrow V$ in parallel and a program P is the combination of several terms and stores in parallel. Note that stores are global, *i.e.* they always occur in empty contexts.

The set of free variables of M is denoted by $\mathsf{FV}(M)$. The capture-avoiding substitution is written $M[V/x]$ and denotes the term M in which each free occurrence of x has been substituted by V. As usual the sequential composition $M; N$ can be encoded by $(\lambda x.N)M$ where $x \notin \mathsf{FV}(N)$.

2.1.2 Reduction

The call-by-value reduction of $\lambda^{\parallel}$ is given in Figure 2.3 which we comment in the following paragraphs.

Programs are considered up to a structural equivalence $\equiv$ which is the least equivalence relation that contains the equations for α-renaming, commutativity and associativity of parallel composition.

We distinguish two kinds of contexts. An evaluation context E specifies a left-to-right call-by-value evaluation strategy. A static evaluation context C acts as an arbitrary scheduler which chooses a random thread to evaluate.

The structural equivalence $\equiv$ is only preserved by static evaluation contexts. For example we have

$$(P_1 \parallel P_2) \parallel P_3 \equiv (P_2 \parallel P_1) \parallel P_3$$

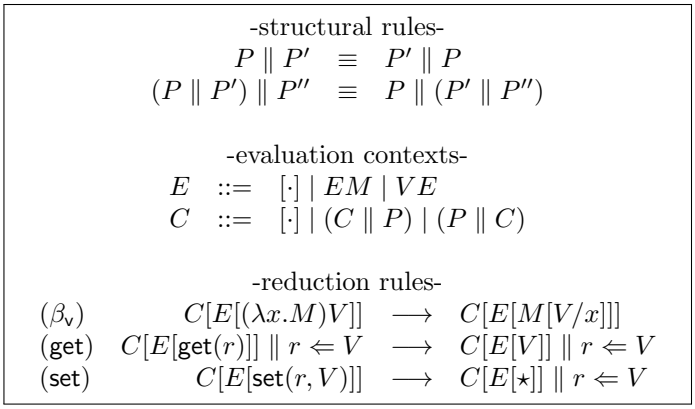


Figure 2.3: Call-by-value reduction of $\lambda^{\parallel}$

where the program $(P_1 \parallel P_2)$ occurs in the static context $C = ([\cdot] \parallel P_3)$ but

$$(\lambda x.M \parallel N)V \not\equiv (\lambda x.N \parallel M)V$$

where $(M \parallel N)$ occurs under a λ-abstraction in the context $E = [\cdot]V$.

The reduction rules apply modulo structural equivalence and each rule is identified by its name: (β_{v}) is the usual β-reduction restrained to values; (get) is for reading *some* value from a region and (set) is for *adding* a value to a region.

We remark that the reduction rule (set) generates an assignment which is new and out of the evaluation contexts; this implies two things:

1. Store assignments are global and shared by every threads.
2. Store assignments are cumulative, that is several values can be assigned to a region. We will see that this allows a single region to abstract an unlimited number of memory locations. In turn, reading a region consists in getting *non-deterministically* one of the assigned values.

The reader may have noticed that the program $\mathsf{set}(r, M)$ is not generated by the syntax if M is not a value. This choice simplifies the shape of evaluation contexts since we do not need to consider the context $\mathsf{set}(r, E)$. On the other hand, this does not cause any loss of expressivity since we consider that $\mathsf{set}(r, M)$ is syntactic sugar for $(\lambda x.\mathsf{set}(r, x))M$.

Example 2.1.1. Here is a programming example. Assume we dispose of integers $\overline{z}$ for $z \in \mathbb{Z}$ and their basic operators. Consider the following function F that takes three arguments g, h and x:

$$F = \lambda g.\lambda h.\lambda x.\mathsf{set}(r, gx) \parallel \mathsf{set}(r, hx) \parallel \mathsf{get}(r) + \mathsf{get}(r)$$

It generates three concurrent threads which respectively do the following: writing the result of the application gx into the region r, writing the result of the

application hx into the region r and reading the region r twice to sum the values. Now consider the following arguments:

$$G = \lambda x.x + \overline{1}$$
$$H = \lambda x.\overline{2}$$

One possible reduction when we apply F to G, H and $\overline{z}$ is:

$$\begin{array}{rcl}
((FG)H)\overline{z} & \longrightarrow^{+} & \mathsf{set}(r, G\overline{z}) \parallel \mathsf{set}(r, H\overline{z}) \parallel \mathsf{get}(r) + \mathsf{get}(r) \\
& \longrightarrow^{+} & \star \parallel \star \parallel \mathsf{get}(r) + \mathsf{get}(r) \parallel r \Leftarrow \overline{z+1} \parallel r \Leftarrow \overline{2} \\
& \longrightarrow & \star \parallel \star \parallel \overline{z+1} + \mathsf{get}(r) \parallel r \Leftarrow \overline{z+1} \parallel r \Leftarrow \overline{2} \\
& \longrightarrow & \star \parallel \star \parallel \overline{z+1} + \overline{2} \parallel r \Leftarrow \overline{z+1} \parallel r \Leftarrow \overline{2} \\
& \longrightarrow & \star \parallel \star \parallel \overline{z+3} \parallel r \Leftarrow \overline{z+1} \parallel r \Leftarrow \overline{2}
\end{array}$$

Other possible reductions lead to structurally equivalent programs, except that we may find the value $\overline{2z+2}$ or $\overline{4}$ instead of $\overline{z+3}$, depending on which value is read from the region r.

2.2 Type and effect system

As we explained in the introduction of this chapter, usual types systems cannot take side effects into account and thus cannot serve to establish the termination of concurrent programs. In this section, we present a *type and effect* system to statically determine the regions on which side effects are produced. Our formalism is the one proposed by R. Amadio [Ama09]. We will see in the next section how this type and effect system can be used to entail the termination of programs.

2.2.1 Types and contexts

The starting point is to consider *effects*, denoted with $e, e', \ldots$ as finite sets of regions. Then, the functional type is annotated with an effect e such that we write

$$A \xrightarrow{e} B$$

for the type of a function that, when given a value of type A, produces side effects on the regions in e and returns a program of type B.

We define the syntax of types and contexts in Figure 2.4. We distinguish two kinds of types:

1. General types are denoted with $\alpha, \alpha', \ldots$ and contain a special *behavior* type $\mathbf{B}$ which is given to stores or concurrent threads which are not supposed to return a value but just to produce side effects.

-effects	$e, e', \ldots$		
-types	α	$::=$	$\mathbf{B} \mid A$
-value types	A	$::=$	$\mathsf{Unit} \mid A \xrightarrow{e} \alpha \mid \mathsf{Reg}_r A$
-variable contexts	Γ	$::=$	$x_1 : A_1, \ldots, x_n : A_n$
-region contexts	R	$::=$	$r_1 : A_1, \ldots, r_n : A_n$

Figure 2.4: Syntax of types, effects and contexts

2. Value types are denoted with $A, B, \ldots$ and are types of entities that may return a value. The only ground type is the unit type Unit. The functional type $A \xrightarrow{e} \alpha$ ensures that functions are only given values as argument but may return a program which evaluates into a value *or* several concurrent threads. The type $\mathsf{Reg}_r A$ is the type of the region r containing values of type A. Hereby types may depend on regions.

We distinguish also two kinds of contexts.

1. Variable contexts are made of distinct variables that are associated to value types, thus we will not be able to build a program of type $\mathbf{B} \xrightarrow{e} \alpha$. We write $dom(\Gamma)$ for the set $\{x_1, \ldots, x_n\}$.

2. Region contexts are made of distinct regions that are associated to value types. The typing system will guarantee that whenever we use a type $\mathsf{Reg}_r A$ the region context contains a hypothesis $r : A$. Thus we will not be able to store non-values in regions. We write $dom(R)$ for the set $\{r_1, \ldots, r_n\}$.

The region type $\mathsf{Reg}_r A$ is carrying an explicit name of region. As we will see in Section 2.4, this dependency between types and regions allows to simulate a calculus with dynamic locations by a calculus with regions. However, we have to be careful in defining the following notions:

- A type is compatible with a region context (judgment $R \downarrow \alpha$).
- A region context is well-formed (judgment $R \vdash$).
- A type is well-formed in a region context (judgment $R \vdash \alpha$), a variable context is well-formed in a region context ($R \vdash \Gamma$), and a type and effect is well-formed in a region context (judgment $R \vdash (\alpha, e)$).

The rules of these judgments are given in Figure 2.5. A more informal way to express these conditions is to say that a judgment $r_1 : A_1, \ldots, r_n : A_n \vdash \alpha$ is well formed provided that:

1. All the region names occurring in the types $A_1, \ldots, A_n, \alpha$ belong to the set $\{r_1, \ldots, r_n\}$,

2. All types of the shape $\mathsf{Reg}_{r_i} B$ with $i \in \{1, \ldots, n\}$ and occurring in the types $A_1, \ldots, A_n, \alpha$ are such that $B = A_i$.

$$\frac{}{R \downarrow \mathsf{Int}} \qquad \frac{}{R \downarrow \mathsf{Unit}} \qquad \frac{}{R \downarrow \mathbf{B}}$$

$$\frac{R \downarrow A \qquad R \downarrow \alpha \qquad e \subseteq dom(R)}{R \downarrow A \xrightarrow{e} \alpha} \qquad \frac{r : A \in R}{R \downarrow \mathsf{Reg}_r A}$$

$$\frac{\forall r : A \in R \qquad R \downarrow A}{R \vdash} \qquad \frac{R \vdash \qquad R \downarrow \alpha}{R \vdash \alpha} \qquad \frac{\forall x : A \in \Gamma \qquad R \vdash A}{R \vdash \Gamma}$$

$$\frac{R \vdash \alpha \qquad e \subseteq dom(R)}{R \vdash (\alpha, e)}$$

Figure 2.5: Formation of types and contexts

Example 2.2.1. The judgment $r : \mathsf{Unit} \xrightarrow{\{r\}} \mathsf{Unit} \vdash$ can be derived while the judgments $r_1 : \mathsf{Reg}_{r_2}\mathsf{Unit} \xrightarrow{\{r_2\}} \mathsf{Unit}, r_2 : \mathsf{Unit} \xrightarrow{\{r_1\}} \mathsf{Unit} \vdash$ and $r : \mathsf{Reg}_r A \vdash$ cannot.

2.2.2 Rules

A typing judgment has the shape

$$R; \Gamma \vdash P : (\alpha, e)$$

It gives the type α to the program P and the effect e is an upper bound on the set of regions that P may read or write. Effects are simply built by exploiting the region names occurring in the syntax; in particular, we can be sure that values and stores produce an empty effect while the terms $\mathsf{get}(r)$ and $\mathsf{set}(r, V)$ produce an effect $\{r\}$. The rules of the type and effect system are spelled out in Figure 2.6.

Remark 2.2.2. Here are some remarks on the rules.

- Contexts are the same in every rule and each axiom (namely var, unit and reg) has a premise $R \vdash \Gamma$ which ensures that, in all branches of the typing derivations, every type is well-formed with respect to the region context.
- Effects are initialized by the rules get and set by referring to the explicit region name of the read/write operator.
- We notice in the rule lam how the effect of a term ends up on the functional arrow in order to build a λ-abstraction with empty effect. Finally, binary rules handle effects in an additive way by set union.
- We distinguish two rules for parallel composition. par_1 indicates that a program P in parallel with a store should have the type of P since

$$\mathsf{var}\,\frac{R \vdash \Gamma \qquad x : A \in \Gamma}{R; \Gamma \vdash x : (A, \emptyset)} \qquad \mathsf{unit}\,\frac{R \vdash \Gamma}{R; \Gamma \vdash \star : (\mathsf{Unit}, \emptyset)}$$

$$\mathsf{reg}\,\frac{R \vdash \Gamma \qquad r : A \in R}{R; \Gamma \vdash r : (\mathsf{Reg}_r A, \emptyset)}$$

$$\mathsf{lam}\,\frac{R; \Gamma, x : A \vdash M : (\alpha, e)}{R : \Gamma \vdash \lambda x.M : (A \xrightarrow{e} \alpha, \emptyset)}$$

$$\mathsf{app}\,\frac{R; \Gamma \vdash M : (A \xrightarrow{e_1} \alpha, e_2) \qquad R; \Gamma \vdash N : (A, e_3)}{R; \Gamma \vdash MN : (\alpha, e_1 \cup e_2 \cup e_3)}$$

$$\mathsf{get}\,\frac{R; \Gamma \vdash r : (\mathsf{Reg}_r A, \emptyset)}{R; \Gamma \vdash \mathsf{get}(r) : (A, \{r\})}$$

$$\mathsf{set}\,\frac{R; \Gamma \vdash r : (\mathsf{Reg}_r A, \emptyset) \qquad R; \Gamma \vdash V : (A, \emptyset)}{R; \Gamma \vdash \mathsf{set}(r, V) : (\mathsf{Unit}, \{r\})}$$

$$\mathsf{store}\,\frac{R; \Gamma \vdash r : (\mathsf{Reg}_r A, \emptyset) \qquad R; \Gamma \vdash V : (A, \emptyset)}{R; \Gamma \vdash r \Leftarrow V : (\mathbf{B}, \emptyset)}$$

$$\mathsf{par}_1\,\frac{R; \Gamma \vdash P : (\alpha, e) \qquad R; \Gamma \vdash S : (\mathbf{B}, \emptyset)}{R; \Gamma \vdash P \parallel S : (\alpha, e)}$$

$$\mathsf{par}_2\,\frac{i = 1, 2 \qquad R; \Gamma \vdash P_i : (\alpha_i, e_1) \qquad P_i \text{ not a store}}{R; \Gamma \vdash P_1 \parallel P_2 : (\mathbf{B}, e_1 \cup e_2)}$$

Figure 2.6: Type and effect rules of $\lambda^{\parallel}$

we might be interested in its result. par_2 indicates that two concurrent threads cannot reduce to a single result. (note that we have omitted the symmetric rules to support commutativity of parallel composition).

We notice that the following rules can be derived for syntactic sugar $\mathsf{set}(r, M)$:

$$\mathsf{set}_M \frac{R;\Gamma \vdash r : (\mathsf{Reg}_r A, \emptyset) \qquad R;\Gamma \vdash M : (A, e)}{R;\Gamma \vdash \mathsf{set}(r, M) : (\mathsf{Unit}, e \cup \{r\})}$$

Example 2.2.3. The program F of Example 2.1.1 written

$$F = \lambda g.\lambda h.\lambda x.\mathsf{set}(r, gx) \parallel \mathsf{set}(r, hx) \parallel \mathsf{get}(r) + \mathsf{get}(r)$$

can be given the following typing judgment:

$$R; - \vdash F : ((A \xrightarrow{\emptyset} \mathsf{Int}) \xrightarrow{\emptyset} (A \xrightarrow{\emptyset} \mathsf{Int}) \xrightarrow{\emptyset} A \xrightarrow{\{r\}} \mathbf{B}, \emptyset)$$

where $r : \mathsf{Int} \in R$. Also, the programs $G = \lambda x.x + \overline{1}$ and $H = \lambda x.\overline{2}$ can be given the type and effect $(\mathsf{Int} \xrightarrow{\emptyset} \mathsf{Int}, \emptyset)$ in order to derive the following judgment:

$$R; - \vdash ((FG)H)\overline{z} : (\mathbf{B}, \{r\})$$

Subtyping

In some cases, the presence of effects may be unnecessarily restrictive. For example, suppose that we first want to write into a region r a value V_1 of type $\mathsf{Unit} \xrightarrow{\emptyset} \mathsf{Unit}$ that may not produce any side effect; then we want to add to r another value V_2 of type $\mathsf{Unit} \xrightarrow{e} \mathsf{Unit}$ where $e \supsetneq \emptyset$. Considering that a region context associates a unique type to the region r, one of the assignments is not typable.

In order to allow for some flexibility, it is convenient to introduce a subtyping relation on types (judgment $R \vdash \alpha \leq \alpha'$) and on types and effects (judgment $R \vdash (\alpha, e) \leq (\alpha', e')$) as specified in Figure 2.7. Intuitively, the new subtyping

$$\frac{}{R \vdash \alpha \leq \alpha} \qquad \frac{e \subseteq e' \subseteq dom(R) \quad R \vdash A' \leq A \quad R \vdash \alpha \leq \alpha'}{R \vdash (A \xrightarrow{e} \alpha) \leq (A' \xrightarrow{e'} \alpha')}$$

$$\frac{e \subseteq e' \subseteq dom(R) \quad R \vdash \alpha \leq \alpha'}{R \vdash (\alpha, e) \leq (\alpha', e')} \qquad \mathsf{sub}\frac{R;\Gamma \vdash M : (\alpha, e) \quad R \vdash (\alpha, e) \leq (\alpha', e')}{R;\Gamma \vdash M : (\alpha', e')}$$

Figure 2.7: Subtyping induced by effect containment

rule sub has the following consequence: the effect e of a functional type $A \xrightarrow{e} \alpha$

or the effect of a program becomes an *upper bound* of the set of regions that may be read/written.

Back to our initial problem, let us define a region context R such that $r : \mathsf{Unit} \xrightarrow{e} \mathsf{Unit} \in R$ where $e \supseteq \emptyset$. By using the subtyping rule, we can give the type and effect $(\mathsf{Unit} \xrightarrow{e} \mathsf{Unit}, \emptyset)$ to the effect free value $V_1 = \lambda x.x$ with the following derivation:

$$\dfrac{\mathsf{sub}\ \dfrac{\dfrac{}{R; x : \mathsf{Unit} \vdash x : (\mathsf{Unit}, \emptyset)}}{R; x : \mathsf{Unit} \vdash x : (\mathsf{Unit}, e)}}{R; - \vdash \lambda x.x : (\mathsf{Unit} \xrightarrow{e} \mathsf{Unit}, \emptyset)}$$

Now take the value V_2 of type and effect $(\mathsf{Unit} \xrightarrow{e} \mathsf{Unit})$. Finally, the program that writes V_1 and V_2 to r can be given the judgment

$$R; - \vdash \mathsf{set}(r, V_1); \mathsf{set}(r, V_2) : (\mathsf{Unit}, \{r\})$$

We notice that the *transitivity rule* for subtyping

$$\dfrac{R \vdash \alpha \leq \alpha' \qquad R \vdash \alpha' \leq \alpha''}{R \vdash \alpha \leq \alpha''}$$

can be derived via a simple induction on the height of the proofs. Moreover, the introduction of the subtyping rules has a limited impact on the structure of the typing proofs. Indeed, if $R \vdash A \leq B$ then we know that A and B may just differ in the effects annotating the functional types. In particular, when looking at the proof of the typing judgment of a value such as $R; \Gamma \vdash \lambda x.M : (A, e)$, we can always argue that A has the shape $A_1 \xrightarrow{e_1} A_2$ and, in case the effect e is not empty, that there is a shorter proof of the judgment $R; \Gamma \vdash \lambda x.M : (B_1 \xrightarrow{e_2} B_2, \emptyset)$ where $R \vdash A_1 \leq B_1$, $R \vdash B_2 \leq A_2$, and $e_2 \subseteq e_1$.

2.2.3 Properties

The usual properties of type systems can be adapted to the type and effect system.

First, it is possible to weaken variable and region contexts, provided that they are well-formed.

Lemma 2.2.4 (Weakening)**.** *If* $R; \Gamma \vdash P : (\alpha, e)$ *and* $R, R' \vdash \Gamma, \Gamma'$ *then* $R, R'; \Gamma, \Gamma' \vdash P : (\alpha, e)$.

Typing is also preserved when we substitute a variable for an effect free value of the same type.

Lemma 2.2.5 (Substitution)**.** *If* $R; \Gamma, x : A \vdash P : (\alpha, e)$ *and* $R; \Gamma \vdash V : (A, \emptyset)$ *then* $R; \Gamma \vdash P[V/x] : (\alpha, e)$.

The above lemmas can be shown by induction on the typing judgments and they are needed to prove subject reduction. Let us write $S_{|f}$ as the store S restricted to the regions in f, for any set of regions f.

Proposition 2.2.6 (Subject reduction)**.** *If* $R_1, R_2; \Gamma \vdash P \parallel S : (\alpha, e)$ *and* $P \parallel S \longrightarrow P' \parallel S'$ *and* $R_1 \vdash (\alpha, e)$ *then:*

1. $R_1, R_2; \Gamma \vdash P' \parallel S' : (\alpha, e)$,
2. $P \parallel S_{|dom(R_1)} \longrightarrow P' \parallel S'_{|dom(R_1)}$ *and* $S'_{|dom(R_2)} = S_{|dom(R_2)}$.

The first statement simply says that the type and effect is preserved by reduction. The second statement guarantees that the program can only read/write regions included in the region context needed to the well-formation of the type and effect. More generally, this shows that that the effect of a program is an upper bound of the set of regions on which side effects are produced. The proof of this statement can be shown by checking that if a program $C[E[M]]$ has an effect e and $M = \mathsf{get}(r)$ or $M = \mathsf{set}(r, V)$, then $r \in e$.

Finally, a *progress property* states that if a program cannot reduce, then every thread is either a value or a term of the shape $E[\mathsf{get}(r)]$ where the region r is empty.

Proposition 2.2.7 (Progress)**.** *Suppose* P *is a typable program which cannot reduce. Then* P *is structurally equivalent to a program*

$$M_1 \parallel \cdots \parallel M_m \parallel S \qquad m \geq 0$$

where M_i *is either a value or can be uniquely decomposed as a term* $E[\mathsf{get}(r)]$ *such that no assignment to* r *exists in the store* S.

The proof is standard and mainly consists in showing that a closed value of type $A \xrightarrow{e} B$ must have the shape $\lambda x.M$, so that a well-typed closed program VN is guarantee to reduce.

2.2.4 Recursion

The current type and effect system allows to write in a region r a function $\lambda x.M$ where M produces an effect on r itself, for instance $\lambda x.\mathsf{get}(r)x$. This kind of circularity may lead to diverging computations like the following one:

$$\begin{array}{rl} & \mathsf{set}(r, \lambda x.\mathsf{get}(r)x); \mathsf{get}(r)\star \\ \longrightarrow^* & \mathsf{get}(r)\star \parallel r \Leftarrow \lambda x.\mathsf{get}(r)x \\ \longrightarrow & (\lambda x.\mathsf{get}(r)x)\star \parallel r \Leftarrow \lambda x.\mathsf{get}(r)x \\ \longrightarrow & \mathsf{get}(r)\star \parallel r \Leftarrow \lambda x.\mathsf{get}(r)x \\ \longrightarrow & \ldots \end{array}$$

This circularity can be used to define an imperative fixpoint combinator, also well-known as *Landin's trick*. More precisely, we define a combinator

$$\mu_r f.\lambda x.M = \mathsf{set}(r, \lambda x.M[\lambda y.\mathsf{get}(r)y/f]); \mathsf{get}(r)\star$$

that relates to a region r and binds f in M. Then the following reduction rule can be derived:

$$(\mu_r)\qquad (\mu_r f.\lambda x.M)V \longrightarrow M[\lambda y.\mathsf{get}(r)y/f, V/x] \parallel r \Leftarrow \lambda x.M[\lambda y.\mathsf{get}(r)y/f]$$

As an example, this combinator can be used to define a counter that keeps being incremented forever and each time writes its value to a region r'. Take

$$\lambda x.M = \lambda x.\mathsf{set}(r', x); f(x + \overline{1})$$

Then we observe

$$\begin{array}{ll} & (\mu_r f.\lambda x.M)\overline{0} \\ \longrightarrow^* & M[\lambda y.\mathsf{get}(r)y/f, \overline{1}/x] \parallel r \Leftarrow \lambda x.M[\lambda y.\mathsf{get}(r)y/f] \parallel r' \Leftarrow \overline{0} \\ \longrightarrow^* & M[\lambda y.\mathsf{get}(r)y/f, \overline{2}/x] \parallel r \Leftarrow \lambda x.M[\lambda y.\mathsf{get}(r)y/f] \parallel r' \Leftarrow \overline{0} \parallel r' \Leftarrow \overline{1} \\ \longrightarrow^* & M[\lambda y.\mathsf{get}(r)y/f, \overline{3}/x] \parallel r \Leftarrow \lambda x.M[\lambda y.\mathsf{get}(r)y/f] \parallel r' \Leftarrow \overline{0} \parallel r' \Leftarrow \overline{1} \parallel r' \Leftarrow \overline{2} \\ \longrightarrow^* & \ldots \end{array}$$

Although the type and effect system takes side effects into account, it does not prevent circularities and the following rule can be derived:

$$\mathsf{fix}\frac{R, r : A \xrightarrow{e\cup\{r\}} \alpha; \Gamma, f : A \xrightarrow{e\cup\{r\}} \alpha \vdash \lambda x.M : (A \xrightarrow{e\cup\{r\}} \alpha, \emptyset)}{R; \Gamma \vdash \mu_r f.\lambda x.M : (A \xrightarrow{e\cup\{r\}} \alpha, \emptyset)}$$

2.3 Termination

G. Boudol introduced [Bou10] the discipline of *stratified* regions in order to recover the termination of concurrent programs. Intuitively, the idea is to fix a well-founded order on regions and make sure that the values stored in a given region may only produce side effects on smaller regions. Back to the type and effect system, stratification means that a value of type $\mathsf{Unit} \xrightarrow{\{r\}} \mathsf{Unit}$ can only be stored in regions which are larger than r. This discipline can be easily ensured by redefining the rules of Figure 2.5 governing the formation of types and contexts .

We give new definitions of the judgments $R \vdash$, $R \vdash \alpha$ and $R \vdash (\alpha, e)$ in Figure 2.8. Regions are thus ordered from left to right, that is if the judgment

$$r_1 : A_1, r_2 : A_2, \ldots, r_i : A_i, \ldots, r_n : A_n \vdash$$

can be derived, effects occurring in A_i may only contain regions r_j where $j < i$.

We denote by '$\vdash_S$' provability in the system where the rules for the formation of unstratified region contexts (Figure 2.5) are replaced by the rules for the formation of stratified region contexts (Figure 2.8). We call $\lambda_S^{\parallel}$ the stratified concurrent λ-calculus. The fixpoint rule fix given in the previous section cannot be derived in $\vdash_S$ since the region context is not stratified (the type of region r has an effect on r). The following theorem can be established.

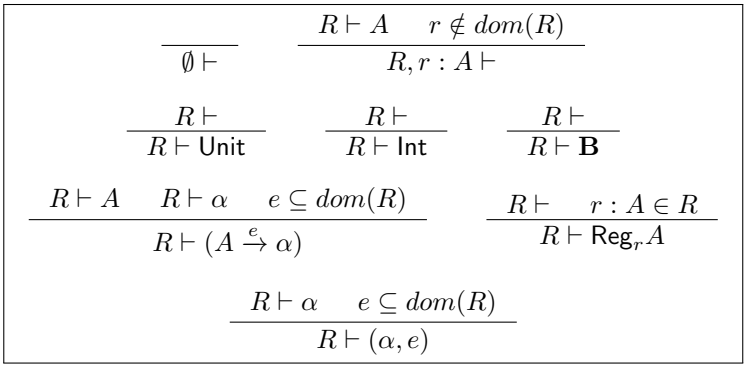


Figure 2.8: Stratified formation of types and contexts

Theorem 2.3.1 (Termination)**.** *If* $R; \Gamma \vdash_S P : (\alpha, e)$ *then* P *terminates.*

The proof is based on an extension of the reducibility candidates method to programs with stores and was originally presented in G. Boudol's paper [Bou10] and later simplified by R. Amadio [Ama09]. The latter version of the proof can be summarized in the following items.

- The starting idea is to define the interpretation $\underline{R \vdash}$ of a stratified region context R as a set of stores and to define the interpretation $\underline{R \vdash (\alpha, e)}$ of a type and effect (α, e) as a set of terms that terminate with respect to stores of $\underline{R \vdash}$. The interpretations are thus mutually defined but well-founded because stratification allows for an inductive definition of the stored values. For instance, the interpretation $\underline{r_1 : A_1, r_2 : A_2 \vdash (\alpha, e)}$ refers to $\underline{r_1 : A_1, r_2 : A_2 \vdash}$ which is a set of stores where all the values of r_2 can only have an effect on r_1 and thus belong to $\underline{r_1 : A_1 \vdash (A_2, e)}$, which in turn refers to the 'smaller' interpretation $\underline{r_1 : A_1 \vdash}$,...

- A key point of the proof is that region contexts are interpreted as 'saturated' stores which already contains all the values that can be possibly written. This gives a simple argument to extend the proof to concurrent programs. Indeed, we know that each thread taken separately terminates with respect to a saturated store. If the parallel composition of a set of threads diverges, then one of the threads must diverge. But since the saturated store cannot be updated by the set of threads, this contradicts the hypothesis that each single thread taken apart terminates.

In Chapter 8, we give another realizability interpretation that takes quantitative information into account so that the length of reductions of an imperative λ-calculus can be bounded.

2.4 Dynamic locations

To conclude this chapter, we introduce a type and effect system for two concurrent λ-calculi with dynamic memory locations instead of regions: one with references that we call $\lambda^{\|\mathsf{Ref}}$, and one with channels that we call $\lambda^{\|\mathsf{Chan}}$. Since $\lambda^{\|\mathsf{Ref}}$ and $\lambda^{\|\mathsf{Chan}}$ still relate to regions at the level of types, we are able to lift the stratification discipline to these calculi. Moreover, $\lambda^{\|}$ simulates $\lambda^{\|\mathsf{Ref}}$ and $\lambda^{\|\mathsf{Chan}}$ by exact correspondence of reduction steps, therefore we can prove the termination of $\lambda^{\|\mathsf{Ref}}$ and $\lambda^{\|\mathsf{Chan}}$.

First, let us present the relation between the systems with dynamic locations and the system with abstract regions informally. In $\lambda^{\|}$, read and write operators refer to region names. On the other hand, the languages $\lambda^{\|\mathsf{Chan}}$ and $\lambda^{\|\mathsf{Ref}}$ introduce terms of the form $\nu x.M$ to generate a new memory location x whose scope is M. We can thus write a program

$$\nu x.(\lambda y.\mathsf{set}(y, V); \mathsf{get}(y))x$$

that generates a fresh location x and gives it as argument to a function that writes and reads at this location. In ML style, this would correspond to the program

$$\mathsf{let}\ x = \mathsf{ref}\ V\ \mathsf{in}\ !x$$

There is a simple typed translation from $\lambda^{\|\mathsf{Chan}}$ and $\lambda^{\|\mathsf{Ref}}$ to $\lambda^{\|}$. For this, a memory location must be a variable that relates to an abstract region r by having the type $\mathsf{Reg}_r A$ for some type A. Then the translation consists in replacing each (free or bound) variable that relates to a region r by the name r. We will see that the program with regions obtained by translation simulates the original because each reduction step in $\lambda^{\|\mathsf{Chan}}$ and $\lambda^{\|\mathsf{Ref}}$ is mapped to exactly one reduction step of $\lambda^{\|}$. Therefore we can apply the termination theorem to $\lambda^{\|\mathsf{Chan}}$ and $\lambda^{\|\mathsf{Ref}}$.

The formalization of $\lambda^{\|\mathsf{Chan}}$ and $\lambda^{\|\mathsf{Ref}}$ is summarized in Figure 2.9. They have the same syntax and typing rules, they only differ in their reduction rules. Concretely:

- The syntax of the languages does not contain region names, instead read and write operators and stores relate to memory locations (that is variables). We also find a ν binder with two associated structural rules (ν_E) and (ν_C) for scope extrusion.
- Each language has its own set of reduction rules (in addition to (β_{v})). Specifically, when we read a reference the value is *copied* from the store and when we write a reference the value *overwrites* the previous one (we assume that the ν binder generates a location with a dummy value that is overwritten at the first write). When we write a channel we *add* the value to the store and when we read a channel we *consume* one of the value in the channel. Thus channels are asynchronous, unbounded and unordered.

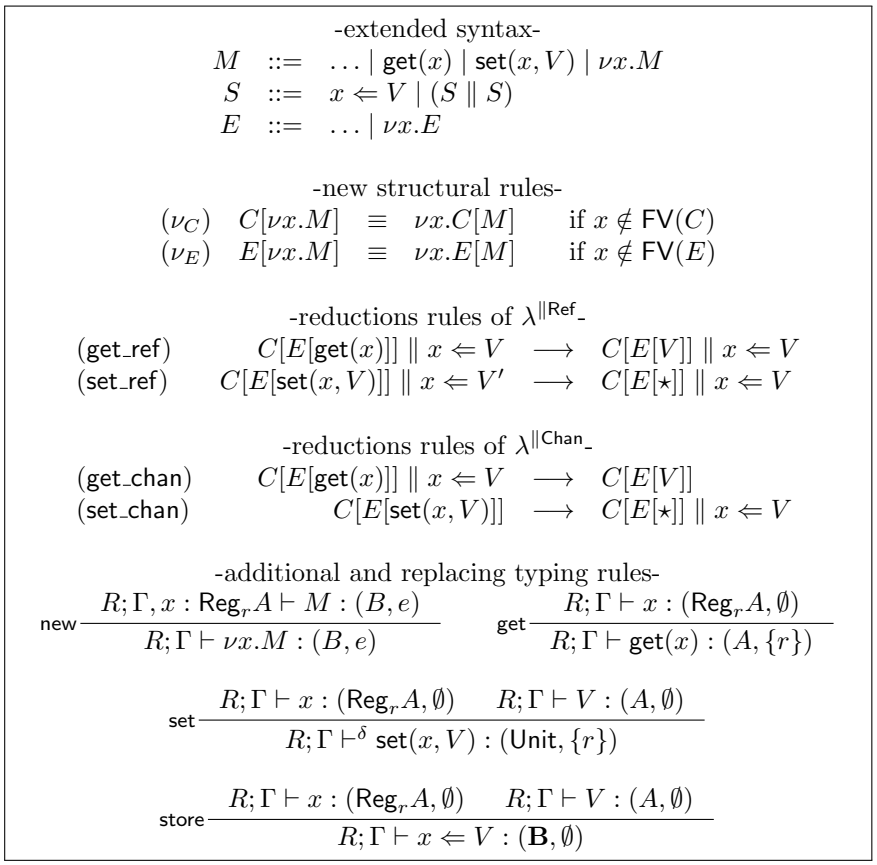


Figure 2.9: Overview of $\lambda^{\parallel \mathsf{Chan}}$ and $\lambda^{\parallel \mathsf{Ref}}$

- The typing rule reg is no longer necessary but we introduce a rule called new that allows to bind variables of region type. The rules get, set and store replace the previous ones and the effect of programs can still be inferred by referring to the region names occurring in the types of memory locations.

Clearly, subject reduction (Proposition 2.2.6) and progress (Proposition 2.2.7) can be lifted to $\lambda^{\parallel \mathsf{Chan}}$ and $\lambda^{\parallel \mathsf{Ref}}$.

Concerning termination, the proof goes by a translation phase from $\lambda^{\parallel \mathsf{Chan}}$ or $\lambda^{\parallel \mathsf{Ref}}$ to $\lambda^{\parallel}$. Without loss of generality, we illustrate this on a concrete program-

ming example of $\lambda^{\|\mathsf{Ref}}$. Consider the following reduction:

$$\begin{array}{ll} & \nu x.\nu y.(\lambda z.\mathsf{set}(z,\overline{n}) \parallel \mathsf{set}(y,\mathsf{get}(z)))x \\ \longrightarrow & \nu x.\nu y.\mathsf{set}(x,\overline{n}) \parallel \mathsf{set}(y,\mathsf{get}(x)) \\ \longrightarrow & \nu x.\nu y.\star \parallel \mathsf{set}(y,\mathsf{get}(x)) \parallel x \Leftarrow \overline{n} \\ \longrightarrow & \nu x.\nu y.\star \parallel \mathsf{set}(y,\overline{n}) \parallel x \Leftarrow \overline{n} \\ \longrightarrow & \nu x.\nu y.\star \parallel \star \parallel x \Leftarrow \overline{n} \parallel y \Leftarrow \overline{n} \end{array}$$

The location x is passed to a function which generates two threads. The first thread writes an integer $\overline{n}$ in reference x, the second thread read $\overline{n}$ from reference x and propagates it into reference y. The program may be given the following type and effect derivation by taking that references x and y relate to a single region r:

$$\dfrac{\dfrac{\vdots}{r : \mathsf{Int}; x : \mathsf{Reg}_r\mathsf{Int}, y : \mathsf{Reg}_r\mathsf{Int} \vdash (\lambda z.\mathsf{set}(z,\overline{n}) \parallel \mathsf{set}(y,\mathsf{get}(z)))x : (\mathbf{B},\{r\})}}{r : \mathsf{Int}; - \vdash \nu x.\nu y.(\lambda z.\mathsf{set}(z,\overline{n}) \parallel \mathsf{set}(y,\mathsf{get}(z)))x : (\mathbf{B},\{r\})}$$

The translation consists in (1) erasing all ν binders and (2) replacing each variable of region type with the corresponding region name. This gives us the following program whose reductions steps are in one-to-one correspondence with the original reductions:

$$\begin{array}{ll} & (\lambda z.\mathsf{set}(r,\overline{n}) \parallel \mathsf{set}(r,\mathsf{get}(r)))r \\ \longrightarrow & \mathsf{set}(r,\overline{n}) \parallel \mathsf{set}(r,\mathsf{get}(r)) \\ \longrightarrow & \star \parallel \mathsf{set}(r,\mathsf{get}(r)) \parallel r \Leftarrow \overline{n} \\ \longrightarrow & \star \parallel \mathsf{set}(r,\overline{n}) \parallel r \Leftarrow \overline{n} \\ \longrightarrow & \star \parallel \star \parallel r \Leftarrow \overline{n} \parallel r \Leftarrow \overline{n} \end{array}$$

We observe that the region r 'accumulates' two values $\overline{n}$ because r relates to two references x and y. In general, the $\lambda^{\|}$ program may produce additional assignments since stored values are never erased, but at least one reduction sequence will simulate the original one. The translated program can be given the following type and effect

$$r : \mathsf{Int}; - \vdash (\lambda z.\mathsf{set}(r,\overline{n}) \parallel \mathsf{set}(r,\mathsf{get}(r)))r : (\mathbf{B},\{r\})$$

which is the same as the original program except that the free memory locations x and y do not occur anymore in the variable contexts of the derivation. Since the stratification of regions is preserved by translation we can conclude that the stratified systems $\lambda_S^{\|\mathsf{Ref}}$ and $\lambda_S^{\|\mathsf{Chan}}$ terminate. Indeed, if there is an infinite reduction in $\lambda_S^{\|\mathsf{Ref}}$ or $\lambda_S^{\|\mathsf{Chan}}$, there must be an infinite reduction in $\lambda^{\|}$ and this contradicts Theorem 2.3.1.

It should be noted that using a single region r to abstract every locations is a rather drastic solution. In our example, we could have alternatively associated a distinct region to each channel x and y. In the context of region-based memory

management [TT97], the problem of *region inference* which consists in finding an assignment from locations to regions in the most optimal way with respect to the performance of the language is a crucial question which is beyond the scope of this thesis. The reader may consult a report on the MLKit compiler [TBE+06] that implements a region-based garbage collector; a retrospective on region-based memory management is also available [TBEH04].

Chapter 3

An affine-intuitionistic concurrent λ-calculus

In this chapter, we propose to refine the concurrent λ-calculus $\lambda^{\|}$ so that we can distinguish between data that can be used at most once and data that can be duplicated. This is our starting point to study the time complexity of concurrent programs in the later parts of this thesis. More concretely, our refinement of $\lambda^{\|}$ relies on the following distinction:

- Ordinary values (variables, λ-abstractions) should not be used more than once and are called *affine*.
- A new constructor named *bang* and written '!' is introduced to mark values as duplicable. These duplicable values are called *intuitionistic*.

We denote by $\lambda^{!\|}$ the affine-intuitionistic concurrent λ-calculus.

This formalization does not come from nowhere and can be traced back to J-Y. Girard's discovery of *Linear Logic* [Gir87] where the bang modality '!' is introduced at the level of proofs to control the multiplicity of formulae. Through the proof-as-program correspondence, the bang constructor can be used to control the multiplicity of data in functional programs, and there have been several attempts [BBdPH93, Ben94, Plo93, MOTW99, Bar96][1] at producing a functional programming language based on these ideas and with a reasonably handy syntax. To be precise, in these languages there are *linear* values which must be used *exactly* once. For our purpose, it is sufficient and more general to consider *affine* values which can be used *at most* once.

The major contribution of this chapter is to design an *affine-intuitionistic* type system for $\lambda^{!\|}$ where regions and the bang modality interact and so that types

[1] We recommend P. Wadler's introduction on Linear Logic and the proof-as-program correspondence [Wad93].

are preserved by reduction. There have been previous work on mixing regions with linear types [WW01, FMA06] to deal with the problem of heap-memory deallocation but the approaches were quite different and the relationship with Linear Logic not very clear. A key point in the soundness of this system is that the primitive read operation of the language is refined so that it is not source of duplication. Moreover, we analyze when we can safely duplicate a value containing side effects.

It turns out that the distinction between affine and intuitionistic values allows us to develop another contribution which is a proper discipline of *region usage* that is used to ensure the confluence of programs. This discipline relies on the following point:

- We distinguish between regions that can be read at most once and written at most once from those that can be read and written arbitrarily many times.

Moreover, by decorating the affine-intuitionistic type system with effects, we can smoothly combine the notion of region usage and region stratification so that confluence *and* termination are guaranteed.

Outline The chapter is organized as follows. In Section 3.1, we present an affine-intuitionistic λ-calculus in order to familiarize ourselves with the bang modality in the functional case. We also show how it relates to the simply typed λ-calculus. In Section 3.2 we present the syntax and reduction of an affine-intuitionistic concurrent λ-calculus, called $\lambda^{!\|}$. We introduce its type system and the notion of region usage in Section 3.3 and show that it enjoys the subject reduction property. In Section 3.4, we show how we can tame region usages so that confluence is ensured. Finally, in Section 3.5 we extend the affine-intuitionistic type system with effects so that stratification can be combined with region usages to ensure confluence *and* termination.

The diagram given in Figure 3.1 illustrates the developments of the chapter. For any calculus X, X_S represent the stratified version and X_C represents the version with restricted region usages.

3.1 An affine-intuitionistic λ-calculus

In the simply typed λ-calculus, λ-abstraction has the power of duplication and erasure which is given by the ability to bind arbitrarily many occurrences of a variable. This happens as follows:

$$(\lambda x.x(xy))M \longrightarrow M(My) \tag{3.1}$$
$$(\lambda z.x(xy))M \longrightarrow x(xy) \tag{3.2}$$

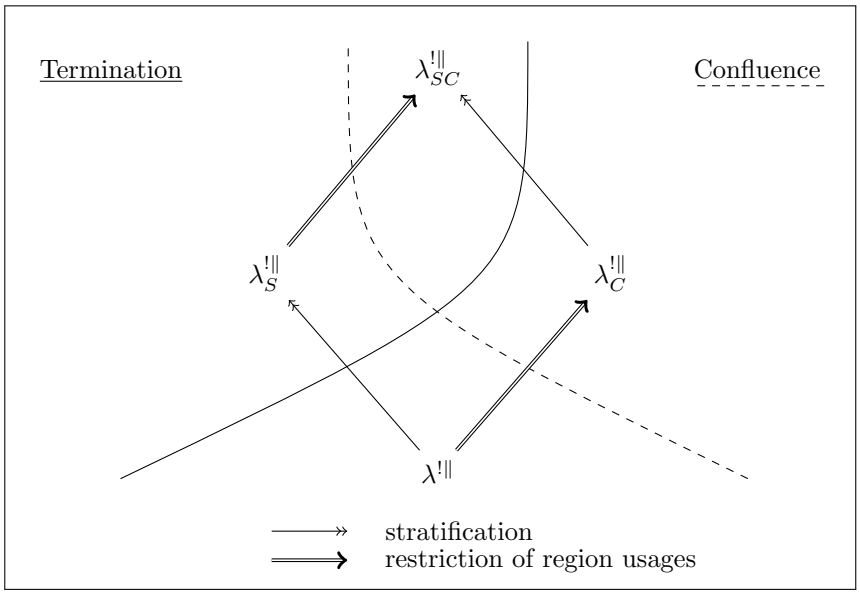


Figure 3.1: Combination of stratification and region usages

In reduction (3.1) the argument M is duplicated by a λ-abstraction that binds two occurrences of its parameter and that may be given the typing judgment

$$y : A \vdash (\lambda x.x(xy)) : (A \to A) \to A \tag{3.3}$$

In reduction (3.2) the argument M is erased by a λ-abstraction that binds zero occurrences of its parameter and that may be given the same type.

In this section, we introduce a typed affine-intuitionistic λ-calculus named $\lambda^!$. We will see that it renders the process of duplication explicit.

3.1.1 Syntax and reduction

The syntax of $\lambda^!$ is defined in Figure 3.2. As in the standard λ-calculus, we find

$$\boxed{M ::= x \mid \lambda x.M \mid MM \mid !M \mid \mathsf{let}\ !x = M\ \mathsf{in}\ M}$$

Figure 3.2: Syntax of $\lambda^!$

variables, λ-abstractions and applications. The new objects are:

- !-terms (or modal terms) written $!M$ that will be duplicable,
- let !-expressions that will be used to bind more than one occurrence of a variable. An expression $\mathsf{let}\ !x = N\ \mathsf{in}\ M$ binds x in M.

The reduction rules are given in Figure 3.3. An evaluation context E is any

$$\begin{array}{rcl} E & ::= & [\cdot] \mid \lambda x.E \mid EM \mid ME \mid !E \\ & & \mathsf{let}\ !x = E\ \mathsf{in}\ M \mid \mathsf{let}\ !x = M\ \mathsf{in}\ E \end{array}$$

$$\begin{array}{rrcl} (\beta) & E[(\lambda x.M)N] & \longrightarrow & E[M[N/x]] \\ (!) & E[\mathsf{let}\ !x = !N\ \mathsf{in}\ M] & \longrightarrow & E[M[N/x]] \end{array}$$

Figure 3.3: Reduction of $\lambda^!$

term with a hole $[\cdot]$. Apart from the usual (β) rule, the reduction comes with an additional rule (!) that eliminates the bang constructor of modal terms.

3.1.2 Typing

Up to now, the process of duplication is not yet explicit since λ-abstractions can still bind many occurrences of variables. We thus introduce an affine-intuitionistic type system that distinguishes between affine (non duplicable) and intuitionistic (duplicable) terms, and ensures that (!) is the only duplicating reduction rule.

The syntax of types and typing context is given in Figure 3.4. Unit is the unit

$$\begin{array}{rcl} A & ::= & \mathsf{Unit} \mid A \multimap A \mid !A \\ \Gamma & ::= & x_1 : (u_1, A_1), \ldots, x_n : (u_n, A_n) \end{array}$$

Figure 3.4: Types and contexts

ground type. $A \multimap A$ is the type of functions that use their argument once and $!A$ is the type of terms that can be duplicated. In a context, each variable is associated with a type and an usage $u \in \{\lambda, !\}$ which specifies if the variable can be bound by a λ-abstraction (usage λ) or a $\mathsf{let}\,!$-expression (usage !). Therefore a variable x such that $x : (u, A) \in \Gamma$ ranges over terms of type A and can be bound according to u. In the following we write Γ_u for $x_1 : (u, A_1), \ldots, x_n : (u, A_n)$.

A typing judgment takes the shape

$$\Gamma \vdash M : A$$

and ensures that, if x occurs free in M and:

- if $x : (\lambda, B) \in \Gamma$ then x *does not* occur inside a modal subterm of M,
- if $x : (!, B) \in \Gamma$ and then x *may* occur inside a modal subterm of M.

The typing rules are introduced in Figure 3.5. The following remarks should give intuitions on the rules.

$$\mathsf{var}\,\frac{x : (u, A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\mathsf{lam}\,\frac{\mathsf{FO}(x, M) \leq 1 \qquad \Gamma, x : (\lambda, A) \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B}$$

$$\mathsf{app}\,\frac{\Gamma \vdash M : A \multimap B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\mathsf{prom}\,\frac{\Gamma_! \vdash M : A}{\Gamma_!, \Delta_\lambda \vdash\, !M : !A} \qquad \mathsf{elim}\,\frac{\Gamma \vdash M : !A \qquad \Gamma, x : (!, A) \vdash N : B}{\Gamma \vdash \mathsf{let}\ !x = M\ \mathsf{in}\ N : B}$$

Figure 3.5: Typing rules of $\lambda^!_{\mathbf{AI}}$

- In order to mark a term as duplicable, the rule prom (for promotion) requires that all the free variables of the term have the usage '!'. Notice that a context Δ_λ can be weakened in order to get the general weakening property.
- The rule lam, where $\mathsf{FO}(x, M)$ stands for the number of free occurrences of x in M, ensures that a λ-abstraction can bind at most once occurrence of a variable. Whereas the rule elim allows a let !-expression to bind arbitrarily many occurrences of a variable.

Remark 3.1.1. In affine-intuitionistic type systems there is usually an explicit 'contraction' rule which only applies to intuitionistic hypotheses. In our system, contexts are handled in an additive way but we use a predicate $\mathsf{FO}(x, M) \leq 1$ in the rule lam to control the number of bound occurrences. This presentation which is inspired by Terui [Ter07] simplify a bit the type system.

3.1.3 Translation from the simply typed λ-calculus

The explicit rendering of duplication in $\lambda^!$ can be better explained by giving a natural translation $\underline{\cdot}$ from the simply typed λ-calculus to $\lambda^!$. The translation, which is also known as Girard's translation [Gir87], is defined by induction on the typing of standard λ-terms:

$$\begin{aligned}
\underline{\mathsf{Unit}} &= \mathsf{Unit} \\
\underline{A \to B} &= !\underline{A} \multimap \underline{B} \\
\underline{x_1 : A_1, \ldots, x_n : A_n} &= x_1 : (!, A_1), \ldots, x_n : (!, A_n)
\end{aligned}$$

$$\begin{aligned}
\underline{x} &= x \\
\underline{\lambda x.M} &= \lambda x.\mathsf{let}\ !x = x\ \mathsf{in}\ \underline{M} \\
\underline{MN} &= \underline{M}!\underline{N}
\end{aligned}$$

Intuitively, the translation expresses the fact that in the simply typed λ-calculus, every argument of an application is potentially duplicated. Thus, the translation of a λ-abstraction internalizes a $\mathsf{let}\,!$-expression that can bind arbitrarily many occurrences of a variable and effectively allows duplication. Also, each argument of an application has to be marked with a bang such as being duplicable. This is exemplified with the reduction (3.1) that translates to the following reduction:

$$\begin{array}{rcl} \underline{(\lambda x.x(xy))M} & = & (\lambda x.\mathsf{let}\,!x = x\ \mathsf{in}\ x(!(x!y)))!\underline{M} \\ & \longrightarrow & \mathsf{let}\,!x = !\underline{M}\ \mathsf{in}\ x(!(x!y)) \\ & \longrightarrow & M(!M(!y)) \end{array}$$

Therefore, the single duplicating (β) step of reduction (3.1) is decomposed into a linear (β) step followed by a duplicating (!) step. And the judgment (3.3) translates to

$$y : (!, A) \vdash \lambda x.\mathsf{let}\,!x = x\ \mathsf{in}\ x(!(x!y)) : !(!\underline{A} \multimap \underline{A}) \multimap \underline{A}$$

In fact, the following simulation property can be shown.

Proposition 3.1.2 (Simulation)**.** *Assume $\vdash_{ST}$ and $\longrightarrow_{ST}$ respectively denote provability and the call-by-name reduction in the simply typed λ-calculus. If $\Gamma \vdash_{ST} M : A$ and $M \longrightarrow_{ST} M'$ then $\underline{\Gamma} \vdash \underline{M} : \underline{A}$ and $\underline{M} \longrightarrow \underline{M'}$.*

Remark 3.1.3. The proposed translation, also known as 'Girard's translation' or 'call-by-name translation' is not optimal in the sense that it collapse all terms of the simply typed λ-calculus to duplicable terms of the affine-intuitionistic λ-calculus, whether they are effectively duplicated or not. The problem of finding a translation that only put bangs where necessary have been studied by Danos *et al.* at the level of proofs [DJS95].

3.2 An affine-intuitionistic concurrent λ-calculus

We now present a concurrent version of the affine-intuitionistic λ-calculus that we call $\lambda^{!\|}$.

A crucial preliminary observation is that, as we have seen that the (β) reduction is potentially source of duplication (see Equation 3.1), side effects may also generate duplication. In particular, the reading rule

$$(\mathsf{get}) \qquad C[E[\mathsf{get}(r)]] \parallel r \Leftarrow V \longrightarrow C[E[V]] \parallel r \Leftarrow V \tag{3.4}$$

taken from the reduction of $\lambda^{\|}$ given in Figure 2.3, *copies* the value V from the store (this is also what happens when a reference cell is read). Therefore, in the hope of designing a sound affine-intuitionistic type system, we are going to define a reduction where values are *consumed* as follows:

$$C[E[\mathsf{get}(r)]] \parallel r \Leftarrow V \longrightarrow C[E[V]]$$

In fact, we will see in Section 3.3.4 that we can still recover the copying mechanism of reduction (3.4).

3.2.1 Syntax

The syntax of $\lambda^{!\|}$ is presented in Figure 3.6. It simply is the combination of

-variables	$x, y, \ldots$		
-regions	$r, r', \ldots$		
-values	V	$::=$	$x \mid r \mid \star \mid \lambda x.M \mid\ !V$
-terms	M	$::=$	$V \mid MM$
			$!M \mid \mathsf{let}\ !x = V\ \mathsf{in}\ M$
			$\mathsf{get}(r) \mid \mathsf{set}(r, V)$
			$(M \parallel M)$
-stores	S	$::=$	$r \Leftarrow V \mid (S \parallel S)$
-programs	P	$::=$	$M \mid S \mid (P \parallel P)$

Figure 3.6: Syntax of $\lambda^{!\|}$

constructs from $\lambda^{\|}$ (Figure 2.2) and $\lambda^{!}$ (Figure 3.2). Notice that $!M$ is only considered a value if M is a value. The notations $\mathsf{FV}(M)$ and $M[V/x]$ extend in a straightforward way and the sequential composition $M; N$ is encoded as usual (see Section 2.1.1).

3.2.2 Reduction

The reduction of $\lambda^{!\|}$ is given in Figure 3.7. It is built from the reduction of $\lambda^{!}$

-structural rules-

$$P \parallel P' \equiv P' \parallel P$$
$$(P \parallel P') \parallel P'' \equiv P \parallel (P' \parallel P'')$$

-evaluation contexts-

$$E ::= [\cdot] \mid EM \mid VE \mid !E$$
$$C ::= [\cdot] \mid (C \parallel P) \mid (P \parallel C)$$

-reduction rules-

$$(\beta_{\mathsf{v}}) \quad C[E[(\lambda x.M)V]] \longrightarrow C[E[M[V/x]]]$$
$$(!_{\mathsf{v}}) \quad C[E[\mathsf{let}\ !x = !V\ \mathsf{in}\ M]] \longrightarrow C[E[M[V/x]]]$$
$$(\mathsf{get}) \quad C[E[\mathsf{get}(r)]] \parallel r \Leftarrow V \longrightarrow C[E[V]] \parallel r \Leftarrow V$$
$$(\mathsf{set}) \quad C[E[\mathsf{set}(r, V)]] \longrightarrow C[E[\star]] \parallel r \Leftarrow V$$

Figure 3.7: Call-by-value reduction of $\lambda^{!\|}$

(Figure 3.3) and $\lambda^{\|}$ (Figure 2.3) so that programs follow a left-to-right call-by-value evaluation strategy. More precisely:

- Arguments are evaluated before substitution in the rules (β_v) and $(!_v)$.
- Evaluation contexts do not allow reduction steps to take place under binders. However, it is possible to evaluate under bangs with the context $!E$.
- As announced in preamble of this section, the rule (get) consumes the value from the store.

Remark 3.2.1. We also define a new syntactic abbreviation (in addition to $\mathsf{set}(r, M)$). Since $\mathsf{let}\ !x = N\ \mathsf{in}\ M$ is not generated by the syntax, we consider it as syntactic sugar for $(\lambda x.\mathsf{let}\ !x = x\ \mathsf{in}\ M)N$. Again, this choice simplifies the shape of evaluation contexts because we do not need to consider the context $\mathsf{let}\ !x = E\ \mathsf{in}\ M$.

3.3 An affine-intuitionistic type system

We now present the affine-intuitionistic type system of $\lambda^{!\|}$ which renders the process of data duplication explicit explicit even in the presence of side effects. In addition, the system includes a discipline of region usages that distinguish regions that are read once and written once from other ones. In the next section, we will see that region usages can be tamed to ensure the confluence of programs.

3.3.1 Types, contexts and usages

We define the syntax of types and contexts in Figure 3.8. It is built from types

-types	α	::=	$\mathbf{B} \mid A$
-value types	A	::=	$\mathsf{Unit} \mid A \multimap \alpha \mid\ !A \mid \mathsf{Reg}_r A$
-variable contexts	Γ	::=	$x_1 : (u_1, A_1), \ldots, x_n : (u_n, A_n)$
-region contexts	R	::=	$r_1 : (U_1, A_1), \ldots, r_n : (U_n, A_n)$

Figure 3.8: Syntax of types and contexts of $\lambda^{!\|}$

and contexts of $\lambda^{\|}$ (Figure 2.4) and $\lambda^{!}$ (Figure 3.4) except that we do not use effects.

As seen previously, a variable usage $u \in \{\lambda, !\}$ specifies if the variable can be bound by a λ-abstraction or a $\mathsf{let}\,!$-expression. The novelty lies in the introduction of *region usages* in region contexts. A *region usage* U is a pair that counts reads and writes on a given region. Concretely, U denotes an element of one of the following three sets:

$$U \in \{[\infty, \infty]\} \cup \{[1, \infty], [0, \infty]\} \cup \{[0, 0], [1, 0], [0, 1], [1, 1]\}$$

By convention we reserve the left component to describe the write usage and the right component to describe the read usage. Therefore, a region with usage $[1, \infty]$ should be written at most once but can be read many times, while a region with usage $[0, 1]$ cannot be written but can be read at most once.

Distinguishing three sets of region usages is necessary to definite a sufficient condition that ensures the confluence of programs. The reason will become clear in the next section. For now, the intuition is that if we can write at most one value into a region, reading the region should be deterministic.

Writing $r : (U, A) \in R$ thus means that region r contains values of type A and can be used according to the usage U. We would like to be able to combine the region usages that are used at different points of a program. For example, if a thread M_1 reads a region r that can be read at most once, we should not be able to put M_1 in parallel with another thread M_2 that also reads r. To this end, we first define a partial binary operation $\uplus : \{0, 1, \infty\} \times \{0, 1, \infty\} \to \{0, 1, \infty\}$ such that

$$x \uplus 0 = x$$
$$0 \uplus x = x$$
$$\infty \uplus \infty = \infty$$

and which is undefined otherwise. Then, the addition $U_1 \uplus U_2$ is defined provided that:

1. the component-wise addition is defined,
2. U_1 and U_2 are in the same set of usages.

The reason for the second condition will become clear in the next section.

Example 3.3.1. $[\infty, \infty] \uplus [0, \infty]$ and $[1, 0] \uplus [1, 0]$ are undefined while $[1, 0] \uplus [0, 1] = [1, 1]$.

Notice that in each set of usages, there is a 'neutral' usage U_0 such that $U_0 \uplus U = U$ for all U in the same set.

The sum on region usages is extended to region contexts as follows:

1. $R_1 \uplus R_2$ is defined provided that $dom(R_1) = dom(R_2)$,
2. if $r : (U_1, A) \in R_1$ and $r : (U_2, A) \in R_2$ then $r : (U_1 \uplus U_2, A) \in R_1 \uplus R_2$ only if $U_1 \uplus U_2$ is defined.

There is no loss of generality in the first condition because if, say $r : (U, A) \in R_1$ and $r \notin dom(R_2)$ then we can always add $r : (U_0, A)$ to R_2 where U_0 is the neutral usage of the set which contains U.

Example 3.3.2.

$$\begin{aligned} & r_1 : ([1,\infty], A), r_2 : ([0,1], B) \\ \uplus\ & r_1 : ([0,\infty], A), r_2 : ([1,0], B) \\ = & \ r_1 : ([1,\infty], A), r_2 : ([1,1], B) \end{aligned}$$

We recall that types depend on region names and, as we did for $\lambda^{\|}$ (Figure 2.5), we state in Figure 3.9 when a type is well-formed in a region context. The

$$\frac{}{R \downarrow \mathsf{Int}} \qquad \frac{}{R \downarrow \mathsf{Unit}} \qquad \frac{}{R \downarrow \mathbf{B}}$$

$$\frac{R \downarrow A}{R \downarrow\ !A} \qquad \frac{R \downarrow A \quad R \downarrow \alpha}{R \downarrow A \multimap \alpha} \qquad \frac{r : (U, A) \in R}{R \downarrow \mathsf{Reg}_r A}$$

$$\frac{\forall r : (U, A) \in R \quad R \downarrow A}{R \vdash} \qquad \frac{R \vdash \quad R \downarrow \alpha}{R \vdash \alpha} \qquad \frac{\forall x : (u, A) \in \Gamma \quad R \vdash A}{R \vdash \Gamma}$$

Figure 3.9: Formation of types and contexts (unstratified)

difference is that we do not consider effects (they will be added later on to recover the termination property), and that we consider the affine-intuitionistic types $!A$ and $A \multimap \alpha$ into account.

3.3.2 Rules

We now introduce the typing rules in Figure 3.10. The rules are built from the type systems of $\lambda^{\|}$ (Figure 2.6) and $\lambda^{!}$ (Figure 3.5) and moreover they handle region usages. The following paragraphs give the required explanations.

First, observe that the reading rule get (respectively the writing rules set and store) requires that the read usage (resp. the write usage) of the region is not null. Then, to control the duplication of region usages, binary rules can only be applied if the sum $R_1 \uplus R_2$ is defined. In a typing derivation, the region contexts thus only differ on region usages.

The formulation of the *promotion* rule (prom) that introduces the bang constructor is crucial and needs to be explained. Intuitively:

- A term whose typing depends on a region which can be read or written at most once should not be duplicated.
- A term which reads a region should only be duplicated if the content of the region is duplicable.

$$\mathsf{var}\frac{R \vdash \Gamma \qquad x : (u, A) \in \Gamma}{R; \Gamma \vdash x : A} \qquad \mathsf{unit}\frac{R \vdash \Gamma}{R; \Gamma \vdash \star : \mathsf{Unit}}$$

$$\mathsf{reg}\frac{R \vdash \Gamma \qquad r : (U, A) \in R}{R; \Gamma \vdash r : \mathsf{Reg}_r A}$$

$$\mathsf{lam}\frac{\mathsf{FO}(x, M) \leq 1 \qquad R; \Gamma, x : (\lambda, A) \vdash M : \alpha}{R; \Gamma \vdash \lambda x.M : A \multimap \alpha}$$

$$\mathsf{app}\frac{R_1; \Gamma \vdash M : A \multimap \alpha \qquad R_2; \Gamma \vdash N : A}{R_1 \uplus R_2; \Gamma \vdash MN : \alpha}$$

$$\mathsf{prom}\frac{\mathit{naff}(R) \qquad R; \Gamma_! \vdash M : A}{R, R'; \Gamma_!, \Delta_\lambda \vdash !M : !A}$$

$$\mathsf{elim}\frac{R_1; \Gamma \vdash V : !A \qquad R_2; \Gamma, x : (!, A) \vdash N : \alpha}{R_1 \uplus R_2; \Gamma \vdash \mathsf{let}\ !x = V\ \mathsf{in}\ N : \alpha}$$

$$\mathsf{get}\frac{r : ([x, y], A) \in R \qquad y \neq 0 \qquad R \vdash \Gamma}{R; \Gamma \vdash \mathsf{get}(r) : A}$$

$$\mathsf{set}\frac{R = r : ([x, y], A) \uplus R' \qquad x \neq 0 \qquad R'; \Gamma \vdash V : A}{R; \Gamma \vdash \mathsf{set}(r, V) : \mathsf{Unit}}$$

$$\mathsf{store}\frac{R = r : ([x, y], A) \uplus R' \qquad x \neq 0 \qquad R'; \Gamma \vdash V : A}{R; \Gamma \vdash r \Leftarrow V : \mathbf{B}}$$

$$\mathsf{par}_1\frac{R_1; \Gamma \vdash P : \alpha \qquad R_2; \Gamma \vdash S : \mathbf{B}}{R_1 \uplus R_2; \Gamma \vdash P \parallel S : \alpha}$$

$$\mathsf{par}_2\frac{i = 1, 2 \qquad R_i; \Gamma \vdash P_i : \alpha_i}{R_1 \uplus R_2; \Gamma \vdash P_1 \parallel P_2 : \mathbf{B}}$$

Figure 3.10: Typing rules of $\lambda^{!\parallel}$

In order to take these two recommendations into account, we define a predicate *naff*(R) (read R *is not affine*) which is true when these two conditions are verified:

1. The regions usages occurring in R are only built from components of the set $\{0, \infty\}$.
2. If $r : ([x, y], A) \in R$ and $y \neq 0$ then A is of the shape $!B$.

Then, it suffices to add the premise *naff*(R) to the promotion rule so that the region usages can be safely duplicated. Indeed, if *naff*(R) is true then $R \uplus R = R$. To conclude on the promotion rule, note that we can add an arbitrary region context R' so that we get a general weakening property on contexts.

As in $\lambda^{\|}$, we can derive typing rules for syntactic sugar expressions; in particular, we have:

$$\mathsf{elim}_M \frac{R_1; \Gamma \vdash M : !A \qquad R_2; \Gamma, x : (!, A) \vdash N : \alpha}{R_1 \uplus R_2; \Gamma \vdash \mathsf{let}\ !x = M\ \mathsf{in}\ N : \alpha}$$

We give a typing example that illustrates why affine region usages (*i.e.* whose components may contain '1') would not be compatible with a purely intuitionistic type system as the one of $\lambda^{\|}$.

Example 3.3.3. Take the derivable typing judgment

$$r : ([0, 0], \mathsf{Int}); - \vdash \lambda f_1.\lambda f_2.(f_1\star \parallel f_2\star) : (\mathsf{Unit} \multimap \mathsf{Int}) \multimap (\mathsf{Int} \multimap \mathsf{Unit}) \multimap \mathbf{B}$$

of a function that takes two other functions f_1 and f_2 as arguments and call them in parallel. Note that the usages of region r are null. Now suppose we want to pass the values $V_1 = \lambda x.\mathsf{get}(r)$ and $V_2 = \lambda x.\mathsf{set}(r, x)$ to this function. They can be given the typing judgments

$$r : ([0, 1], \mathsf{Int}); - \vdash \lambda x.\mathsf{get}(r) : \mathsf{Unit} \multimap \mathsf{Int}$$
$$r : ([1, 0], \mathsf{Int}); - \vdash \lambda x.\mathsf{set}(r, x) : \mathsf{Int} \multimap \mathsf{Unit}$$

which respectively contain an affine read usage and an affine write usage on region r. It is then possible to derive the judgment

$$r : ([1, 1], \mathsf{Int}); - \vdash ((\lambda f_1.\lambda f_2.(f_1\star \parallel f_2\star))V_1)V_2 : \mathbf{B}$$

and we obtain the region usage $[1, 1]$ which guarantees that r will be read and written at most once. Such an affine region usage would not be compatible with an intuitionistic type system that cannot guarantee that the arguments of a function are used at most once. In fact, V_1 and V_2 cannot be marked with a bang since the predicate *naff*(R) is not compatible with affine region usages.

3.3.3 Properties

The properties established for the purely intuitionistic type system of $\lambda^{\|}$ can be lifted to the affine-intuitionistic type system of $\lambda^{!\|}$.

First, the weakening property adapts in a straightforward way.

Lemma 3.3.4 (Weakening)**.** *If* $R;\Gamma \vdash P : \alpha$ *and* $R \uplus R' \vdash \Gamma, \Gamma'$ *then* $R \uplus R';\Gamma,\Gamma' \vdash P : \alpha$.

Proof. By induction on the typing of P. It should be noted that the proof relies on the fact that the promotion rule permits to extend variable and region contexts with affine hypotheses. □

The substitution lemma should ensure that if a term M, where x occurs free, reads a region r which can be read at most once, it should not be possible to substitute a value V for x if V also reads r. Therefore, the lemma can only be applied if the region usages of M and V can be summed together. Moreover, we distinguish between the substitution of affine and intuitionistic variables and require in the affine case that x occur at most once such that no affine region usages can be duplicated.

Lemma 3.3.5 (Substitution)**.** *If* $R_1 \uplus R_2$ *is defined then:*

1. *if* $R_1;\Gamma, x : (\lambda, A) \vdash M : \alpha$ *and* $\mathsf{FO}(x, M) \leq 1$ *and* $R_2;\Gamma \vdash V : A$ *then* $R_1 \uplus R_2;\Gamma \vdash M[V/x] : \alpha$.
2. *if* $R_1;\Gamma, x : (!, A) \vdash M : \alpha$ *and* $R_2;\Gamma \vdash !V : !A$ *then* $R_1 \uplus R_2;\Gamma \vdash M[V/x] : \alpha$.

Proof. By induction on the typing of M. We highlight the two representative cases of application (rule app) and promotion (rule prom).

(app) We treat the affine and intuitionistic sub-cases separately.

1. We have

$$\frac{R_3;\Gamma, x : (\lambda, A) \vdash M : B \multimap \alpha \qquad R_4;\Gamma, x : (\lambda, A) \vdash N : B}{R_3 \uplus R_4;\Gamma, x : (\lambda, A) \vdash MN : \alpha}$$

   Since $\mathsf{FO}(x, M) \leq 1$, x may only occur in M or N. Without loss of generality assume x occurs in N. By induction hypothesis we have

$$R_4 \uplus R_2;\Gamma \vdash N[V/x] : B$$

   and since x does not occur in M, we also have

$$R_3;\Gamma \vdash M[V/x] : B \multimap \alpha$$

   which let us conclude

$$\frac{R_3\Gamma \vdash M[V/x] : B \multimap \alpha \qquad R_4 \uplus R_2;\Gamma \vdash N[V/x] : B}{R_3 \uplus R_4 \uplus R_2;\Gamma \vdash (MN)[V/x] : \alpha}$$

2. We have

$$\frac{R_3;\Gamma, x : (!, A) \vdash M : B \multimap \alpha \qquad R_4;\Gamma, x : (!, A) \vdash N : B}{R_3 \uplus R_4;\Gamma, x : (!, A) \vdash MN : \alpha}$$

and the last rule applied to $!V$ must be

$$\frac{\mathit{naff}(R_5) \qquad R; \Delta_! \vdash V : A}{R_2; \Gamma \vdash\ !V :\ !A}$$

where $R_2 = R_5 \uplus R'$ and $\Gamma = \Delta_!, \Theta_\lambda$ for any Θ_λ and R'. By taking Θ_λ and R' to be empty the inductive hypotheses are

$$R_3 \uplus R_5; \Gamma \vdash M[V/x] : B \multimap \alpha$$
$$R_4 \uplus R_5; \Gamma \vdash N[V/x] : B$$

The crucial point is to remark that R_5 does not contain affine region usages ($\mathit{naff}(R_5)$) such that we get $R_5 \uplus R_5 = R_5$. We can then derive

$$\frac{R_3 \uplus R_5; \Gamma \vdash M[V/x] : B \multimap \alpha \qquad R_4 \uplus R_5; \Gamma \vdash N[V/x] : B}{R_3 \uplus R_4 \uplus R_5; \Gamma \vdash (MN)[V/x] : \alpha}$$

By weakening (Lemma 3.3.4) we conclude

$$R_3 \uplus R_4 \uplus R_2; \Gamma \vdash (MN)[V/x] : \alpha$$

(prom) 1. We have

$$\frac{\mathit{naff}(R_1) \qquad R_1; \Gamma_! \vdash M : B}{R_1 \uplus R'; \Gamma_!, \Delta_\lambda, x : (\lambda, A) \vdash\ !M :\ !B}$$

It follows that x does not occur free in $!M$ and we get

$$R_1 \uplus R'; \Gamma_!, \Delta_\lambda \vdash (!M)[V/x] :\ !B$$

and by weakening (Lemma 3.3.4) we get

$$R_1 \uplus R' \uplus R_2; \Gamma_!, \Delta_\lambda \vdash (!M)[V/x] :\ !B$$

2. We have

$$\frac{\mathit{naff}(R_1) \qquad R_1; \Gamma_!, x : (!, A) \vdash M : B}{R_1 \uplus R'; \Gamma_!, x : (!, A), \Delta_\lambda \vdash\ !M :\ !B}$$

and the last rule applied to $!V$ must be

$$\frac{\mathit{naff}(R_2) \qquad R_2; \Gamma_! \vdash V : A}{R_2, R''; \Gamma_!, \Delta_\lambda \vdash\ !V :\ !A}$$

By taking R'' and Δ_λ to be empty the inductive hypothesis is

$$R_1 \uplus R_2; \Gamma_!, \vdash M[V/x] : B$$

Since $\mathit{naff}(R_1)$ and $\mathit{naff}(R_2)$ we have $\mathit{naff}(R_1 \uplus R_2)$ and we can conclude

$$\frac{\mathit{naff}(R_1 \uplus R_2) \qquad R_1 \uplus R_2; \Gamma_! \vdash M[V/x] : B}{R_1 \uplus R_2, R', R''; \Gamma_!, \Delta_\lambda \vdash\ !M[V/x] :\ !B}$$

□

We then want to show that typing is preserved by reduction.

Proposition 3.3.6 (Subject reduction)**.** *If* $R;\Gamma \vdash P : \alpha$ *and* $P \longrightarrow P'$ *then* $R;\Gamma \vdash P' : \alpha$.

To prove the subject reduction property, we need to state in the following lemmas that the structural equivalence and the reduction rules preserve typing.

Lemma 3.3.7 (Structural equivalence preserves typing)**.** *If* $R;\Gamma \vdash P : \alpha$ *and* $P \equiv P'$ *then* $R;\Gamma \vdash P' : \alpha$.

Proof. By induction on the proof of structural equivalence. This is is mainly a matter of reordering the pieces of the typing proof of P so as to obtain a typing proof of P'. □

Lemma 3.3.8 (Contexts preserve typing)**.** *Suppose that in the proof of* $R;\Gamma \vdash C[E[M]] : \alpha$ *we prove* $R';\Gamma' \vdash M : A$*. Then replacing* M *with a* M' *such that* $R';\Gamma' \vdash M' : A$*, we can still derive* $R;\Gamma \vdash C[E[M']] : \alpha$.

Proof. By induction on the structure of E and C. □

Lemma 3.3.9 (Functional redexes)**.** *If* $R;\Gamma \vdash C[E[\Delta]] : \alpha$ *where* Δ *has the shape* $(\lambda x.M)V$ *or* $\mathsf{let}\ !x = V\ \mathsf{in}\ M$ *then* $R;\Gamma \vdash C[E[M[V/x]]] : \alpha$.

Proof. If $\Delta = (\lambda x.M)V$ it must be the case that $\mathsf{FO}(x, M) \leq 1$ and we can appeal to the affine substitution lemma (Lemma 3.3.5-1) and if $\Delta = \mathsf{let}\ !x = V\ \mathsf{in}\ M$ we rely on the intuitionistic lemma (Lemma 3.3.5-2). This settles the case where the evaluation context E is trivial. If it is complex then we also need Lemma 3.3.8. □

Lemma 3.3.10 (Side effect redexes)**.** *If* $R;\Gamma \vdash \Delta : \alpha$ *where* Δ *is one of the programs on the left-hand side of the rules below then* $R;\Gamma \vdash \Delta' : \alpha$ *where* Δ' *is the corresponding program on the right-hand side.*

$$\begin{array}{lrcl} (\mathsf{get}) & C[E[\mathsf{get}(r)]] \parallel r \Leftarrow V & \longrightarrow & C[E[V]] \\ (\mathsf{set}) & C[E[\mathsf{set}(r, V)]] & \longrightarrow & C[E[\star]] \parallel r \Leftarrow V \end{array}$$

Proof. We proceed by case analysis.

(get) We have

$$R_1 \uplus R_2;\Gamma \vdash C[E[\mathsf{get}(r)]] \parallel r \Leftarrow V : \alpha$$

which must be derived from

$$\dfrac{\dfrac{r : ([x, y], A) \in R_3 \quad y \neq 0}{R_3;\Gamma \vdash \mathsf{get}(r) : A}}{\vdots}$$
$$R_1;\Gamma \vdash C[E[\mathsf{get}(r)]] : \alpha$$

for some R_3 and

$$\frac{R_2 = r : ([x, y], A) \uplus R_2' \quad x \neq 0 \quad R_2'; \Gamma \vdash V : A}{R_2; \Gamma \vdash r \Leftarrow V : \mathbf{B}}$$

We distinguish two cases: whether E is of the shape $E_1[!E_2]$ or not.

(a) $E \neq E_1[!E_2]$.
Since $R_1 \uplus R_2$ is defined it is clear that $R_2 \uplus R_3$ is also defined. Hence by weakening we have

$$R_2' \uplus R_3; \Gamma \vdash V : A$$

Moreover E is not of the shape $E_1[!E_2]$ hence it is clear that the promotion rule prom is not used to type $C[E[\mathsf{get}(r)]]$. Therefore we can derive

$$R_1 \uplus R_2; \Gamma \vdash C[E[V]] : \alpha$$

even though $\mathit{naff}(R_2 \uplus R_3)$ is false.

(b) $E = E_1[!E_2]$.
The promotion rule prom must be used to type $C[E[\mathsf{get}(r)]]$, hence the predicate $\mathit{naff}(R_3)$ must be true, which implies that A is of the shape $!B$ and so that V is of the shape $!V'$. Thus the last rule applied to $!V'$ must be prom:

$$\frac{\mathit{naff}(R_4) \qquad R_4; \Gamma \vdash V' : B}{R_2; \Gamma \vdash !V' : !B}$$

where $R_2 = R_4, R_5$ for some R_5. Since $R_4 \uplus R_3$ must be defined, by weakening we have

$$R_4 \uplus R_3; \Gamma \vdash !V' : !B$$

By observing that $\mathit{naff}(R_4 \uplus R_3)$ is true we can derive

$$R_1 \uplus R_2; \Gamma \vdash C[E[!V']] : \alpha$$

where the context R_5 may be weakened in the end of the derivation.

(set) We have

$$\dfrac{\dfrac{R_1 = r : ([x, y], A) \uplus R_1' \quad x \neq 0 \quad R_1'; \Gamma \vdash V : A}{R_1; \Gamma \vdash \mathsf{set}(r, V) : \mathsf{Unit}}}{\dfrac{\vdots}{R; \Gamma \vdash C[E[\mathsf{set}(r, V)]] : \alpha}}$$

for some R_1. Therefore we also have

$$R_1; \Gamma \vdash r \Leftarrow V : \mathsf{Unit}$$

Then from

$$R_0; \Gamma \vdash \star : \mathsf{Unit}$$

where R_0 only contains neutral region usages we can derive

$$R_2; \Gamma \vdash C[E[\star]] : \alpha$$

such that $R_1 \uplus R_2 = R$. Thus we conclude

$$R; \Gamma \vdash C[E[\star]] \parallel r \Leftarrow V : \alpha$$

□

With the above lemmas we can now turn to the proof of subject reduction.

Proof of Proposition 3.3.6 (subject reduction). We recall that $P \longrightarrow P'$ means that $P \equiv \Delta$ such that Δ reduces to Δ' without using structural equivalence and $\Delta' \equiv P'$. By Lemma 3.3.7 we get $R; \Gamma \vdash \Delta : \alpha$. By Lemmas 3.3.9 and 3.3.10 we derive $R; \Gamma \vdash \Delta' : \alpha$. Again by Lemma 3.3.7 we conclude $R; \Gamma \vdash P' : \alpha$. □

Finally we can state the progress property.

Proposition 3.3.11 (Progress)**.** *Suppose P is closed typable program which cannot reduce. Then P is structurally equivalent to a program*

$$M_1 \parallel \cdots \parallel M_m \parallel S \qquad m \geq 0$$

where M_i is either a value or can be uniquely decomposed as a term $E[\mathsf{get}(r)]$ such that no assignment to r exists in the store S.

The proof follows by observing that a closed value of type $!A$ must have the shape $!V$, which is formalized in the following lemma.

Lemma 3.3.12 (Classification)**.** *Assume $R; - \vdash V : A$. Then:*

- *if $A = A_1 \multimap A_2$ then $V = \lambda x.M$,*
- *if $A = !A_1$ then $V = !V_1$*

Proof. By observing that the last typing rules applied to V must be lam or prom. □

Then we are sure that a well-typed closed term $\mathsf{let}\ !x = V\ \mathsf{in}\ M$ is guaranteed to reduce since V must match the pattern $!V'$. The general proof is a follows.

Proof of Proposition 3.3.11. We proceed by induction on the structure of the threads M_i to show that each one of them is either a value or a stuck get. We highlight two relevant cases.

- $M_i = PQ$
  By looking at the evaluation contexts we know that P cannot reduce. By induction hypothesis we have two cases: either P is a value or P is a stuck get.

– Suppose P is a value. We have

$$\frac{R_1; - \vdash P : A \multimap \alpha \qquad R_2; - \vdash Q : A}{R_1 \uplus R_2; - \vdash PQ : \alpha}$$

and by Lemma 3.3.12 we get $P = \lambda x.M$. It must be the case that Q cannot reduce but it cannot be a value for otherwise PQ reduces by a (β_v) step. Hence Q is a stuck get and PQ is of the form $PE[\mathsf{get}(r)]$.

– Suppose P is a stuck get. Then PQ is of the form $E[\mathsf{get}(r)]Q$.

- $M_i = \mathsf{let}\ !x = P\ \mathsf{in}\ Q$
We know that $\mathsf{let}\ !x = P\ \mathsf{in}\ Q$ cannot reduce which by looking at the evaluation contexts means that P cannot reduce. By induction hypothesis we have two cases: either P is a value or P is a stuck get.

  – Suppose P is a value. We have

$$\frac{R_1; - \vdash P : !A \qquad R_2; x : (!, A) \vdash Q : \alpha}{R_1 \uplus R_2; - \vdash \mathsf{let}\ !x = P\ \mathsf{in}\ Q : \alpha}$$

By Lemma 3.3.12 we have $P = !V$ hence $\mathsf{let}\ !x = !V\ \mathsf{in}\ Q$ is a redex and this contradicts the hypothesis that $\mathsf{let}\ !x = P\ \mathsf{in}\ Q$ cannot reduce. Thus P cannot be a value.

  – Suppose P is a stuck get. Then $\mathsf{let}\ !x = P\ \mathsf{in}\ Q$ is of the form $\mathsf{let}\ !x = E[\mathsf{get}(r)]\ \mathsf{in}\ Q$.

□

3.3.4 References

We have seen in Section 3.1 that $\lambda^!$ simulates the simply typed λ-calculus. We may then wonder if $\lambda^{!\|}$ simulate $\lambda^\|$. The only ambiguous point comes from the reading rule (get). Recall that in $\lambda^\|$, reading a region amounts to *copy* the value from the store with the following rule:

$$\mathsf{get}(r) \parallel r \Leftarrow V \longrightarrow V \parallel r \Leftarrow V$$

We have omitted the evaluation contexts for simplicity. In $\lambda^{!\|}$, for duplication reasons, reading a region amounts to *consume* the value from the store:

$$\mathsf{get}(r) \parallel r \Leftarrow V \longrightarrow V \tag{3.5}$$

In other words, $\lambda^{!\|}$ allows the implementation of the operational semantics of communication channels but apparently not the one of imperative references.

The goal of this subsection is to quickly show that in $\lambda^{!\|}$, a region may abstract a set of references only if the content of the region is duplicable. For this it suffices to use a *consume-and-rewrite* mechanism as follows:

$$\mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, !x); !x \parallel r \Leftarrow !V \longrightarrow^* !V \parallel r \Leftarrow !V \tag{3.6}$$

After being consumed, the value $!V$ is duplicated: one occurrence is immediately rewritten to the store and the other one is made available to the rest of the program. This program is typable if the region r is associated to a type of the shape $!A$.

We notice however that the above program (3.6) requires the usage $[1,1]$ or $[\infty,\infty]$ for r whereas the program (3.5) only requires the usage $[0,1]$ or $[0,\infty]$ where the write component is neutral. The consume-and-rewrite mechanism thus implies that no affine write usage can be used at another point of the program. We will see in the next section that write usages are precious, and in order to get around this limitation it is convenient to consider these two reading rules in $\lambda^{!\|}$:

$$\begin{array}{lllll} (\mathsf{get}) & C[E[\mathsf{get}(r)]] \parallel r \Leftarrow V & \longrightarrow & C[E[V]] & \text{if } V \neq !V' \\ (\mathsf{get}_!) & C[E[\mathsf{get}(r)]] \parallel r \Leftarrow !V & \longrightarrow & C[E[!V]] \parallel r \Leftarrow !V & \end{array}$$

The $(\mathsf{get}_!)$ rule internalizes the consume-and-rewrite mechanism. The side condition of the rule (get) ensures that the choice of the reduction rule is deterministic if V is of the shape $!V$.

To conclude, the rule $(\mathsf{get}_!)$ provides a mechanism to implement the reading of a reference with a neutral write usage and it is straightforward to see that the proofs of subject reduction (Proposition 3.3.6) and progress (Proposition 3.3.11) can be adapted. In the rest of this chapter we consider that $\lambda^{!\|}$ make the distinction between (get) and $(\mathsf{get}_!)$.

3.4 Confluence

We now explain how region usages can be tamed to ensure the confluence of well-typed programs.

In $\lambda^{!\|}$, non-determinism may come from accesses to the store. For example if different values (whether they are affine or intuitionistic) are stored in the same region:

$$\begin{array}{ccc} & \mathsf{get}(r) \parallel r \Leftarrow V_1 \parallel r \Leftarrow V_2 & \\ \swarrow & & \searrow \\ V_1 \parallel r \Leftarrow V_2 & & V_2 \parallel r \Leftarrow V_1 \end{array} \tag{3.7}$$

$$\begin{array}{ccc} & \mathsf{get}(r) \parallel r \Leftarrow !V_1 \parallel r \Leftarrow !V_2 & \\ \swarrow & & \searrow \\ !V_1 \parallel r \Leftarrow !V_1 \parallel r \Leftarrow !V_2 & & !V_1 \parallel r \Leftarrow !V_1 \parallel r \Leftarrow !V_2 \end{array} \tag{3.8}$$

where V_1 and V_2 are not of the shape $!V$. Or if there is a race condition on a region that contains an affine value:

$$\begin{array}{ccc} & E_1[\mathsf{get}(r)] \parallel E_2[\mathsf{get}(r)] \parallel r \Leftarrow V & \\ \swarrow & & \searrow \\ E_1[V] \parallel E_2[\mathsf{get}(r)] & & E_1[\mathsf{get}(r)] \parallel E_2[V] \end{array} \tag{3.9}$$

where V is not of the shape $!V'$. On the other hand, a race condition on an intuitionistic value such as

$$\begin{array}{ccc} & E_1[\mathsf{get}(r)] \parallel E_2[\mathsf{get}(r)] \parallel r \Leftarrow !V & \\ \swarrow & & \searrow \\ E_1[!V] \parallel E_2[\mathsf{get}(r)] \parallel r \Leftarrow !V & & E_1[\mathsf{get}(r)] \parallel E_2[!V] \parallel r \Leftarrow !V \\ \searrow & & \swarrow \\ & E_1[!V] \parallel E_2[!V] \parallel r \Leftarrow !V & \end{array} \tag{3.10}$$

does not compromise determinism because the two possible reductions commute.

We propose two conditions to rule out the problematic situations (3.7), (3.8) and (3.9):

1. Every region can be assigned one value, at most.
2. Affine stores (that contain non duplicable values) can be read at most once.

The first condition can be imposed by removing the usage $[\infty, \infty]$ and for the second condition it suffices to type affine regions with usages of the set $\{[1,1],[1,0],[0,1],[0,0]\}$. To this end, we introduce in Figure 3.11 rules for the typing of writes and stores that are alternative to those of Figure 3.10. We denote with $\vdash_C$ provability in this restricted system.

It is easy to see that the program (3.10) is still typable in this restricted system. If we assume E_1 and E_2 do not contain any write operation on r, the typing of $E_1[\mathsf{get}(r)] \parallel E_2[\mathsf{get}(r)]$ requires r to have the usage $[0, \infty]$. And since the store contains an intuitionistic value, we can use the rule $\mathsf{store}_!$ to associate the usage $[1, \infty]$ to r. We conclude that the program is typable by observing that $[0, \infty] \uplus [1, \infty]$ is defined.

This system still enjoys the subject reduction property and moreover the typable programs are strongly confluent.

Proposition 3.4.1 (Subject reduction and confluence)**.** *Suppose* $R; \Gamma \vdash_C P : \alpha$. *Then:*

region usages are restricted to these two sets :
$\{[1,\infty],[0,\infty]\} \cup \{[1,1],[1,0],[0,1],[0,0]\}$

$$\mathsf{set}\ \frac{A \neq\ !B \qquad U \in \{[1,1],[1,0]\} \qquad R = r : (U, A) \uplus R' \qquad R'; \Gamma \vdash V : A}{R; \Gamma \vdash \mathsf{set}(r, V) : \mathsf{Unit}}$$

$$\mathsf{store}\ \frac{A \neq\ !B \qquad U \in \{[1,1],[1,0]\} \qquad R = r : (U, A) \uplus R' \qquad R'; \Gamma \vdash V : A}{R; \Gamma \vdash r \Leftarrow V : \mathbf{B}}$$

$$\mathsf{set}_!\ \frac{U \in \{[1,\infty]\} \cup \{[1,1],[1,0]\} \qquad R = r : (U, !A) \uplus R' \qquad R'; \Gamma \vdash V : !A}{R; \Gamma \vdash \mathsf{set}(r, V) : \mathsf{Unit}}$$

$$\mathsf{store}_!\ \frac{U \in \{[1,\infty]\} \cup \{[1,1],[1,0]\} \qquad R = r : (U, !A) \uplus R' \qquad R'; \Gamma \vdash V : !A}{R; \Gamma \vdash r \Leftarrow V : \mathbf{B}}$$

Figure 3.11: Restricted region usages and rules for confluence

1. *If* $P \longrightarrow P'$ *then* $R; \Gamma \vdash_C P' : \alpha$.
2. *If* $P'' \longleftarrow P \longrightarrow P'$ *then there is a* Q *such that* $P' \longrightarrow Q \longleftarrow P''$.

Proof.

1. We just have to reconsider the case of the rule (set) and see that the proof simply adapts from the one of Proposition 3.3.6.
2. Let us consider each problematic critical pair and see how the restrictions on region usages forbid their typing.
   - (3.7) and (3.8). Clearly, the rules store and $\mathsf{store}_!$ do not allow the typing of several stores in parallel.
   - (3.9) The two $\mathsf{get}(r)$s in parallel require r to have the usage $[0,\infty]$ while the the affine store which must be typed with the rule store associates r with the usage $[1,0]$. But since $[0,\infty]$ and $[1,0]$ do not belong to the same set of usages they cannot be summed.

□

Remark 3.4.2. We note that the rules for ensuring confluence require that at most one value is associated with a region, which usually called a *single-assignment* discipline. This is quite restrictive but one has to keep in mind that it targets regions that can be accessed concurrently by several threads. Of course, the discipline could be relaxed for the regions that are accessed by one

single sequential thread and this could perhaps be realized with the help of an effect system.

Related work The conditions to guarantee confluence are inspired by the work of Kobayashi *et al.* on linearity in the π-calculus [KPT99] and one should expect a comparable expressive power. Also, much more elaborate notions of usages have been proposed to analyze the resource usage of programs [Kob02, IK05]. And recently, Amadio and Dogguy derived a notion of affine usage in the context of synchronous programming [AD08] that guarantees the determinacy of programs.

3.5 Termination

In this last section we show that the discipline of region stratification that we introduced earlier for $\lambda^{\parallel}$ combines smoothly with the discipline of region usage, so that confluence *and* termination can be guaranteed.

The general approach is similar to the intuitionistic case: we decorate the affine-intuitionistic type system with effects and we impose the stratification of regions via well-formation rules on region contexts. In order to prove the termination of a stratified $\lambda_S^{!\parallel}$, it suffices to give a translation from the affine-intuitionistic system to the intuitionistic system that maps a reduction step of $\lambda_S^{!\parallel}$ to exactly one reduction step of $\lambda_S^{\parallel}$. Therefore, termination in $\lambda_S^{\parallel}$ (Theorem 2.3.1) entails termination in $\lambda_S^{!\parallel}$.

3.5.1 An affine-intuitionistic type and effect system

First we need to extend the syntax of types presented in Figure 3.8 such that the affine functional type is decorated with an effect:

$$A \overset{e}{\multimap} \alpha$$

We can then state in Figure 3.12 the rules that stratify region contexts. Similarly to the stratified region contexts of $\lambda^{\parallel}$ (Figure 2.8), a region may only contain values that have side effects on 'lower' regions.

A type and effect judgment has the shape

$$R; \Gamma \vdash P : (\alpha, e)$$

where the effect e provides and upper bound on the set of regions on which the program P may read or write. The rules of the affine-intuitionistic type and effect system are spelled out in Figure 3.13.

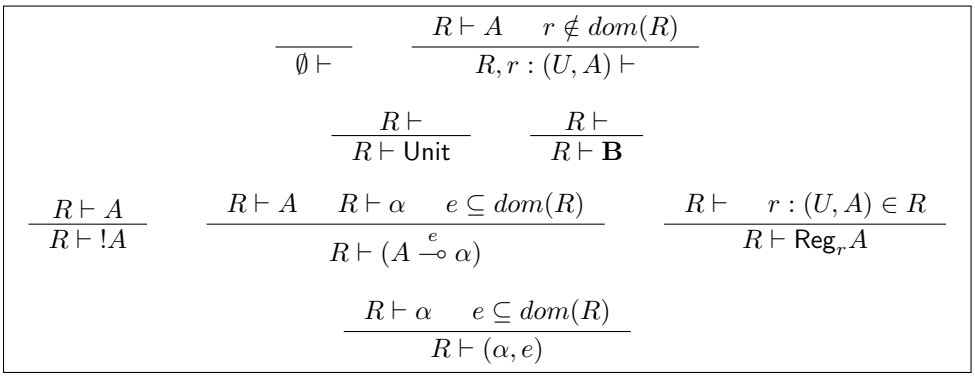


Figure 3.12: Stratified formation of types and contexts

Also, subtyping rules adapt in a straightforward way to affine typing as specified in Figure 3.14.

It is easy to verify that the stratified system is a restriction of the unstratified one and that the subject reduction property still holds in the restricted stratified system. If confluence is required, it suffices to add the restrictions spelled out in Figure 3.11.

3.5.2 A forgetful translation

In this last section we show that $\lambda^{\parallel}$ simulates $\lambda^{!\parallel}$ by a forgetful translation such that reduction steps are in one-to-one correspondence. This allows us at the end to prove the termination of $\lambda_S^{!\parallel}$.

We recall that there is a standard forgetful translation from affine-intuitionistic logic to intuitionistic logic which amounts to forget about usages and the bang modality, and to regard the affine implication as an ordinary intuitionistic implication. In Figure 3.15, we lift the forgetful translation to go from $\lambda_S^{!\parallel}$ to $\lambda_S^{\parallel}$.

To avoid confusion, in the following we write $\vdash_{!S}$ and $\longrightarrow_{!S}$ (respectively $\vdash_S$ and $\longrightarrow_S$) for provability and the reduction relation in $\lambda_S^{!\parallel}$ (resp. $\lambda_S^{\parallel}$).

Lemma 3.5.1. *The forgetful translation preserves provability in the following sense:*

1. *If $R \vdash_{!S}$ then $\underline{R} \vdash_S$.*
2. *If $R \vdash_{!S} \alpha$ then $\underline{R} \vdash_S \underline{\alpha}$.*
3. *If $R \vdash_{!S} (\alpha, e)$ then $\underline{R} \vdash_S (\underline{\alpha}, e)$.*

$$\mathsf{var}\frac{R \vdash \Gamma \qquad x : (u, A) \in \Gamma}{R; \Gamma \vdash x : (A, \emptyset)} \qquad \mathsf{unit}\frac{R \vdash \Gamma}{R; \Gamma \vdash \star : (\mathsf{Unit}, \emptyset)}$$

$$\mathsf{reg}\frac{R \vdash \Gamma \qquad r : (U, A) \in R}{R; \Gamma \vdash r : (\mathsf{Reg}_r A, \emptyset)}$$

$$\mathsf{lam}\frac{\mathsf{FO}(x, M) \leq 1 \qquad R; \Gamma, x : (\lambda, A) \vdash M : (\alpha, e)}{R; \Gamma \vdash \lambda x.M : (A \xrightarrow{e} \alpha, \emptyset)}$$

$$\mathsf{app}\frac{R_1; \Gamma \vdash M : (A \xrightarrow{e_1} \alpha, e_2) \qquad R_2; \Gamma \vdash N : (A, e_3)}{R_1 \uplus R_2; \Gamma \vdash MN : (\alpha, e_1 \cup e_2 \cup e_3)}$$

$$\mathsf{prom}\frac{\mathit{naff}(R) \qquad R; \Gamma_! \vdash M : (A, e)}{R, R'; \Gamma_!, \Delta_\lambda \vdash !M : (!A, e)}$$

$$\mathsf{elim}\frac{R_1; \Gamma \vdash V : (!A, \emptyset) \qquad R_2; \Gamma, x : (!, A) \vdash N : (\alpha, e)}{R_1 \uplus R_2; \Gamma \vdash \mathsf{let}\ !x = V\ \mathsf{in}\ N : (\alpha, e)}$$

$$\mathsf{get}\frac{r : ([x, y], A) \in R \qquad y \neq 0 \qquad R \vdash \Gamma}{R; \Gamma \vdash \mathsf{get}(r) : (A, \{r\})}$$

$$\mathsf{set}\frac{R = r : ([x, y], A) \uplus R' \qquad x \neq 0 \qquad R'; \Gamma \vdash V : (A, \emptyset)}{R; \Gamma \vdash \mathsf{set}(r, V) : (\mathsf{Unit}, \{r\})}$$

$$\mathsf{store}\frac{R = r : ([x, y], A) \uplus R' \qquad x \neq 0 \qquad R'; \Gamma \vdash V : (A, \emptyset)}{R; \Gamma \vdash r \Leftarrow V : (\mathbf{B}, \emptyset)}$$

$$\mathsf{par}_1\frac{R_1; \Gamma \vdash P : (\alpha, e) \qquad R_2; \Gamma \vdash S : (\mathbf{B}, \emptyset)}{R_1 \uplus R_2; \Gamma \vdash P \parallel S : (\alpha, e)}$$

$$\mathsf{par}_2\frac{i = 1, 2 \qquad R_i; \Gamma \vdash P_i : (\alpha_i, e_i)}{R_1 \uplus R_2; \Gamma \vdash P_1 \parallel P_2 : (\mathbf{B}, e_1 \cup e_2)}$$

Figure 3.13: Affine-intuitionistic type and effect system of $\lambda^{!\parallel}$

$$\frac{}{R \vdash \alpha \leq \alpha} \qquad \frac{R \vdash A \leq A'}{R \vdash\, !A \leq\, !A'}$$

$$\frac{e \subseteq e' \subseteq dom(R) \quad R \vdash A' \leq A \quad R \vdash \alpha \leq \alpha'}{R \vdash (A \overset{e}{\multimap} \alpha) \leq (A' \overset{e'}{\multimap} \alpha')}$$

$$\frac{e \subseteq e' \subseteq dom(R) \quad R \vdash \alpha \leq \alpha'}{R \vdash (\alpha, e) \leq (\alpha', e')} \qquad \mathsf{sub}\frac{R;\Gamma \vdash M : (\alpha, e) \quad R \vdash (\alpha, e) \leq (\alpha', e')}{R;\Gamma \vdash M : (\alpha', e')}$$

Figure 3.14: Subtyping induced by effect containment

$$\begin{array}{rcl}
\underline{\mathsf{Unit}} & = & \mathsf{Unit} \\
\underline{\mathsf{Int}} & = & \mathsf{Int} \\
\underline{A \overset{e}{\multimap} \alpha} & = & \underline{A} \overset{e}{\to} \underline{\alpha} \\
\underline{!A} & = & \underline{A} \\
 & & \\
\underline{r_1 : (U_1, A_1), \ldots, r_n : (U_n, A_n)} & = & r_1 : \underline{A_1}, \ldots, r_n : \underline{A_n} \\
\underline{x_1 : (u_1, A_1), \ldots, x_n : (u_n, A_n)} & = & x_1 : \underline{A_1}, \ldots, x_n : \underline{A_n} \\
 & & \\
\underline{x} & = & x \\
\underline{r} & = & r \\
\underline{\star} & = & \star \\
\underline{\lambda x.M} & = & \lambda x.\underline{M} \\
\underline{MN} & = & \underline{M}\,\underline{N} \\
\underline{!M} & = & \underline{M} \\
\underline{\mathsf{let}\ !x = V\ \mathsf{in}\ M} & = & (\lambda x.\underline{M})\underline{V} \\
\underline{\mathsf{get}(r)} & = & \mathsf{get}(r) \\
\underline{\mathsf{set}(r, V)} & = & \mathsf{set}(r, \underline{V}) \\
\underline{r \Leftarrow V} & = & r \Leftarrow \underline{V} \\
\underline{P \parallel P} & = & \underline{P} \parallel \underline{P}
\end{array}$$

Figure 3.15: Forgetful typed translation from $\lambda^{!\parallel}$ to $\lambda^{\parallel}$

4. *If* $R \vdash_{!S} \alpha \leq \alpha'$ *then* $\underline{R} \vdash_S \underline{\alpha} \leq \underline{\alpha'}$.
5. *If* $R \vdash_{!S} (\alpha, e) \leq (\alpha', e')$ *then* $\underline{R} \vdash_S (\underline{\alpha}, e) \leq (\underline{\alpha'}, e')$.
6. *If* $R \vdash_{!S} \Gamma$ *then* $\underline{R} \vdash_S \underline{\Gamma}$.
7. *If* $R; \Gamma \vdash_{!S} P : (\alpha, e)$ *then* $\underline{R}; \underline{\Gamma} \vdash_S \underline{P} : (\underline{\alpha}, e)$.

Proof. By induction on the provability relation $\vdash_{!S}$. Concerning the rules for types and region contexts formation and for subtyping, the forgetful translation provides a one-to-one mapping from the rules of the affine-intuitionistic system to the rules of the intuitionistic one (the only exception are the rules for $!A$ which become trivial in the intuitionistic framework). Also note that $dom(R) = dom(\underline{R})$ and if $R_1 \uplus R_2$ is defined then $\underline{R_1} = \underline{R_2} = \underline{R_1 \uplus R_2}$. With these remarks in mind, the proofs are straightforward. □

Next we want to relate the reduction of a program and of its translation. The little subtlety is that in the intuitionistic system values are copied from the store while they might be consumed in the affine-intuitionistic system. Consequently, a reduction such as:

$$\mathsf{get}(r) \parallel r \Leftarrow V \longrightarrow_{!S} V$$

where V is not of the shape $!V'$ might be simulated by

$$\mathsf{get}(r) \parallel r \Leftarrow \underline{V} \longrightarrow_S \underline{V} \parallel r \Leftarrow \underline{V}$$

In other terms, the translated program may contain more values in the store than the source program. To account for this, we introduce a 'simulation' relation $\mathcal{S}$ indexed on a pair $R; \Gamma$ such that $R \vdash \Gamma$:

$$\begin{aligned} \mathcal{S}_{R;\Gamma} = \{(P, Q) \mid\ & R; \Gamma \vdash_{!S} P : (\alpha, e), \\ & \underline{R}; \underline{\Gamma} \vdash_S Q : (\underline{\alpha}, e), \\ & Q \equiv (\underline{P} \parallel S)\} \end{aligned}$$

Lemma 3.5.2 (Simulation)**.** *If* $(P, Q) \in \mathcal{S}_{R;\Gamma}$ *and* $P \longrightarrow_{!S} P'$ *then* $Q \longrightarrow_S Q'$ *and* $(P', Q') \in \mathcal{S}_{R;\Gamma}$.

Proof. Suppose $(P, Q) \in \mathcal{S}_{R;\Gamma}$. First note that the typing must be preserved by subject reduction of $\vdash_{!S}$ and $\vdash_S$. Then by definition, $P \longrightarrow_{!S} P'$ means that P is structurally equivalent to a process P_1 which can be decomposed in a *redex* $C[E[M]]$. We notice that the forgetful translation preserves structural equivalence, namely if $P \equiv P_1$ then $\underline{P} \equiv \underline{P_1}$. Indeed, the commutativity and associativity rules of the affine-intuitionistic system match those of the intuitionistic system. We also remark that the forgetful translation can be extended to evaluation contexts simply by defining $\underline{[\cdot]} = [\cdot]$. Then we note that the translation of an evaluation context is an intuitionistic evaluation context. In particular, this holds because the translation of a value is still a value.

Following these remarks, we can derive that $Q \equiv \underline{C}[\underline{E}[\underline{M}]] \parallel S$. Thus it is enough to focus on M and show that each reduction in the affine-intuitionistic system is mapped to a reduction in the intuitionistic one and that the resulting program is

still related to the program P' via the relation $\mathcal{S}_{R;\Gamma}$. To this end, we notice that the translation commutes with the substitution so that $\underline{M[V/x]} = \underline{M}[\underline{V}/x]$. Then one proceeds by case analysis of M. Let us look at two cases in some detail.

- $M = \mathsf{let}\ !x = !V\ \mathsf{in}\ N$.
  We have
  $$\mathsf{let}\ !x = !V\ \mathsf{in}\ N \longrightarrow_{!S} N[V/x]$$
  and then
  $$\begin{array}{lcl} \underline{\mathsf{let}\ !x = !V\ \mathsf{in}\ N} & = & (\lambda x.\underline{N})\underline{V} \\ & \longrightarrow_S & \underline{N}[\underline{V}/x] \\ & = & \underline{N[V/x]} \end{array}$$

- $\mathsf{get}(r) \parallel r \Leftarrow V$ where V is not of the shape $!V'$.
  We have
  $$\mathsf{get}(r) \parallel r \Leftarrow V \longrightarrow_{!S} V$$
  and then
  $$\begin{array}{lcl} \underline{\mathsf{get}(r) \parallel r \Leftarrow V} & = & \mathsf{get}(r) \parallel r \Leftarrow \underline{V} \\ & \longrightarrow_S & \underline{V} \parallel r \Leftarrow \underline{V} \end{array}$$
  Notice that in this case we have an additional store $r \Leftarrow \underline{V}$ which is the reason why in the definition of the relation $\mathcal{S}$ we relate a program to its translation in parallel with some additional store.

□

Corollary 3.5.3 (Termination)**.** *If $R;\Gamma \vdash_{!S} P : (\alpha, e)$ then all reductions starting from P terminate.*

Proof. By contradiction. We have $(P, \underline{P}) \in \mathcal{S}_{R;\Gamma}$ and $\underline{R};\underline{\Gamma} \vdash_S \underline{P} : (\underline{\alpha}, e)$. If there is an infinite reduction starting from P then the simulation Lemma 3.5.2 entails that there is an infinite reduction starting form $\underline{P}$ and this contradicts the termination of the intuitionistic system (Theorem 2.3.1). □

Part II

Combinatorial and Syntactic Analyzes

Chapter 4

An elementary λ-calculus

In this chapter, we introduce an *elementary λ-calculus* based on **EAL**. The main contribution is to prove that a class of *well-formed* programs terminate in elementary time under an arbitrary reduction strategy. The proof is based on an original combinatorial analysis which, as we will see in the next chapter, extends smoothly to concurrency.

Outline The chapter is organized as follows. In Section 4.1 we recall the syntax and reduction of the elementary λ-calculus which is that of $\lambda^!$. At the same time, we also introduce formally the notion of depth. The key point of **EAL** is to *stratify programs by depth levels*[1] so that the depth of occurrences is preserved by reduction. We present in Section 4.2 the principle of stratification by depth levels and show intuitively why it entails termination in elementary time of a *specific* reduction strategy. The starting point of our contribution is to propose in Section 4.3 a formal system called *elementary affine depth system* ($\lambda^!_{EAD}$), that stratifies programs by depth levels and which is a variant of the system proposed by Terui for the Light Affine λ-calculus [Ter07]. It is usually shown that a specific *shallow-first* reduction strategy (*i.e.* redexes are eliminated in depth-increasing order) can be computed in elementary time [DJ03]. In Section 4.4, we extend this result by showing that terms well-formed in $\lambda^!_{EAD}$ are guaranteed to terminate in elementary time under an arbitrary reduction strategy. For this we provide an original combinatorial proof that relies on an analysis of the depth of occurrences.

The contribution can be summarized by the diagram of Figure 4.1.

[1] Not to be confused with the stratification of regions introduced in Chapter 2.

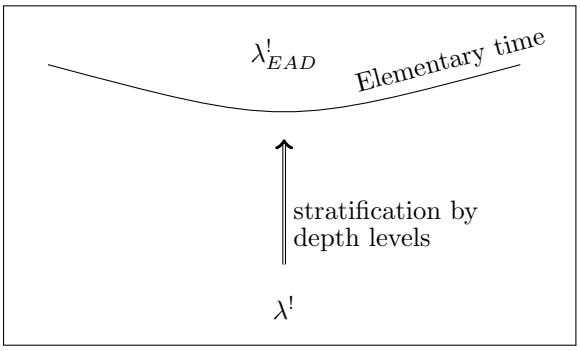


Figure 4.1: Every strategy of $\lambda^!_{EAD}$ is elementary time

4.1 Syntax, reduction and depth

The syntax and reduction of the elementary λ-calculus is that of $\lambda^!$ which we recall in Figure 4.2. It is based on the Light Affine λ-calculus of Terui [Ter07]

-terms-

$$M ::= x \mid \lambda x.M \mid MM \mid !M \mid \mathsf{let}\ !x = M\ \mathsf{in}\ M$$

-evaluation contexts-

$$E ::= [\cdot] \mid \lambda x.E \mid EM \mid ME \mid !E \mid \mathsf{let}\ !x = E\ \mathsf{in}\ M \mid \mathsf{let}\ !x = M\ \mathsf{in}\ E$$

-reduction rules-

$$(\beta) \quad E[(\lambda x.M)N] \longrightarrow E[M[N/x]]$$
$$(!) \quad E[\mathsf{let}\ !x = !N\ \mathsf{in}\ M] \longrightarrow E[M[N/x]]$$

Figure 4.2: Syntax and reduction of $\lambda^!$

except that we do not consider the paragraph modality '§' which will be used for polynomial time. We stress that an evaluation context can be any arbitrary term with a hole $[\cdot]$.

We define

$$!^0 M = M$$
$$!^{n+1} M = !(!^n M)$$

Terms $\lambda x.M$ and $\mathsf{let}\ \dagger x = N\ \mathsf{in}\ M$ bind occurrences of x in M. The set of free variables of M is denoted by $\mathsf{FV}(M)$. The number of free occurrences of x in M is denoted by $\mathsf{FO}(x, M)$. The number of free occurrences in M is denoted by $\mathsf{FO}(M)$. $M[N/x]$ denotes the term M in which each free occurrence of x has been substituted by N.

In the reduction rules, the *redex* denotes the term inside the context of the left hand-side and the *contractum* denotes the term inside the context of the right hand-side.

In order to define the notion of depth it is easier to represent a term by an *abstract syntax tree*. Variables, regions and unit constants are leaves, λ-abstractions and !-terms have one child, and applications and let !-operators have two children. A path starting from the root to a node of the tree denotes

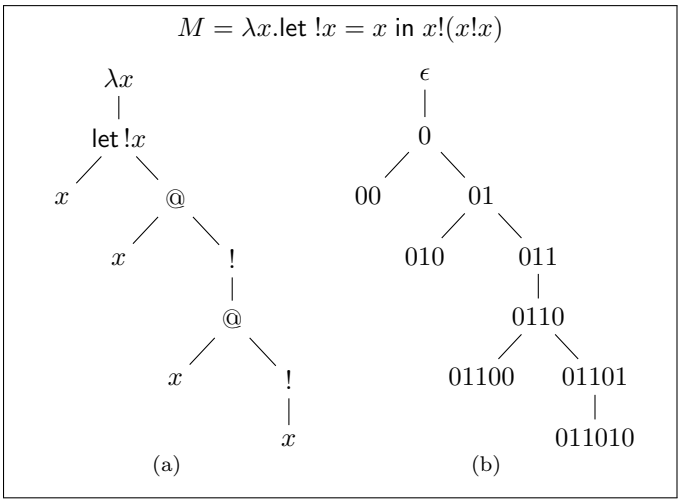


Figure 4.3: Syntax tree and addresses of M

an *occurrence* of the program whose address is a word $w \in \{0,1\}^*$. As an example, see the syntax tree given in Figure 4.3(a) and the corresponding addresses in Figure 4.3(b).

Then, we define the notion of depth.

Definition 4.1.1 (Depth)**.** The *depth* $d(w)$ of an occurrence w in a term M is the number of '!' labels that the path leading to the end node crosses. The depth $d(M)$ of the term M is the maximum depth of its occurrences.

With reference to Figure 4.3 we have:

$$\begin{aligned} d(\epsilon) = d(0) = d(00) = d(01) = d(010) = d(011) &= 0 \\ d(0110) = d(01100) = d(01101) &= 1 \\ d(011010) &= 2 \\ d(M) &= 2 \end{aligned}$$

What matters in computing the depth of an occurrence is the number of !'s that precede strictly the end node.

In the sequel, we write $\xrightarrow{i}$ when the redex occurs at depth i.

Finally, we define the size of a term as follows.

Definition 4.1.2 (Size)**.** The *size* $s(M)$ of a term M is the number of occurrences in M. The *size at depth* i $s_i(M)$ is the number of occurrences at depth i in M.

4.2 Stratification by depth levels

The logic **EAL** is designed so that the depth of occurrences is preserved by reduction. This is called the stratification of programs by depth levels. Interestingly, this stratification principle only relies on the notion of depth (types are not necessary) and the two following syntactic criteria:

1. if a λ-abstraction occurs at depth i and binds a variable x, then x must occur at most once and x must occur at depth i.
2. if a $\mathsf{let}\,!$-expression occurs at depth i and binds a variable x, then x must occur at depth $i+1$ and x may occur arbitrarily many times.

We give examples of stratified and unstratified programs in Figure 4.4. For

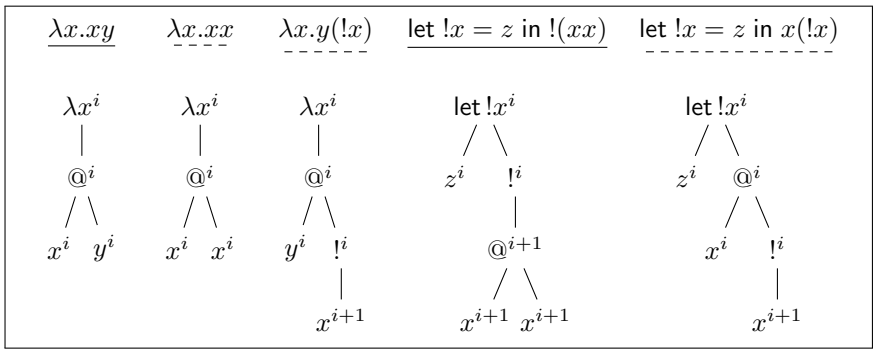


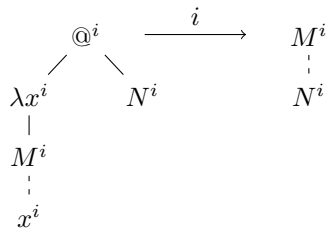
Figure 4.4: Stratified and unstratified terms

convenience, each occurrence is labelled with its depth. Underlined terms are stratified while underlined terms with a dashed line are not stratified. We explain why they are not: in $(\lambda x.xx)$, more than one occurrence of x is bound; in $(\lambda x.y(!x))$, x does not occur at depth i; and in $\mathsf{let}\ !x = z\ \mathsf{in}\ x(!x)$, one occurrence of x does not occur at depth $i+1$.

It is easy to observe that these two criteria are enough to preserve the depth of occurrences by reduction. Let us consider each reduction rule:

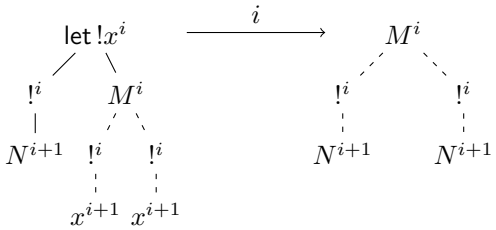
(β) $E[(\lambda x.M)N] \longrightarrow E[M[N/x]]$
Suppose the redex $(\lambda x.M)N$ occurs at depth i in E. By stratification, x may occur at most once and at depth i in M. By observing the reduction

on syntax trees, we see that N stays at depth i:



(4.1)

(!) $E[\mathsf{let}\ !x = !N\ \mathsf{in}\ M] \longrightarrow E[M[N/x]]$
Suppose the redex $\mathsf{let}\ !x = !N\ \mathsf{in}\ M$ occurs at depth i in E. By stratification, x may occur arbitrarily many times and at depth $i+1$ in M. We see that N stays at depth $i+1$:



(4.2)

In fact, we can show that stratification entails the following properties: when $M \xrightarrow{i}{}^{*} M'$,

$$d(M') \leq d(M) \tag{4.3}$$
$$s_j(M') \leq s_j(M) \text{ for } j < i \tag{4.4}$$
$$s_i(M') < s_i(M) \tag{4.5}$$
$$s(M') \leq 2^{s(M)} \tag{4.6}$$

Property (4.3) immediately follows from the fact that the depth of occurrences is preserved. Properties (4.4) and (4.5) follow from the fact that (!) is the only duplicating rule and that duplication happens at higher depth than the redex. Property (4.6) can be shown by an analysis of the size of terms.

The proof of termination in elementary time usually relies on a specific reduction strategy which applies redexes in depth-increasing order. This strategy is often referred in the literature as the *depth-by-depth* strategy but we prefer the more explicit name of *shallow-first* strategy.

Definition 4.2.1 (Shallow-first)**.** A *shallow-first reduction sequence* $M_1 \xrightarrow{i_1} M_2 \xrightarrow{i_2} \ldots \xrightarrow{i_n} M_n$ is such that $m < n$ implies $i_m \leq i_n$. A *shallow-first strategy* is a strategy that produces shallow-first sequences.

Let us prove that the shallow-first evaluation of a stratified term can be computed in elementary time. We recall that a function f on integers is elementary if there exists a k such that for any n, $f(n)$ can be computed in time $\mathcal{O}(t(n,k))$ where:

$$t(n,0) = n$$
$$t(n,k+1) = 2^{t(n,k)}$$

Proposition 4.2.2 (Elementary bounds)**.** *The shallow-first evaluation of a stratified term M terminates in at most $t(s,d)$ steps where $s = s(M)$ and $d = d(M)$. Moreover the size of the final term is bounded by $t(s,d)$.*

Proof. For simplicity, assume M is a program such that $d(M) = 2$. By properties (4.3),(4.4),(4.5) we can eliminate all the redexes of M with the shallow-first sequence $M \overset{0}{\longrightarrow}^* M' \overset{1}{\longrightarrow}^* M'' \overset{2}{\longrightarrow}^* M'''$. By property (4.6), $s(M'') \leq 2^{2^{s(M)}}$. We can safely assume that $s(M''') \leq s(M'') \leq 2^{2^{s(M)}}$. By properties (4.5) the length l of the sequence is such that $l \leq s(M) + s(M') + s(M'') = p$. Since we can show that $p \leq 2^{2^{s(M)}}$, we conclude that the shallow-first evaluation of M can be computed in $t(s(M), 2)$ steps. □

4.3 An elementary affine depth system

In this section, we introduce an *elementary affine depth system* ($\lambda^!_{EAD}$) as a set of inferences rules. Every term which is well-formed in this depth system is stratified by depth levels and terminates in elementary time for an *arbitrary* reduction strategy.

First, we introduce variable contexts Γ as follows:

$$\Gamma = x_1 : \delta_1, \ldots, x_n : \delta_n$$

Each variable x_i is associated with a natural number δ_i. We write $dom(\Gamma)$ for the set $\{x_1, \ldots, x_n\}$. A depth judgment has the shape

$$\Gamma \vdash^\delta M$$

It should entail that if x_i occurs free in M then x_i occurs at depth δ_i in $!^\delta M$.

The inference rules of the depth system are presented in Figure 4.5. Note how the depth of the bound variable in the rules lam and elim is such as to match the syntactic criteria presented at the beginning of Section 4.2. λ-abstraction are affine by the predicate $\mathsf{FO}(x, M) \leq 1$. The depth δ of the judgment is decremented by the rule prom; the intuition is that if we look at the abstract syntax tree of $!M$, M occurs deeper than $!M$.

Definition 4.3.1 (Well-formedness)**.** A term M is *well-formed* if for some Γ and δ a judgment $\Gamma \vdash^\delta M$ can be derived.

$$\mathsf{var}\,\frac{}{\Gamma, x : \delta \vdash^{\delta} x}$$

$$\mathsf{lam}\,\frac{\mathsf{FO}(x, M) \leq 1 \qquad \Gamma, x : \delta \vdash^{\delta} M}{\Gamma \vdash^{\delta} \lambda x.M} \qquad \mathsf{app}\,\frac{\Gamma \vdash^{\delta} M \qquad \Gamma \vdash^{\delta} N}{\Gamma \vdash^{\delta} MN}$$

$$\mathsf{prom}\,\frac{\Gamma \vdash^{\delta+1} M}{\Gamma \vdash^{\delta} !M} \qquad \mathsf{elim}\,\frac{\Gamma \vdash^{\delta} N \qquad \Gamma, x : (\delta + 1) \vdash^{\delta} M}{\Gamma \vdash^{\delta} \mathsf{let}\ !x = N\ \mathsf{in}\ M}$$

Figure 4.5: Elementary affine depth system $\lambda^{!}_{EAD}$

Example 4.3.2. The term $\lambda x.\mathsf{let}\ !y = x\ \mathsf{in}\ !(yy)$ is well-formed:

$$\dfrac{x : \delta \vdash^{\delta} x \qquad \dfrac{\dfrac{x : \delta, y : \delta + 1 \vdash^{\delta+1} y \qquad x : \delta, y : \delta + 1 \vdash^{\delta+1} y}{x : \delta, y : \delta + 1 \vdash^{\delta+1} yy}}{x : \delta, y : \delta + 1 \vdash^{\delta} !(yy)}}{\dfrac{x : \delta \vdash^{\delta} \mathsf{let}\ !y = x\ \mathsf{in}\ !(yy)}{\vdash^{\delta} \lambda x.\mathsf{let}\ !y = x\ \mathsf{in}\ !(yy)}}$$

On the other hand the term of Figure 4.3 is not well-formed.

The following proposition will be useful to analyze the depth of occurrences during reduction.

Proposition 4.3.3. *If $\Gamma \vdash^{\delta} M$ and x occurs free in M then $x : \delta'$ belongs to Γ and all occurrences of x in $!^{\delta}M$ are at depth δ'.*

Proof. By induction on the inference rules. We only present the case prom. We have

$$\frac{\Gamma \vdash^{\delta+1} M}{\Gamma \vdash^{\delta} !M}$$

Suppose x occurs free in $!M$. By induction $x : \delta' \in \Gamma$ and all occurrences of x are at depth δ' in $!^{\delta+1}M$. It follows that all occurrences of x are at depth δ' in $!^{\delta}(!M)$. □

The depth system satisfies a subject reduction property. First, we need to establish the following lemmas.

Lemma 4.3.4 (Weakening)**.** *If $\Gamma \vdash^{\delta} M$ then $\Gamma, \Gamma' \vdash^{\delta} M$.*

Proof. By induction on the inference rules. □

Lemma 4.3.5 (Substitution)**.** *If $\Gamma, x : \delta' \vdash^{\delta} M$ and $\Gamma \vdash^{\delta'} N$ then:*

1. $\Gamma \vdash^{\delta} M[N/x]$,
2. $d(!^{\delta}M[N/x]) \leq max(d(!^{\delta}M), d(!^{\delta'}N))$.

Proof. Item 1 can be proved by induction on the inference rules. Let us consider item 2. By item 1, we know that all occurrences of x in $!^{\delta}M$ are at depth δ'. By definition of depth, it follows that $\delta' \geq \delta$ and the occurrences of x in M are at depth $(\delta' - \delta)$. An occurrence in $!^{\delta'}N$ at depth $\delta' + \delta''$ will generate an occurrence in $!^{\delta}M[N/x]$ at the same depth $\delta + (\delta' - \delta) + \delta''$. □

Proposition 4.3.6 (Subject reduction)**.** *If* $\Gamma \vdash^{\delta} M$ *and* $M \longrightarrow N$ *then* $\Gamma \vdash^{\delta} N$ *and* $d(M) \geq d(N)$.

Proof. By case analysis on the reduction rules, Lemma 4.3.4 (weakening) and Lemma 4.3.5 (substitution). □

4.4 Termination in elementary time

In this section, we prove that well-formed terms in $\lambda^{!}_{EAD}$ terminate in elementary time under an arbitrary reduction strategy. To this end, we define a measure on terms based on the number of occurrences at each depth.

Definition 4.4.1 (Measure)**.** Given a term M and $0 \leq i \leq d(M)$, assume $s_i(M) \geq 2$. We define $\mu^i_n(M)$ for $n \geq i \geq 0$ as follows:

$$\mu^i_n(M) = (s_n(M), \ldots, s_{i+1}(M), s_i(M))$$

We write $\mu_n(M)$ for $\mu^0_n(M)$.

We order vectors of $n+1$ natural number with the well-founded lexicographic order $>$ from right to left. Therefore shallow occurrences have more weight than deeper occurrences. For example we have:

$$\begin{array}{rcl} (5,4,3) & > & (10,6,2) \\ (10,6,2) & > & (8,5,2) \end{array}$$

We derive a termination property by observing that the measure strictly decreases during reduction.

Proposition 4.4.2 (Termination)**.** *If* M *is well-formed,* $M \longrightarrow M'$ *and* $n \geq d(M)$ *then* $\mu_n(M) > \mu_n(M')$.

Proof. We proceed by case analysis on the reduction rules. We assume the redexes occur at depth i in E and for convenience we refer to the reduction on syntax trees (4.1) and (4.2).

(β) $E[(\lambda x.M)N] \longrightarrow M' = E[M[N/x]]$
The restrictions on the formation of terms require that x occurs at most once in M at depth i. We see that $s_i(M) - 3 \geq s_i(M')$ because we remove the nodes for application and λ-abstraction and either N disappears or the occurrence of the variable x in M disappears (both being at the same depth as the redex). Clearly $s_j(M) = s_j(M')$ if $j \neq i$, hence

$$\mu_n(M') \leq (s_n(M), \ldots, s_{i+1}(M), s_i(M) - 3, \mu_{i-1}M) \qquad (4.7)$$

and $\mu_n(M) > \mu_n(M')$.

(!) $E[\mathsf{let}\ !x = !N\ \mathsf{in}\ M] \longrightarrow E[M[N/x]]$
The restrictions on the formation of terms require that x may only occur in M at depth $i+1$. We have that $s_i(M) \leq s_i(M) - 2$ because at least a $\mathsf{let}\,!$ node and a $!$ node disappear. Clearly $s_j(M) = s_j(M')$ if $j < i$. The number of occurrences of x in M is bounded by $k = s_{i+1}(M) \geq 2$. Thus if $j > i$ we have $s_j(M') \leq k \cdot s_j(M)$. For $0 \leq i \leq n$, let us note

$$\mu_n^i(M) \cdot k = (s_n(M) \cdot k, s_{n-1}(M) \cdot k, \ldots, s_i(M) \cdot k)$$

We have

$$\mu_n(M') \leq (\mu_n^{i+1}(M) \cdot k, s_i(M) - 2, \mu_{i-1}(M)) \tag{4.8}$$

and we conclude $\mu_n(M) > \mu_n(M')$.

□

We now want to show that termination is in elementary time. We refer to Proposition 4.2.2 for a definition of elementary time functions.

Definition 4.4.3 (Tower functions)**.** We define a family of tower functions $t_\alpha(x_1, \ldots, x_n)$ by induction on n where we assume $\alpha \geq 1$ and $x_i \geq 2$:

$$\begin{array}{rcll} t_\alpha() & = & 0 & \\ t_\alpha(x_1, x_2, \ldots, x_n) & = & (\alpha \cdot x_1)^{2^{t_\alpha(x_2, \ldots, x_n)}} & n \geq 1 \end{array}$$

In the following, we write $\mathbf{x}$ for any vector of n natural numbers $(x_1, x_2, \ldots, x_n)$. Also, for any $m \geq 1$ we write $t_\alpha(y_1, y_2, \ldots, y_m, \mathbf{x})$ for $t_\alpha(y_1, y_2, \ldots, y_m, x_1, x_2, \ldots, x_n)$.

We need to prove the following crucial lemma.

Lemma 4.4.4 (Shift)**.** *Assuming $\alpha \geq 1$ and $\beta \geq 2$, the following property holds for the tower functions with $x, \mathbf{x}$ ranging over numbers greater or equal to 2:*

$$t_\alpha(\beta \cdot x, x', \mathbf{x}) \leq t_\alpha(x, \beta \cdot x', \mathbf{x})$$

Some intermediate arithmetic properties are needed to prove it. We start by remarking some basic inequalities.

Lemma 4.4.5. *The following properties hold on natural numbers.*

1. $\forall\, x \geq 2, y \geq 0\ \ (y+1) \leq x^y$
2. $\forall\, x \geq 2, y \geq 0\ \ (x \cdot y) \leq x^y$
3. $\forall\, x \geq 2, y, z \geq 0\ \ (x \cdot y)^z \leq x^{(y \cdot z)}$
4. $\forall\, x \geq 2, y \geq 0, z \geq 1\ \ x^z \cdot y \leq x^{(y \cdot z)}$
5. *If $x \geq y \geq 0$ then $(x-y)^k \leq (x^k - y^k)$*

Proof. See Appendix A. □

We also need the following property.

Lemma 4.4.6 (Pre-shift)**.** *Assuming $\alpha \geq 1$ and $\beta \geq 2$, the following property holds for the tower functions with $x, \mathbf{x}$ ranging over numbers greater or equal than 2:*

$$\beta \cdot t_\alpha(x, \mathbf{x}) \leq t_\alpha(\beta \cdot x, \mathbf{x})$$

Proof. This follows from $\beta \leq \beta^{2^{t_\alpha(\mathbf{x})}}$. □

Then we can derive the proof of the shift lemma as follows.

Proof of Lemma 4.4.4. Let $k = t_\alpha(x', \mathbf{x}) \geq 2$. Then

$$\begin{aligned} t_\alpha(\beta \cdot x, x', \mathbf{x}) = \beta \cdot (\alpha \cdot x)^{2^k} &\leq (\alpha \cdot x)^{\beta \cdot 2^k} && \text{(by Lemma 4.4.5-3)} \\ &\leq (\alpha \cdot x)^{(\beta \cdot 2)^k} \\ &\leq (\alpha \cdot x)^{2^{(\beta \cdot k)}} && \text{(by Lemma 4.4.5-3)} \end{aligned}$$

and by Lemma 4.4.6, $\beta \cdot t_\alpha(x', \mathbf{x}) \leq t_\alpha(\beta \cdot x', \mathbf{x})$. Hence

$$(\alpha \cdot x)^{2^{(\beta \cdot k)}} \leq (\alpha \cdot x)^{2^{t_\alpha(\beta \cdot x', \mathbf{x})}} = t_\alpha(x, \beta \cdot x', \mathbf{x})$$

□

Now, by a closer look at the shape of the lexicographic ordering during reduction, we are able to compose the decreasing measure with a tower function.

Theorem 4.4.7 (Elementary bound)**.** *Let M be a well-formed term with $\alpha = d(M)$ and let t_α denote the tower function with $\alpha + 1$ arguments. If $M \longrightarrow M'$ then $t_\alpha(\mu_\alpha(M)) > t_\alpha(\mu_\alpha(M'))$.*

Proof. By case analysis on the reduction rules. The case of the rule (β) is trivial since we see that exactly one component of the measure is strictly decreasing in Equation (4.7). We illustrate the crucial case of the duplicating rule (!) where

$$E[\mathsf{let}\ !x = !N\ \mathsf{in}\ M] \longrightarrow E[M[N/x]]$$

For simplicity assume $\alpha = 2$ and $E = [\cdot]$ (we provide a complete proof in Appendix A).

Let

$$\mu_2(\mathsf{let}\ !x = !N\ \mathsf{in}\ M) = (x, y, z)$$

such that

$$\begin{aligned} x &= s_2(\mathsf{let}\ !x = !N\ \mathsf{in}\ M) \\ y &= s_1(\mathsf{let}\ !x = !N\ \mathsf{in}\ M) \\ z &= s_0(\mathsf{let}\ !x = !N\ \mathsf{in}\ M) \end{aligned}$$

We want to show

$$t_2(\mu_2(M[N/x])) < t_2(\mu_2(\mathsf{let}\ !x = !N\ \mathsf{in}\ M))$$

We have

$$\begin{array}{rcll} t_2(\mu_2(M[N/x])) & \leq & t_2(x \cdot y, y \cdot y, z-2) & \text{by Inequality (4.8)} \\ & \leq & t_2(x, y^3, z-2) & \text{by Lemma 4.4.4} \end{array}$$

Hence we are left to show

$$t_2(y^3, z-2) < t_2(y, z) \quad \textit{i.e.} \quad (2y^3)^{2^{2(z-2)}} < (2y)^{2^{2z}}$$

We have

$$(2y^3)^{2^{2(z-2)}} \quad \leq \quad (2y)^{3 \cdot 2^{2(z-2)}}$$

Thus we need to show

$$3 \cdot 2^{2(z-2)} < 2^{2z}$$

Dividing by 2^{2z} we get

$$3 \cdot 2^{-4} < 1$$

which is obviously true. Hence $t_2(\mu_2(M')) < t_2(\mu_2(M))$. □

This shows that the number of reduction steps of a term M is bound by an elementary time function where the height of the tower depends on $d(M)$. The following corollary lifts the result to *elementary time*.

Corollary 4.4.8 (Elementary time)**.** *The reduction of a well-formed term M can be computed by a Turing machine in time bounded by a tower of exponentials whose height only depends on $d(M)$.*

Proof. It suffices to remark that each reduction step $M_i \longrightarrow M_{i+1}$ can be performed in time quadratic in the size of M_i, which is bounded by $t(s(M), d(M))$. Indeed, in the worst situation, the reduction rule (!) is substituting an argument of size bounded by $s(M_i)$ for at most $s(M_i)$ occurrences of a free variable. □

Chapter 5

An elementary concurrent λ-calculus

In this chapter, we present an *elementary concurrent λ-calculus*. The main result is that we characterize a class of *well-formed* concurrent programs that terminate in elementary time under a call-by-value reduction strategy. Our developments can be summarized in the following items:

- We contribute an analysis of the impact of side effects on the depth of occurrences that leads to the design of an elementary affine depth system $\lambda^{!\|}_{EAD}$ that stratifies regions by depth levels. Termination in elementary time then strongly relies on this depth system.
- We show that the depth system $\lambda^{!\|}_{EAD}$ captures concurrent programs that iterate side effects over inductive data structures. Moreover, we provide an elementary affine type system $\lambda^{!\|}_{EAT}$ that also captures these programming examples.
- The interesting point is that the depth system $\lambda^{!\|}_{EAD}$ does not rely on the stratification of regions by *effects*, as seen in Chapter 2 and 3, but on the stratification of regions by *depth levels* that appears to allow for some more flexibility.

Outline The chapter is organized as follows. The syntax and reduction of the elementary λ-calculus is that of $\lambda^{!\|}$ which is recalled in Section 5.1 with the notion of depth. In Section 5.2, we provide an analysis of the impact of side effects on the depth of the occurrences which leads us to revise the notion of depth and design the system $\lambda^{!\|}_{EAD}$ that stratifies regions by depth levels. We show in Section 5.3 that it guarantees termination of programs in elementary time under a call-by-value evaluation strategy. Surprisingly, the combinatorial

proof requires very few adaptions from the functional case. In Section 5.4, we refine the depth system into a second order (polymorphic) elementary affine type system $\lambda^{!\|}_{EAT}$ and show that the resulting system enjoys subject reduction and progress (besides termination in elementary time). Finally, in Section 5.5 we discuss the expressivity of the elementary affine type system. We first check that the usual encoding of elementary time functions goes through. Then, and more interestingly, we provide examples of iterative concurrent programs with side effects. Also, we compare the expressive power offered by the stratification of regions depth levels with respect to stratification by effects.

The contributions can be summarized by the diagram of Figure 5.1.

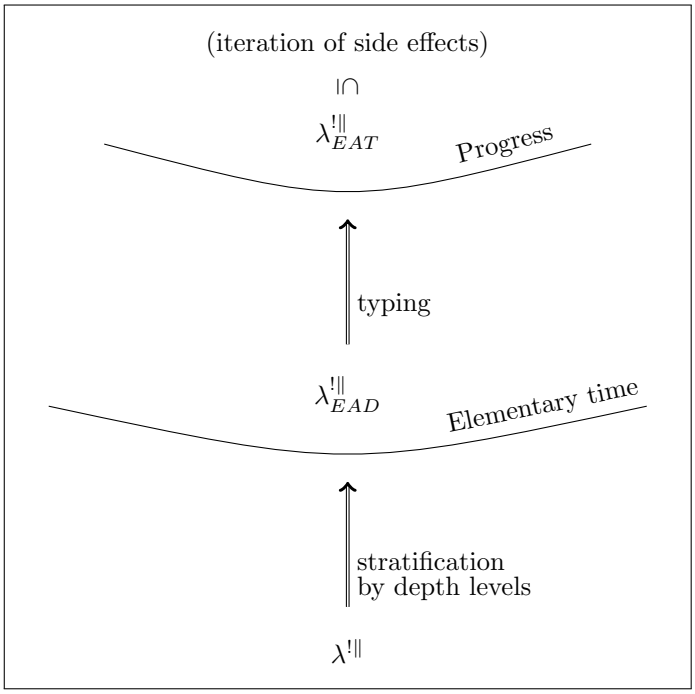


Figure 5.1: From $\lambda^{!\|}$ to concurrent iterations in $\lambda^{!\|}_{EAT}$

5.1 Syntax, reduction and depth

The syntax and reduction of the elementary concurrent λ-calculus is exactly that of $\lambda^{!\|}$ given in Figure 3.6 and Figure 3.7. We recall it in Figure 5.2.

Abstract syntax trees extend straightforwardly to concurrent programs as exemplified in Figure 5.3.

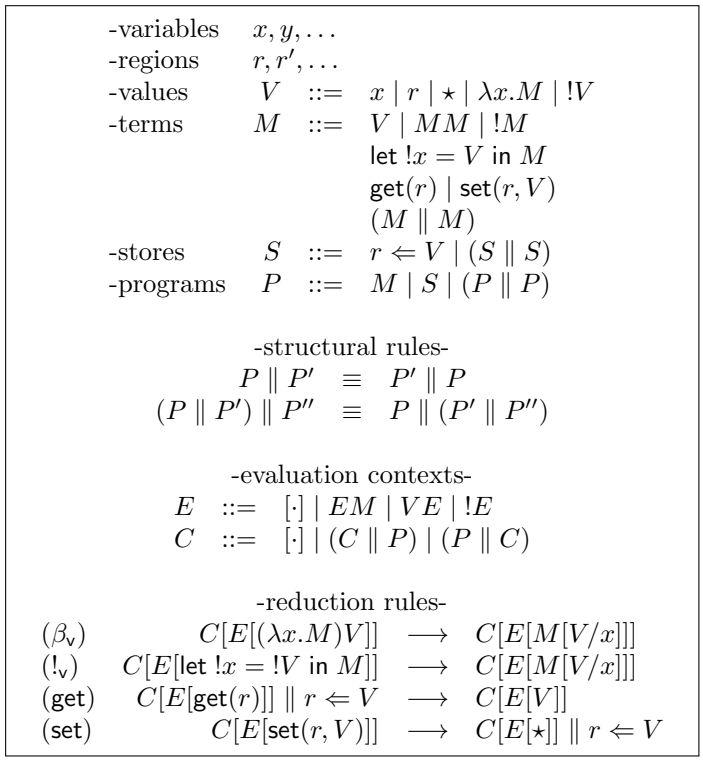


Figure 5.2: Syntax and reduction of $\lambda^{!\parallel}$

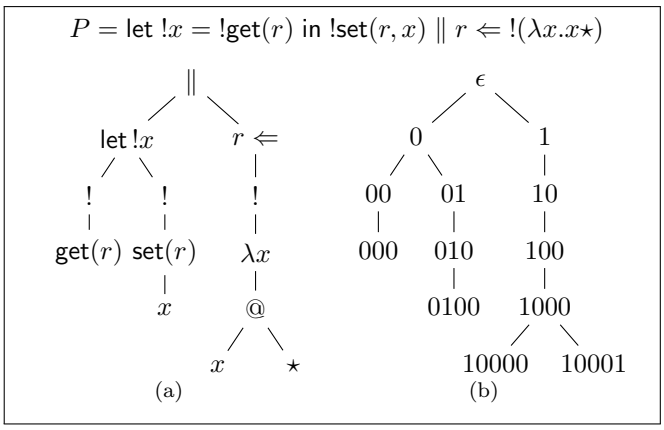


Figure 5.3: Syntax tree and addresses of P

We recall the definition of depth.

Definition 5.1.1 (Depth)**.** The *depth* $d(w)$ of an occurrence w in a term M is the number of '!' labels that the path leading to the end node crosses. The depth $d(M)$ of the term M is the maximum depth of its occurrences.

5.2 An elementary affine depth system

In this section, we present the elementary affine depth system $\lambda^{!\|}_{EAD}$. As a first step, we analyze in Section 5.2.1 the impact of side effects on the depth of occurrences, which leads us to revise the notion of depth. Then, in Section 5.2.2 we present the inference rules of $\lambda^{!\|}_{EAD}$. Well-formed programs in the depth system preserve the depth of their occurrences by reduction, even though they transit through the store. Finally, in Section 5.2.3 we derive the subject reduction property.

5.2.1 Revised depth

We observe that side effects may change the depth of occurrences, thus breaking the stratified nature of programs. In the reduction sequence

$$\begin{aligned}(\lambda x.\mathsf{set}(r,x) \parallel !\mathsf{get}(r))!V &\longrightarrow^* !\mathsf{get}(r) \parallel r \Leftarrow !V \\ &\longrightarrow !!V\end{aligned} \tag{5.1}$$

the occurrence V moves from depth 1 to depth 2 during the last reduction step, because the read occurs at depth 1 while the write occurs at depth 0.

We propose to introduce *region contexts* in order to constrain the depth at which side effects occur. A region context

$$R = r_1 : \delta_1, \ldots, r_n : \delta_n$$

associates a natural number δ_i to each region r_i in a finite set of regions $\{r_1, \ldots, r_n\}$ that we write $dom(R)$. We write $R(r_i)$ for δ_i. The idea is then to extend the functional depth system $\lambda^{!}_{EAD}$ so that $\mathsf{get}(r_i)$ and $\mathsf{set}(r_i, M)$ may only occur at the fixed depth δ_i, thus rejecting (5.1).

We also remark that, since stores are global, they always occur at depth 0 and assigning a term to a region breaks stratification whenever $\delta_i > 0$. Indeed, in the reduction

$$!\mathsf{set}(r, V) \longrightarrow !\star \parallel r \Leftarrow V \tag{5.2}$$

where $R(r)$ should be 1, the occurrence V moves from depth 1 to depth 0. Therefore, we propose to revise the definition of depth as follows.

Definition 5.2.1 (Revised depth)**.** Let P be a program and R a region context where $dom(R)$ contains all the regions of P. The *revised depth* $rd(w)$ of an occurrence w of P is the number of '!' labels that the path leading to the end node crosses, plus $R(r)$ if the path crosses a store label '$r \Leftarrow$'. The revised depth $rd(P)$ of a program P is the maximum revised depth of its occurrences.

Note that, on functional terms, the revised depth exactly corresponds to the standard depth notion given in Definition 5.1.1. But in reduction (5.2), the occurrence V now stays at revised depth 1, and in Figure 5.3 we get

$$\begin{aligned} &rd(\epsilon) = rd(0) = rd(00) = rd(01) = rd(1) = 0 \\ &rd(000) = rd(010) = rd(0100) = 1 \\ &rd(10) = R(r) \\ &rd(100) = rd(1000) = rd(10000) = rd(10001) = R(r) + 1 \end{aligned}$$

In the sequel, we will exclusively use the revised definition of depth, we will simply say 'depth' for 'revised depth' and write $d(w)$ for $rd(w)$.

5.2.2 Rules

A depth judgment has the shape

$$R; \Gamma \vdash^{\delta} P$$

and it should entail the following:

- if $x : \delta' \in \Gamma$ and x occurs free in P then all free occurrences of x appear at depth δ' in $\dagger^{\delta} P$,
- if $r : \delta' \in R$ then $\mathsf{get}(r)/\mathsf{set}(r)$ may only occur at depth δ' in $\dagger^{\delta} P$.

The inference rules are given in Figure 5.4. The key rules are get and set where we require $R(r) = \delta$ so that side effects preserve the depth of occurrences. Also, since stores are global, the rule store gives depth 0 to store assignments whereas the stored value has depth $R(r)$. This reflects the revised notion of depth where one has to count $R(r)$ if an occurrence appears in a store assignment.

Definition 5.2.2 (Well-formedness)**.** A program P is *well-formed* if for some R, Γ, δ a judgment $R; \Gamma \vdash^{\delta} P$ can be derived.

Example 5.2.3. The program of Figure 5.3 is well-formed with the following derivation where $R(r) = 1$:

$$\dfrac{\dfrac{\dfrac{}{R; \Gamma \vdash^{1} \mathsf{get}(r)} \quad \dfrac{\dfrac{}{R; \Gamma, x : 1 \vdash^{1} x}}{R; \Gamma, x : 1 \vdash^{1} \mathsf{set}(r, x)}}{R; \Gamma \vdash^{0} !\mathsf{get}(r) \quad R; \Gamma, x : 1 \vdash^{0} !\mathsf{set}(r, x)} \quad \dfrac{\vdots}{R; \Gamma \vdash^{0} r \Leftarrow !(\lambda x.x\star)}}{R; \Gamma \vdash^{0} \mathsf{let}\ !x = !\mathsf{get}(r)\ \mathsf{in}\ !\mathsf{set}(r, x)}}{R; \Gamma \vdash^{0} \mathsf{let}\ !x = !\mathsf{get}(r)\ \mathsf{in}\ !\mathsf{set}(r, x) \parallel r \Leftarrow !(\lambda x.x\star)}$$

$$\mathsf{var}\ \frac{}{R;\Gamma, x:\delta \vdash^{\delta} x} \qquad \mathsf{reg}\ \frac{}{R;\Gamma \vdash^{\delta} r} \qquad \mathsf{unit}\ \frac{}{R;\Gamma \vdash^{\delta} \star}$$

$$\mathsf{lam}\ \frac{\mathsf{FO}(x,M) \leq 1 \qquad R;\Gamma, x:\delta \vdash^{\delta} M}{R;\Gamma \vdash^{\delta} \lambda x.M} \qquad \mathsf{app}\ \frac{R;\Gamma \vdash^{\delta} M \qquad R;\Gamma \vdash^{\delta} N}{R;\Gamma \vdash^{\delta} MN}$$

$$\mathsf{prom}\ \frac{R;\Gamma \vdash^{\delta+1} M}{R;\Gamma \vdash^{\delta} !M} \qquad \mathsf{elim}\ \frac{R;\Gamma \vdash^{\delta} V \qquad R;\Gamma, x:(\delta+1) \vdash^{\delta} M}{R;\Gamma \vdash^{\delta} \mathsf{let}\ !x = V\ \mathsf{in}\ M}$$

$$\mathsf{get}\ \frac{}{R, r:\delta;\Gamma \vdash^{\delta} \mathsf{get}(r)} \qquad \mathsf{set}\ \frac{R, r:\delta;\Gamma \vdash^{\delta} V}{R, r:\delta;\Gamma \vdash^{\delta} \mathsf{set}(r,V)}$$

$$\mathsf{store}\ \frac{R, r:\delta;\Gamma \vdash^{\delta} V}{R, r:\delta;\Gamma \vdash^{0} r \Leftarrow V} \qquad \mathsf{par}\ \frac{i = 1,2 \qquad R;\Gamma \vdash^{\delta} P_i}{R;\Gamma \vdash^{\delta} (P_1 \parallel P_2)}$$

Figure 5.4: The elementary affine depth system $\lambda^{!\|}_{EAD}$

We see that the troublesome program (5.1) considered in the introduction of this section is not well-formed since the read and write operation do not occur at the same depth. On the other hand, program (5.2) is well-formed provided that $R(r) = 1$.

5.2.3 Properties

We derive a property of subject reduction in a way similar to the functional case.

Proposition 5.2.4. *If $R;\Gamma \vdash^{\delta} P$ then:*

- *if $x : \delta' \in \Gamma$ and x occurs free in P then all occurrences of x appear at depth δ' in $\dagger^{\delta} P$,*
- *if $r : \delta' \in R$ then $\mathsf{get}(r)/\mathsf{set}(r)$ may only occur at depth δ' in $\dagger^{\delta} P$.*

Proof. By induction on the inference rules. □

Lemma 5.2.5 (Weakening)**.** *If $R;\Gamma \vdash^{\delta} P$ then $R, R';\Gamma, \Gamma' \vdash^{\delta} P$.*

Proof. By induction on the inference rules. □

Lemma 5.2.6 (Substitution)**.** *If $R;\Gamma, x:\delta' \vdash^{\delta} M$ and $R;\Gamma \vdash^{\delta'} V$ then:*

1. *$R;\Gamma \vdash^{\delta} M[V/x]$,*
2. *$d(!^{\delta} M[V/x]) \leq max(d(!^{\delta} M)(!^{\delta'} V))$.*

Proof. Item 1 can be proved by induction on the typing rules and Item 2 can be shown as in the functional case. □

Proposition 5.2.7 (Subject reduction)**.** *If* $R;\Gamma \vdash^0 P$ *and* $P \longrightarrow P'$ *then* $R;\Gamma \vdash^0 P'$ *and* $d(P') \leq d(P)$.

Proof. By case analysis on the reduction rules and the above lemmas. We only highlight the rules with side effects.

(set) $C[E[\mathsf{set}(r,V)]] \longrightarrow C[E[\star]] \parallel r \Leftarrow V$
We have $R;\Gamma \vdash^0 C[E[\mathsf{set}(r,V)]]$ from which we derive

$$\frac{R;\Gamma \vdash^\delta V}{R;\Gamma \vdash^\delta \mathsf{set}(r,V)}$$

for some $\delta \geq 0$ with $r : \delta \in R$. Hence we can derive

$$\frac{R;\Gamma \vdash^\delta V}{R;\Gamma \vdash^0 r \Leftarrow V}$$

Moreover, we have $R;\Gamma \vdash^\delta \star$ thus we can derive $R;\Gamma \vdash^0 C[E[\star]]$. Applying the rule par we finally get

$$R;\Gamma \vdash^0 C[E[\star]] \parallel r \Leftarrow V$$

Concerning the depth bound we clearly have $d(C[E[\star]] \parallel r \Leftarrow V) = d(C[E[\mathsf{set}(r,V)]])$.

(get) $C[E[\mathsf{get}(r)] \parallel r \Leftarrow V \longrightarrow C[E[V]]$
We have $R;\Gamma \vdash^0 C[E[\mathsf{get}(r)]] \parallel r \Leftarrow V$ from which we derive

$$\frac{}{R;\Gamma \vdash^\delta \mathsf{get}(r)}$$

and

$$\frac{R;\Gamma \vdash^\delta V}{R;\Gamma \vdash^0 r \Leftarrow V}$$

for some $\delta \geq 0$ with $r : \delta \in R$. Hence we can derive

$$R;\Gamma \vdash^0 C[E[V]]$$

Concerning the depth bound we clearly have $d(E[V]) = d(E[\mathsf{get}(r)] \mid r \Leftarrow V)$.

□

5.3 Termination in elementary time

In this section, we prove that programs well-formed in $\lambda^{!\parallel}_{EAD}$ terminate in elementary time under a call-by-value reduction strategy. The combinatorial analysis requires very few adaptations from the functional case and in fact, the main

contribution has been to carefully design the depth system so that it takes side effects into account.

The measure on the occurrences of terms given in Definition 4.4.1 extends trivially to programs of $\lambda^{!\|}_{EAD}$. We recall it here.

Definition 5.3.1 (Measure)**.** Given a program P and $0 \leq i \leq d(P)$, assume $s_i(P) \geq 2$. We define $\mu^i_n(P)$ for $n \geq i \geq 0$ as follows:

$$\mu^i_n(P) = (s_n(P), \ldots, s_{i+1}(P), s_i(P))$$

We write $\mu_n(P)$ for $\mu^0_n(P)$.

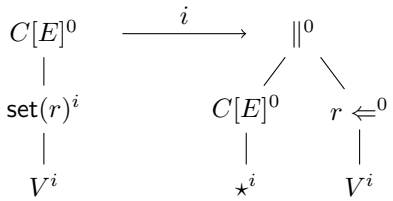
We also recall that vectors of $n+1$ natural are ordered by the well-founded lexicographic order $>$ from right to left (*e.g.* $(10, 6, 2) > (12, 5, 2)$).

In order to simplify the combinatorial analysis of programs, we assume that the occurrences labelled with '$\|$' and '$r \leftarrow$' do not count and that '$\mathsf{set}(r)$' counts for two occurrences. In this way the measure strictly decreases on the rule (set) and we can derive the termination property.

Proposition 5.3.2 (Termination)**.** *If P is well-formed, $P \longrightarrow P'$ and $n \geq d(P)$ then $\mu_n(P) > \mu_n(P')$.*

Proof. By a case analysis on the reduction rules. We only highlight the two cases with side effects.

(set) $C[E[\mathsf{set}(r, V)]] \longrightarrow C[E[\star]] \parallel r \Leftarrow V$
Suppose $R; \Gamma \vdash^0 C[E[\mathsf{set}(r, V)]]$ with $R(r) = i$. By Proposition 5.2.4 the redex $\mathsf{set}(r, V)$ occurs at depth i in $C[E]$ and by Proposition 5.2.7 we have $R; \Gamma \vdash^0 r \Leftarrow V$, therefore V must occur at depth i in the store. Let us observe the reduction on syntax trees where occurrences are labelled with their depth:



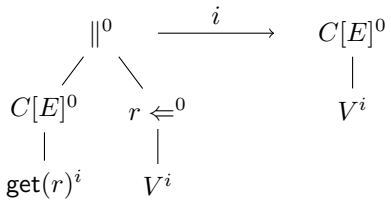
We remark that the occurrence $\mathsf{set}(r)$ that counts for two disappears while only one new occurrence $\star$ appears ($\|$ and $r \Leftarrow$ do not count). Thus:

$$s_i(C[E[\star]] \parallel r \Leftarrow V) = s_i(C[E[\mathsf{set}(r, V)]]) - 1$$

Since the numbers of occurrences at other depths stay unchanged, we conclude

$$\mu_n(C[E[\mathsf{set}(r, V)]]) > \mu_n(C[E[\star]] \parallel r \Leftarrow V)$$

(get) $C[E[\mathsf{get}(r)]] \parallel r \Leftarrow V \longrightarrow C[E[V]]$
Suppose $R;\Gamma \vdash^0 C[E[\mathsf{get}(r)]] \parallel r \Leftarrow V$ with $R(r) = i$. By Proposition 5.2.4 the redex $\mathsf{get}(r)$ occurs at depth i and by definition of depth V also occurs at depth i in the store. By Proposition 5.2.7 we have $R;\Gamma \vdash^0 C[E[V]]$ therefore V stays at depth i. Let us observe the reduction on syntax trees:



We remark that exactly one occurrence $\mathsf{get}(r)$ disappears ($\parallel$ and $r \Leftarrow$ do not count). Thus:

$$s_i(C[E[V]]) = s_i(C[E[\mathsf{get}(r)]] \parallel r \Leftarrow V) - 1$$

Since the number of occurrences at other depths stay unchanged, we conclude

$$\mu_n(C[E[\mathsf{get}(r)]] \parallel r \Leftarrow V) > \mu_n(C[E[V]])$$

□

We are then able to state an elementary bound.

Theorem 5.3.3 (Elementary bound)**.** *Let P be a well-formed program with $\alpha = d(P)$ and let t_α denote the tower function with $\alpha + 1$ arguments. Then if $P \longrightarrow P'$ then $t_\alpha(\mu_\alpha(P)) > t_\alpha(\mu_\alpha(P'))$.*

Proof. The proof requires very few adaptions from the functional one. Indeed the rules with side effects (get) and (set) do not duplicate anything and we remark in the above proof of termination that exactly one component of the measure is decreasing while other components stay unchanged. Therefore it is trivial to show that the value of the tower function is strictly decreasing. □

Corollary 5.3.4. *The call-by-value reduction of a well-formed program P of depth d can be computed by a Turing machine in time bounded by a tower of exponentials of whose height only depends on d.*

5.4 An elementary affine type system

The depth system $\lambda^{!\parallel}_{EAD}$ entails termination in elementary time but does *not* guarantee that programs 'do not go wrong'. In particular, the well-formed program

$$\mathsf{let}\ !y = (\lambda x.x)\ \mathsf{in}\ !(yy) \tag{5.3}$$

which is not a value cannot reduce further. In this section, we propose a solution to this problem by introducing a polymorphic elementary affine type system $\lambda^{!\|}_{EAT}$ which is simple decoration of $\lambda^{!\|}_{EAD}$ with types. Then, we derive a progress proposition which guarantees that well-typed programs cannot deadlock (except when trying to read an empty region).

We define the syntax of types and contexts in Figure 5.5. The only difference

-type variables	$t, t', \ldots$		
-types	α	$::=$	$\mathbf{B} \mid A$
-res. types	A	$::=$	$t \mid \mathsf{Unit} \mid A \multimap \alpha \mid !A \mid \forall t.A \mid \mathsf{Reg}_r A$
-var. contexts	Γ	$::=$	$x_1 : (\delta_1, A_1), \ldots, x_n : (\delta_n, A_n)$
-reg. contexts	R	$::=$	$r_1 : (\delta_1, A_1), \ldots, r_n : (\delta_n, A_n)$

Figure 5.5: Types and contexts of $\lambda^{!\|}_{EAT}$

with types and contexts of $\lambda^{!\|}$ (see Figure 3.8) is that we added the polymorphic type $\forall t.A$ and that contexts use natural numbers δ_i instead of usages. In contexts, natural numbers play the same role as in the depth system. Writing $x : (\delta, A)$ means that the variable x ranges on terms of type A and may occur at depth δ. Writing $r : (\delta, A)$ means that the region r contain terms of type A and that $\mathsf{get}(r)$ and $\mathsf{set}(r, V)$ may only occur at depth δ.

As usual, types depend on region names and we have to be careful in stating in Figure 5.6 when a type is well-formed in a region context.

$$\frac{}{R \downarrow t} \qquad \frac{}{R \downarrow \mathsf{Unit}} \qquad \frac{}{R \downarrow \mathbf{B}}$$

$$\frac{R \downarrow A}{R \downarrow !A} \qquad \frac{R \downarrow A \quad R \downarrow \alpha}{R \downarrow (A \multimap \alpha)} \qquad \frac{r : (\delta, A) \in R}{R \downarrow \mathsf{Reg}_r A} \qquad \frac{R \downarrow A \quad t \notin R}{R \downarrow \forall t.A}$$

$$\frac{\forall r : (\delta, A) \in R \quad R \downarrow A}{R \vdash} \qquad \frac{R \vdash \quad R \downarrow \alpha}{R \vdash \alpha} \qquad \frac{\forall x : (\delta, A) \in \Gamma \quad R \vdash A}{R \vdash \Gamma}$$

Figure 5.6: Formation of types and contexts

We notice the following substitution property on types.

Proposition 5.4.1. *If $R \vdash \forall t.A$ and $R \vdash B$ then $R \vdash A[B/t]$.*

Proof. By induction on A. □

A typing judgment takes the form:

$$R; \Gamma \vdash^{\delta} P : \alpha$$

It attributes a type α to the program P occurring at depth δ, according to region context R and variable context Γ.

Figure 5.7 introduces the rules of $\lambda^{!\|}_{EAD}$. The skeleton of the rules is the el-

$$\mathsf{var}\,\frac{R \vdash \Gamma \qquad x : (\delta, A) \in \Gamma}{R; \Gamma \vdash^{\delta} x : A} \qquad \mathsf{unit}\,\frac{R \vdash \Gamma}{R; \Gamma \vdash^{\delta} \star : \mathsf{Unit}}$$

$$\mathsf{reg}\,\frac{R \vdash \Gamma \qquad r : (\delta', A) \in R}{R; \Gamma \vdash^{\delta} r : \mathsf{Reg}_r A}$$

$$\mathsf{lam}\,\frac{\mathsf{FO}(x, M) \leq 1 \qquad R; \Gamma, x : (\delta, A) \vdash^{\delta} M : \alpha}{R; \Gamma \vdash^{\delta} \lambda x.M : A \multimap \alpha}$$

$$\mathsf{app}\,\frac{R; \Gamma \vdash^{\delta} M : A \multimap \alpha \qquad R; \Gamma \vdash^{\delta} N : A}{R; \Gamma \vdash^{\delta} MN : \alpha}$$

$$\mathsf{prom}\,\frac{R; \Gamma \vdash^{\delta+1} M : A}{R; \Gamma \vdash^{\delta} !M : !A}$$

$$\mathsf{elim}\,\frac{R; \Gamma \vdash^{\delta} V : !A \qquad R; \Gamma, x : (\delta + 1, A) \vdash^{\delta} M : B}{R; \Gamma \vdash^{\delta} \mathsf{let}\ !x = V\ \mathsf{in}\ M : B}$$

$$\mathsf{forall}\,\frac{R; \Gamma \vdash^{\delta} M : A \qquad t \notin (R; \Gamma)}{R; \Gamma \vdash^{\delta} M : \forall t.A} \qquad \mathsf{inst}\,\frac{R; \Gamma \vdash^{\delta} M : \forall t.A \qquad R \vdash B}{R; \Gamma \vdash^{\delta} M : A[B/t]}$$

$$\mathsf{get}\,\frac{r : (\delta, A) \in R \qquad R \vdash \Gamma}{R; \Gamma \vdash^{\delta} \mathsf{get}(r) : A}$$

$$\mathsf{set}\,\frac{r : (\delta, A) \in R \qquad R; \Gamma \vdash^{\delta} V : A}{R; \Gamma \vdash^{\delta} \mathsf{set}(r, V) : \mathsf{Unit}} \qquad \mathsf{store}\,\frac{r : (\delta, A) \in R \qquad R; \Gamma \vdash^{\delta} V : A}{R; \Gamma \vdash^{0} r \Leftarrow V : \mathbf{B}}$$

$$\mathsf{par}_1\,\frac{R; \Gamma \vdash^{\delta} P : \alpha \qquad R; \Gamma \vdash^{\delta} S : \mathbf{B}}{R; \Gamma \vdash^{\delta} (P \parallel S) : \alpha} \qquad \mathsf{par}_2\,\frac{P_i \text{ not a store } i = 1, 2 \qquad R; \Gamma \vdash^{\delta} P_i : \alpha_i}{R; \Gamma \vdash^{\delta} (P_1 \parallel P_2) : \mathbf{B}}$$

Figure 5.7: The elementary affine type system $\lambda^{!\|}_{EAT}$

ementary affine depth system $\lambda^{!\|}_{EAD}$ while the type machinery is that of the affine-intuitionistic type system of $\lambda^{!\|}$ (see Figure 3.10). The only new rules are the polymorphic rules forall and inst.

Definition 5.4.2 (Well-typing)**.** A program P is *well-typed* if for some R, Γ,

δ, α a judgment $R; \Gamma \vdash^{\delta} P : \alpha$ can be derived.

Example 5.4.3. The program of Figure 5.3 is well-typed since the judgment

$$R; - \vdash^{\delta} \mathsf{let}\ !x = !\mathsf{get}(r)\ \mathsf{in}\ !\mathsf{set}(r, x) \parallel r \Leftarrow !(\lambda x.x\star) : !\mathsf{Unit}$$

can be derived with $R = r : (\delta + 1, \forall t.!((\mathsf{Unit} \multimap t) \multimap t))$. However the program (5.3) (page 97) is not well-typed.

Remark 5.4.4. We can easily see that a well-typed program is also well-formed.

The usual weakening and substitution properties can be established.

Lemma 5.4.5 (Weakening)**.** *If $R; \Gamma \vdash^{\delta} P : \alpha$ and $R, R' \vdash \Gamma, \Gamma'$ then $R, R'; \Gamma, \Gamma' \vdash^{\delta} P : \alpha$.*

Proof. By induction on the typing of P. □

Lemma 5.4.6 (Substitution)**.** *If $R; \Gamma, x : (\delta', A) \vdash^{\delta} P : \alpha$ and $R; \Gamma \vdash^{\delta'} V : A$ then $R; \Gamma \vdash^{\delta} P[V/x] : \alpha$.*

Proof. By induction on the typing of P. □

This allows us to derive the subject reduction property.

Proposition 5.4.7 (Subject reduction)**.** *If $R; \Gamma \vdash^{0} P : \alpha$ and $P \longrightarrow P'$ then $R; \Gamma \vdash^{0} P' : \alpha$.*

Proof. We proceed similarly to the proof of subject reduction for the affine-intuitionistic type system (Proposition 3.3.6). We first show that structural equivalence preserves typing. Then by Lemma 5.4.6 we can show that functional redexes preserve typing. There remains to check that redexes with side effects preserve typing which can be easily shown by looking at the proof of subject reduction of the depth system. □

Finally, we establish a progress proposition which shows that any well-typed program reduces to several threads in parallel which are values or deadlocking reads.

Proposition 5.4.8 (Progress)**.** *Suppose P is a closed typable call-by-value program which cannot reduce. Then P is structurally equivalent to a program*

$$M_1 \parallel \cdots \parallel M_m \parallel S \quad m \geq 0$$

where M_i is either a value or can only be decomposed as a term $E[\mathsf{get}(r)]$ such that no value is associated with the region r in the store S.

Proof. Again the proof is similar to that of the affine-intuitionistic system (Proposition 3.3.11). The main ingredient is to observe that values of type $!A$ are of the shape $!V$, so that a well-typed term $\mathsf{let}\ !x = V\ \mathsf{in}\ M$ is guaranteed to reduce. □

5.5 Expressivity

In this section, we evaluate the expressivity of $\lambda^{!\|}_{EAT}$. We first show that every elementary time function is representable (Section 5.5.1). Then we evaluate how one can program in this type system by giving an example iterating program producing side effects over an inductive data structure (Section 5.5.2). Finally we compare the stratification of regions by effects system and the stratification of regions by depth levels (Section 5.5.3).

5.5.1 Completeness

We show that every elementary time function can be represented by a well-typed program, that is $\lambda^{!\|}_{EAT}$ is extensionally complete. The result is adapted from Danos and Joinet's proof [DJ03]. Since it only relies on the functional core of the system, we omit region contexts for simplicity.

The precise notion of representation is spelled out in the following definitions. We denote with $\mathbb{N}$ the set of natural numbers and by strong β-reduction we mean that reduction under binders is allowed.

Definition 5.5.1 (Number representation)**.** Let $\emptyset \vdash^{\delta} M : \mathsf{Nat}$. We say M *represents* $n \in \mathbb{N}$, written $M \Vdash n$, if $M \longrightarrow^{*} \overline{n}$ by strong β-reduction.

Definition 5.5.2 (Function representation)**.** Let $\emptyset \vdash^{\delta} F : (\mathsf{Nat}_1 \multimap \ldots \multimap \mathsf{Nat}_k) \multimap !^{p}\mathsf{Nat}$ where $p \geq 0$ and $f : \mathbb{N}^k \to \mathbb{N}$. We say F *represents* f, written $F \Vdash f$, if for all M_i and $n_i \in \mathbb{N}$ where $1 \leq i \leq k$ such that $\emptyset \vdash^{\delta} M_i : N$ and $M_i \Vdash n_i$ we have $FM_1 \ldots M_k \Vdash f(n_1, \ldots, n_k)$.

Building on the standard concept of Church numeral, Figure 5.8 provides a representation for natural numbers and some arithmetic functions. It is also necessary to use pairs to represent subtraction and bounded summation/product. Their representation with the projection functions are given in Figure 5.9.

We can then derive the following theorem.

Theorem 5.5.3 (Completeness)**.** *Every function which can be computed by a Turing machine in time bounded by an elementary function of height d can be represented by a term of type* $\mathsf{Nat} \multimap !^{d}\mathsf{Nat}$.

Proof. The elementary time functions are characterized as the smallest class of functions containing zero, successor, projection, subtraction, composition, bounded summation and bounded product. We have to check that they are representable in the sense of Definition 5.5.2. The proof which involves tedious syntactic manipulations and is adapted from Danos and Joinet [DJ03] is left to Appendix A. □

Nat	$=$	$\forall t.!(t \multimap t) \multimap !(t \multimap t)$	(type of numerals)
zero	$:$	Nat	(zero)
zero	$=$	$\lambda f.!(\lambda x.x)$	
succ	$:$	$\mathsf{Nat} \multimap \mathsf{Nat}$	(successor)
succ	$=$	$\lambda n.\lambda f.\mathsf{let}\ !f = f\ \mathsf{in}$	
		$\mathsf{let}\ !y = n!f\ \mathsf{in}\ !(\lambda x.f(yx))$	
$\overline{n}$	$:$	Nat	(numerals)
$\overline{n}$	$=$	$\lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ !(\lambda x.f(\cdots(fx)\cdots))$	
add	$:$	$\mathsf{Nat} \multimap (\mathsf{Nat} \multimap \mathsf{Nat})$	(addition)
add	$=$	$\lambda n.\lambda m.\lambda f.\mathsf{let}\ !f = f\ \mathsf{in}$	
		$\mathsf{let}\ !y = n!f\ \mathsf{in}$	
		$\mathsf{let}\ !y' = m!f\ \mathsf{in}\ !(\lambda x.y(y'x))$	
mult	$:$	$\mathsf{Nat} \multimap (\mathsf{Nat} \multimap \mathsf{Nat})$	(multiplication)
mult	$=$	$\lambda n.\lambda m.\lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ n(m!f)$	

Figure 5.8: Representation of natural numbers and some arithmetic functions

$A \times B$	$=$	$\forall t.(A \multimap B \multimap t) \multimap t$	(type of pairs)
$\langle M, N\rangle$	$:$	$A \times B$	(pair representation)
$\langle M, N\rangle$	$=$	$\lambda x.xMN$	
fst	$:$	$\forall t, t'.t \times t' \multimap t$	(left destructor)
fst	$=$	$\lambda p.p(\lambda x.\lambda y.x)$	
snd	$:$	$\forall t, t'.t \times t' \multimap t'$	(right destructor)
snd	$=$	$\lambda p.p(\lambda x.\lambda y.y)$	

Figure 5.9: Representation of pairs

5.5.2 Elementary programming

We now would like to demonstrate the intensional expressivity of $\lambda^{!\|}_{EAT}$. As a first step, we show that it is possible to implement the operational semantics of references. Then we show how to program the iteration of operations producing side effects on an inductive data structure, possibly in parallel. Eventually we show to what extent it is possible to exchange data between regions of different depths.

Implementing references

The primitive read operation (get) of $\lambda^{!\|}_{EAT}$ is to consume the value from the store, that is the operational semantics of communication channels. We have seen that, in the affine-intuitionistic system of $\lambda^{!\|}$, we can copy a value from the store only if the value is duplicable, *i.e.* of the shape $!V$ (see Section 3.3.4). This amounts to simulate the operational semantics of imperative references.

We would like to point out that references can also be implemented in $\lambda^{!\|}_{EAT}$. Indeed it is easy to see that the *consume-and-rewrite* mechanism recalled in Equation (5.4) can be typed in $\lambda^{!\|}_{EAT}$.

$$\mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, !x); !x \parallel r \Leftarrow !V \longrightarrow^* !V \parallel r \Leftarrow !V \tag{5.4}$$

Therefore, in the rest of this section we assume that we dispose of these two reading rules as primitive operations:

$$\begin{array}{lllll} (\mathsf{get}) & C[E[\mathsf{get}(r)]] \parallel r \Leftarrow V & \longrightarrow & C[E[V]] & \text{if } V \neq !V' \\ (\mathsf{get}_!) & C[E[\mathsf{get}(r)]] \parallel r \Leftarrow !V & \longrightarrow & C[E[!V]] \parallel r \Leftarrow !V & \end{array}$$

Since ($\mathsf{get}_!$) can be translated into a consume-and-rewrite mechanism, we can safely assume that the elementary bound (Theorem 5.3.3) is still valid with these two reading rules. Subject reduction (Proposition 5.4.7) and progress (Proposition 5.4.8) also remain valid.

Iteration with side effects

Now we show that it is possible to program the iteration of operations producing side effects on an inductive data structure, possibly in parallel. In order to ease the programming style, we assume the language features dynamic locations rather than constant regions names. For a more detailed correspondence between locations and regions see Section 2.4.

Following Church encodings, we define the representation of lists and the associated foldr function in Figure 5.10. In the following we abbreviate $\lambda x.\mathsf{let}\ !x = x\ \mathsf{in}\ M$ by $\lambda^{!}x.M$ and we write $\mathsf{set}(r, M)$ for $(\lambda x.\mathsf{set}(r, x))M$.

$$
\begin{array}{rcll}
\mathsf{List}\ A & = & \forall t.!(A \multimap t \multimap t) \multimap !(t \multimap t) & \text{(type of lists)} \\
\\
[u_1, \ldots, u_n] & : & \mathsf{List}\ A & \text{(list represent.)} \\
[u_1, \ldots, u_n] & = & \lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ !(\lambda x.fu_1(fu_2 \ldots (fu_n x)) & \\
\\
\mathsf{foldr} & : & \forall u.\forall t.!(u \multimap t \multimap t) \multimap \mathsf{List}\ u \multimap !t \multimap !t & \text{(fold right)} \\
\mathsf{foldr} & = & \lambda f.\lambda l.\lambda z.\mathsf{let}\ !z = z\ \mathsf{in}\ \mathsf{let}\ !y = lf\ \mathsf{in}\ !(yz) &
\end{array}
$$

Figure 5.10: Representation of lists

Here is the function update taking as argument a memory location x related to region r and multiplying by 2 the numeral stored at that location:

$$
\begin{array}{l}
r : (3, !\mathsf{Nat}); - \vdash^2 \mathsf{update} : !\mathsf{Reg}_r!\mathsf{Nat} \multimap !\mathsf{Unit} \multimap !\mathsf{Unit} \\
\mathsf{update} = \lambda^! x.\lambda z.!\mathsf{set}(x, \mathsf{let}\ !y = \mathsf{get}(x)\ \mathsf{in}\ !(\mathsf{mult}\ \overline{2}\ y))
\end{array}
$$

Then we define the program run that iterates the function update over a list $[!x, !y, !z]$ of 3 memory locations:

$$
\begin{array}{l}
r : (3, !\mathsf{Nat}); - \vdash^1 \mathsf{run} : !!\mathsf{Unit} \\
\mathsf{run} = \mathsf{foldr}\ !\mathsf{update}\ [!x, !y, !z]\ !!\star
\end{array}
$$

All addresses have type $!\mathsf{Reg}_r!\mathsf{Nat}$ and thus relate to the same region r. Finally, the program run in parallel with some store assignments reduces as expected:

$$
\begin{array}{c}
\mathsf{run} \parallel x \Leftarrow !\overline{m} \parallel y \Leftarrow !\overline{n} \parallel z \Leftarrow !\overline{p} \\
\longrightarrow^* !!\star \parallel x \Leftarrow !\overline{2m} \parallel y \Leftarrow !\overline{2n} \parallel z \Leftarrow !\overline{2p}
\end{array}
$$

Building on this example, suppose we want to write a program of three threads where each thread concurrently increments the numerals pointed by the memory locations of the list. Here is the function gen_threads taking a functional f and a value x as arguments and generating three threads where f is applied to x:

$$
\begin{array}{l}
r : (3, !\mathsf{Nat}); - \vdash^0 \mathsf{gen\_threads} : \forall t.\forall t'.!(t \multimap t') \multimap !t \multimap \mathbf{B} \\
\mathsf{gen\_threads} = \lambda^! f.\lambda^! x.!(fx) \parallel !(fx) \parallel !(fx)
\end{array}
$$

We define the functional F like run but parametric in the list:

$$
\begin{array}{l}
r : (3, !\mathsf{Nat}); - \vdash^1 \mathsf{F} : \mathsf{List}\ !\mathsf{Reg}_r!\mathsf{Nat} \multimap !!\mathsf{Unit} \\
\mathsf{F} = \lambda l.\mathsf{foldr}\ !\mathsf{update}\ l\ !!\star
\end{array}
$$

Finally the concurrent iteration is defined in run_threads:

$$
\begin{array}{l}
r : (3, !\mathsf{Nat}); - \vdash^0 \mathsf{run\_threads} : \mathbf{B} \\
\mathsf{run\_threads} = \mathsf{gen\_threads}\ !\mathsf{F}\ ![!x, !y, !z]
\end{array}
$$

The program is well-typed and has depth 4 with side effects occurring at depth 3. Different thread interleavings are possible and here is one of them:

$$\begin{array}{l}\mathsf{run\_threads} \parallel x \Leftarrow !\overline{m} \parallel y \Leftarrow !\overline{n} \parallel z \Leftarrow !\overline{p} \\ \longrightarrow^* !!!\star \parallel x \Leftarrow !\overline{6m} \parallel y \Leftarrow !\overline{6n} \parallel z \Leftarrow !\overline{6p}\end{array}$$

Remark 5.5.4. The reader may have noticed a flaw in the above programming examples. The Church representation of data types assumes a strong reduction that allows reduction under binders, while we only use a call-by-value evaluation strategy. For example, the program $\mathsf{mult}\ \overline{2}\ \overline{1}$ which should reduces to $\overline{2}$ only reduces to $\lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ \overline{2}(\overline{1}!f)$ with a call-by-value strategy.

One solution is to consider primitive data types instead of Church-style encodings. For natural numbers, this requires to add a new reduction rule

$$C[E[\overline{n} * \overline{m}]] \longrightarrow C[E[\overline{n * m}]]$$

where $*$ stands for any arithmetic operation and to consider the evaluation contexts $E * M$ and $V * E$. Additional typing rules also have to be considered. We preferred not to introduce primitive data types in order keep the language as simple as possible.

A second solution is to extend $\longrightarrow$ to be the largest reduction relation, thus allowing reductions under binders. This would require to extend the proofs of elementary bound, subject reduction and progress to this new reduction relation. We are confident that the proof extensions would hold since they do in the functional case. However, strong reduction does not make any sense in ML-like languages with side effects.

Exchanging data between regions

In the above programming example, we see that the depth of a region is determined by the structure of the program. A natural question that arises is if it is possible to exchange data between regions of different depth. We show that the answer is positive if the depth of the exchanged data can be preserved.

Let us consider an untyped region context

$$R = r : \delta, r' : \delta'$$

with two regions r and r' such that $\delta < \delta'$. We first consider the case where we want to transfer a value from r to r' and then the converse.

1. Intuitively, if we want to transfer a value V which is stored at depth δ in r to a deeper region r', V must be of the shape $!^{\delta'-\delta}V'$ so that the depth of V' can be preserved. Let us define the following abbreviation by induction on $i \geq 1$:

$$\begin{array}{rcll}(\mathsf{let}\ !^1x = M\ \mathsf{in}\ !^1N) & = & (\mathsf{let}\ !x = M\ \mathsf{in}\ !N) & \\ (\mathsf{let}\ !^ix = M\ \mathsf{in}\ !^iN) & = & (\mathsf{let}\ !x = M\ \mathsf{in}\ !(\mathsf{let}\ !^{i-1}x = x\ \mathsf{in}\ !^{i-1}N)) & i > 1\end{array}$$

The program

$$P = !^{\delta}(\mathsf{let}\ !^{\delta'-\delta}z = \mathsf{get}(r)\ \mathsf{in}\ !^{\delta'-\delta}\mathsf{set}(r', z))$$

is well-formed with region context R. Indeed, the read on r occurs at depth δ and the write on r' occurs at depth δ'. The transfer then happens as follows:

$$P \parallel r \Leftarrow !^{\delta'-\delta}V' \longrightarrow^{+} !^{\delta'}\star \parallel r' \Leftarrow V'$$

2. To transfer a value V from r' to the shallower region r take the program

$$Q = !^{\delta}((\lambda z.\mathsf{set}(r, z))!^{\delta'-\delta}\mathsf{get}(r'))$$

It is well-formed with region context R because the write on r occurs at depth δ and the read on r' occurs at depth δ'. The transfer happens as follows:

$$Q \parallel r' \Leftarrow V \longrightarrow^{+} !^{\delta}\star \parallel r \Leftarrow !^{\delta'-\delta}V$$

We observe that the operation consists in adding the required bangs such that the depth of V is preserved.

5.5.3 On stratification

It is interesting to note that the stratification of regions by means of a type and effect system, as seen in Chapter 2, is not required to ensure the termination of programs. Instead, the elementary affine depth system stratifies regions by depth levels. For example, take

$$M_r = \mathsf{let}\ !z = \mathsf{get}(r)\ \mathsf{in}\ !(z\star)$$

We can write a diverging program that keeps incrementing its depth:

$$\begin{array}{ll} & !M_r \parallel r \Leftarrow !(\lambda y.M_r) \\ \longrightarrow^{*} & !!M_r \parallel r \Leftarrow !(\lambda y.M_r) \\ \longrightarrow^{*} & !!!M_r \parallel r \Leftarrow !(\lambda y.M_r) \\ \longrightarrow^{*} & \ldots \end{array}$$

This program is not well-formed since $\mathsf{get}(r)$ occurs at depth $R(r) + 1$ in the store. In fact, a duplicable value which is stored in a region r cannot generate side effects on the region r itself.

The stratification by depth levels allows however to consider programs which would not be stratified by a type and effect system. Take

$$N_r = \mathsf{get}(r)\star$$

We have

$$\begin{array}{ll} & N_r \parallel r \Leftarrow \lambda y.N_r \parallel r \Leftarrow \lambda y.N_r \\ \longrightarrow^{*} & N_r \parallel r \Leftarrow \lambda y.N_r \\ \longrightarrow^{*} & N_r \\ \not\longrightarrow^{*} & \end{array}$$

This program is well-formed since $\mathsf{get}(r)$ occurs at depth $R(r) = 0$ in every part of the program. In fact, *Landin's trick* that we recall in Equation (5.5)

$$\mu_r f.\lambda x.M = \mathsf{set}(r, \lambda x.M[\lambda y.\mathsf{get}(r)y/f]); \mathsf{get}(r)\star \tag{5.5}$$

is even well-formed since every $\mathsf{get}(r)/\mathsf{set}(r)$ occurs at the same fixed depth. However, while we have seen in Section 2.2.4 that it can produce divergence, here it gets stuck due to the consumption of the stored values. Take

$$\lambda x.M = \lambda x.\mathsf{set}(r', x); f(x + \overline{1})$$

We observe

$$\begin{array}{ll} & (\mu_r f.\lambda x.M)\overline{0} \\ \longrightarrow^* & M[\lambda y.\mathsf{get}(r)y/f, \overline{1}/x] \parallel r' \Leftarrow \overline{0} \\ \longrightarrow^* & \mathsf{get}(r)\overline{2} \parallel r' \Leftarrow \overline{0} \parallel r' \Leftarrow \overline{1} \\ \not\longrightarrow & \end{array}$$

This shows that the stratification by depth levels allows to consider circular side effects, as long as the stored values are not duplicable. Yet, we do not know if this has a practical application.

Chapter 6

A polynomial λ-calculus

In this chapter, we introduce a *polynomial λ-calculus* and prove that a class of well-formed programs terminate in polynomial time under an arbitrary reduction strategy. We precise that this result is not our contribution but a recasting of K. Terui's work on the *light affine λ-calculus* [Ter07].

Considering a class of well-formed programs, the specific *shallow-first* strategy[1] is known to be polynomial by a combinatorial argument that goes back to Girard [Gir98]. Contrary to our contribution on the elementary case, the combinatorial argument does not lift to every reduction strategy. Terui's contribution [Ter07] is to show that *every* strategy terminates in polynomial time by proving that every reduction sequence can be transformed into a *longer* one which is shallow-first.

This chapter is of particular importance because it presents the above transformation method which we will use to contribute a *polynomial concurrent λ-calculus* in the next chapter.

Outline The chapter is organized as follows. In Section 6.1, we present the syntax and reduction of the polynomial λ-calculus. It features two modalities '!' (*bang*) and '§' (*paragraph*), thus we call it $\lambda^{!\S}$. In Section 6.2, we present the principle of *light* stratification by depth levels and show why it entails termination in polynomial time of the shallow-first strategy. In Section 6.3, we propose a *light linear depth system* ($\lambda^{!\S}_{LLD}$) that applies light stratification on programs and which is a linear variant of Terui's system [Ter07]. In Section 6.4 we show how arbitrary reduction sequences can be transformed into longer shallow-first sequences. The strict linearity of the system allows for a much simpler presentation than Terui. Finally, in Section 6.5 we prove that the shallow-first strategy is polynomial, which entails that every strategy is polynomial.

[1] Redexes are eliminated in depth-increasing order.

In Figure 6.1, we illustrate the difference of proof method between the elementary case and the polynomial case. In $\lambda^{!}_{EAD}$, a combinatorial argument suffices

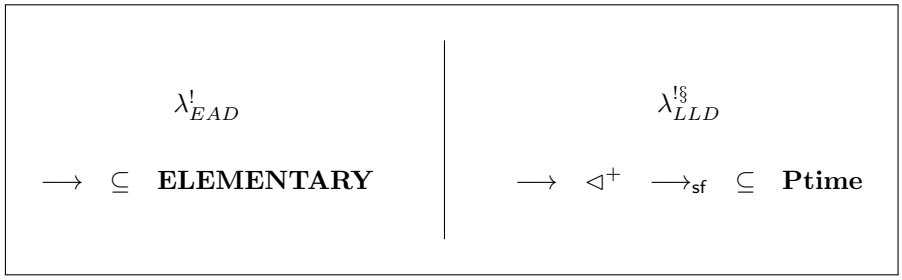


Figure 6.1: Comparison of proof methods

to show that every reduction strategy $\longrightarrow$ is in elementary time. In $\lambda^{!\S}_{LLD}$, a combinatorial argument only shows that the shallow-first strategy $\longrightarrow_{\mathsf{sf}}$ is in polynomial time. Then, one proves that every reduction sequence can be transformed into a shallow-first sequence which is longer (that is written $\longrightarrow \lhd^{+} \longrightarrow_{\mathsf{sf}}$). Thus we conclude that every reduction strategy is in polynomial time.

6.1 Syntax and reduction

The logic **LAL** is bimodal: in addition to the bang modality '!', it features a *paragraph* modality '§'. We will see in the next section how they are used to characterize polynomial time. In this section, we introduce the syntax and reduction of the polynomial λ-calculus ($\lambda^{!\S}$) that has two modal constructors '!' and '§', and the corresponding $\mathsf{let}\,!$ and $\mathsf{let}\,\S$-expressions.

The syntax and reduction of $\lambda^{!\S}$ is introduced in Figure 6.2. They correspond exactly to the syntax and reduction of the light affine λ-calculus given by Terui [Ter07]. The reduction handles the two modalities in a uniform way. The only difference with $\lambda^{!}$ (Figure 4.2) is the addition of constructs for the paragraph modality.

In the sequel, we write $\dagger$ for any $\dagger \in \{!, \S\}$ and we define

$$\dagger^{0} M = M$$
$$\dagger^{n+1} M = \dagger(\dagger^{n} M)$$

For example $\dagger^{2} M$ represents $!!M$, $!\S M$, $\S !M$ and $\S\S M$.

The representation of terms by abstract syntax trees extends trivially to $\lambda^{!\S}$ and we denote and count occurrences of terms as previously. The notion of depth is extended to count both the number of bangs and paragraphs.

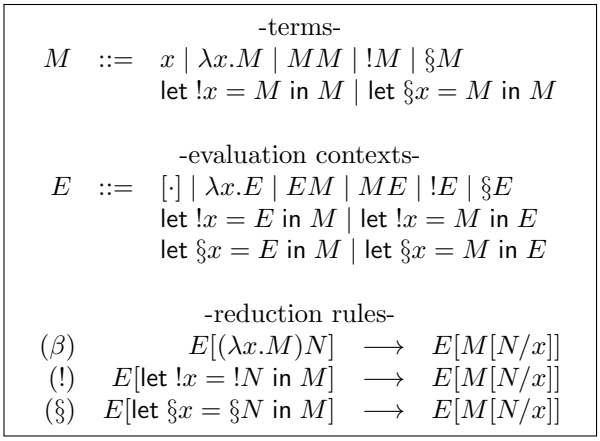


Figure 6.2: Syntax and reduction of $\lambda^{!\S}$

Definition 6.1.1 (Depth)**.** The *depth* $d(w)$ of an occurrence w in a term M is the number of '†' labels that the path leading to the end node crosses. The depth $d(M)$ of the term M is the maximum depth of its occurrences.

In the functional case, it makes no difference whether the definition is revised (Definition 5.2.1) or not.

6.2 Light stratification

In this section, we present the principle of *light stratification* that is inherent to **LAL** and shows why it entails the termination in polynomial time of programs.

As in the elementary case, the principle of light stratification is to preserve the depth of occurrences by reduction. We have seen that this can be ensured by the two following criteria:

1. if a λ-abstraction occurs at depth i and binds a variable x, then x must occur at most once and x must occur at depth i;
2. if a $\mathsf{let}\,!$-expression occurs at depth i and binds a variable x, then x must occur at depth $i+1$ and x may occur arbitrarily many times.

These criteria are guaranteed by the elementary affine depth system and ensure that programs terminate in elementary time. It is instructive to observe how these criteria let the size of a term grow exponentially. Consider the following term borrowed from Terui [Ter07]:

$$Z = \lambda x.\mathsf{let}\ !y = x\ \mathsf{in}\ !(yy) \tag{6.1}$$

It satisfies the above criteria; then observe the following reduction:

$$\underbrace{Z \dots (Z(Z}_{n \text{ times}}!z)) \longrightarrow^* !(\underbrace{zz \dots z}_{2^n \text{ times}}) \tag{6.2}$$

The size of the term explodes exponentially by repeated application of the duplicating rule (!). In order to get down to polynomial time, Terui considers these two additional criteria:

3. if a $\mathsf{let}\,\S$-expression occurs at depth i and binds a variable x, then x must occur at depth $i+1$ and x must occur at most once;
4. a term $!M$ contain at most one occurrence of free variable while a term $\S M$ can contain arbitrarily many occurrences of free variables.

The term Z in (6.1) does not respect the fourth criterion since its subterm $!(xx)$ contains two occurrences of free variables. We will see that this fourth criteria is intended to guarantee a quadratic size growth of the term. By replacing the bang modality of the term Z with a paragraph modality, we obtain the term Y which respects the four criteria:

$$Y = \lambda x.\mathsf{let}\ !y = x\ \mathsf{in}\ \S(yy) \tag{6.3}$$

We observe that it is not possible to apply the rule (!) repeatedly with Y:

$$\underbrace{Y \dots (Y(Y}_{n \text{ times}}!z)) \longrightarrow^* \underbrace{Y \dots (Y(Y}_{n-2 \text{ times}}(\mathsf{let}\ !y = \S(zz)\ \mathsf{in}\ \S(yy)))) \not\longrightarrow \tag{6.4}$$

Now if we consider

$$W = \lambda x.\mathsf{let}\ \S y = x\ \mathsf{in}\ \S(yy)$$

we can produce an exponential blowup by repeated application of the rule (§). However W is not be well-formed because of the third criterion. In fact, the only duplicating rule is the (!) rule.

As in the elementary case, the polynomial soundness relies on a shallow-first strategy which eliminate redexes in depth-increasing order (see Definition 4.2.1). More precisely, the following properties are shown when $M \overset{i}{\longrightarrow}^* M'$:

$$d(M') \leq d(M) \tag{6.5}$$
$$s(M')_j \leq s(M)_j \text{ for } j < i \tag{6.6}$$
$$s(M')_i < s(M)_i \tag{6.7}$$
$$s(M') \leq s(M)^2 \tag{6.8}$$

Comparing with the properties induced by elementary stratification (Section 4.2), the only difference is that here the size of the term grows at most quadratically. By applying the same arithmetic reasoning, we can show the following proposition.

Proposition 6.2.1 (Polynomial bounds)**.** *The shallow-first evaluation of a lightly stratified term M terminates in at most $s(M)^{2^d}$ steps where d is the depth of M. Moreover the size of the final term is bounded by $s(M)^{2^d}$.*

Proof. For simplicity, assume M is a term such that $d(M) = 2$. By Properties (6.5),(6.6),(6.7) we can eliminate all the redexes of M with the shallow-first sequence $M \overset{0}{\longrightarrow}{}^* M' \overset{1}{\longrightarrow}{}^* M'' \overset{2}{\longrightarrow}{}^* M'''$. By Property (6.8), $s(M'') \leq s(M)^4$. We can safely assume that $s(M''') \leq s(M'') \leq s(M)^4$. By Property (6.7) the length l of the sequence is such that $l \leq s(M) + s(M') + s(M'') = p$. Since we can show that $p \leq s(M)^4$ we can conclude. □

6.3 A light linear depth system

In this section we introduce a *light linear depth system* ($\lambda^{!\S}_{LLD}$) that guarantees that terms respect the four criteria presented in the previous section. The system $\lambda^{!\S}_{LLD}$ is a linear reformulation of Terui's affine system [Ter07] as a set of inference rules. In fact, our system is strictly linear in the sense that it is not possible to bind 0 occurrences of variables and thus it is not possible to discard data. This is a notable restriction but we will see in Section 6.4 that it simplifies the transformation of reduction sequences into shallow-first ones. Also, we will see in the next chapter that side effects can be used to discard data even though the depth system is strictly linear.

While the elementary affine depth system $\lambda^{!}_{EAD}$ is exclusively based on the notion of depth, $\lambda^{!\S}_{LLD}$ is based on the notion of usage to distinguish between occurrences that appear under bangs and occurrences that appear under paragraphs. We chose to keep the name 'depth system' since usages control the depth of occurrences.

First, we define variable contexts Γ as follows:

$$\Gamma \quad = \quad x_1 : u_1, \ldots, x_n : u_n$$

A variable context associates each variable with a usage $u \in \{\lambda, \S, !\}$ which constrains the variable to be bound by a λ-abstraction, a let §-binder or a let !-binder respectively. We write Γ_u if $dom(\Gamma)$ only contains variables with usage u.

A depth judgement has the shape

$$\Gamma \vdash M$$

It should entail the following:

- if $x : \lambda \in \Gamma$ and x occurs free in M then all free occurrences of x appear at depth 0 in M;

- if $x : ! \in \Gamma$ and x occurs free in M then all free occurrences of x appear at depth 1 in M;
- if $x : \S \in \Gamma$ and x occurs free in M then all free occurrences of x appear at depth 1 in M, and it must be in the scope of a $\S$ constructor.

The inference rules of the depth system are presented in Figure 6.3. We give

$$\mathsf{var}\frac{x : \lambda \in \Gamma}{\Gamma \vdash x}$$

$$\mathsf{lam}\frac{\mathsf{FO}(x, M) = 1 \quad \Gamma, x : \lambda \vdash M}{\Gamma \vdash \lambda x.M} \qquad \mathsf{app}\frac{\Gamma \vdash M \quad \Gamma \vdash N}{\Gamma \vdash MN}$$

$$\mathsf{prom}_!\frac{\mathsf{FO}(M) \leq 1 \quad \Gamma_\lambda \vdash M}{\Gamma_!, \Delta_\S, \Psi_\lambda \vdash !M} \qquad \mathsf{elim}_!\frac{\mathsf{FO}(x, N) \geq 1 \quad \Gamma \vdash M \quad \Gamma, x : ! \vdash N}{\Gamma \vdash \mathsf{let}\ !x = M\ \mathsf{in}\ N}$$

$$\mathsf{prom}_\S\frac{\Gamma_\lambda, \Delta_\lambda \vdash M}{\Gamma_!, \Delta_\S, \Psi_\lambda \vdash \S M} \qquad \mathsf{elim}_\S\frac{\mathsf{FO}(x, N) = 1 \quad \Gamma \vdash M \quad \Gamma, x : \S \vdash N}{\Gamma \vdash \mathsf{let}\ \S x = M\ \mathsf{in}\ N}$$

Figure 6.3: A light linear depth system

some intuitions on the rules in the following two items:

- Variables are introduced with usage λ by the rule var. The rule $\mathsf{prom}_!$ updates the usage of variables to ! if they all previously had usage λ. The rule $\mathsf{prom}_\S$ updates the usage of variables to $\S$ for one part and ! for the other part if they all previously had usage λ. In both rules $\mathsf{prom}_!$ and $\mathsf{prom}_\S$, contexts with other usages can be weakened. In fact, λ-abstractions bind variables occurring at depth 0, $\mathsf{let}\,!$-expressions bind variables occurring at depth 1, and $\mathsf{let}\,\S$-expressions bind variables occurring at depth 1 in the scope of a $\S$-terms.
- To control the duplication of data, the rules for binders have predicates which specify how many occurrences can be bound. λ-abstractions and $\mathsf{let}\,\S$-expressions are linear by the predicate $\mathsf{FO}(x, M) = 1$ and $\mathsf{let}\,!$-expressions are at least linear by the predicate $\mathsf{FO}(x, M) \geq 1$. It is therefore not possible to bind 0 occurrences.

Definition 6.3.1 (Well-formedness)**.** A term M is *well-formed* if a judgement $\Gamma \vdash M$ can be derived for some Γ.

Example 6.3.2. The term Y in (6.3) is well-formed with the following derivation:

$$\dfrac{\dfrac{\dfrac{}{x:\lambda\vdash x} \quad \dfrac{\dfrac{\dfrac{}{x:\lambda,y:\lambda\vdash y} \quad \dfrac{}{x:\lambda,y:\lambda\vdash y}}{x:\lambda,y:\lambda\vdash yy}}{x:\lambda,y:!\vdash \S(yy)}}{x:\lambda\vdash \mathsf{let}\ !y=x\ \mathsf{in}\ \S(yy)}}{\vdash \lambda x.\mathsf{let}\ !y=x\ \mathsf{in}\ \S(yy)}$$

The term Z of Equation (6.1) is not well-formed since the rule $\mathsf{prom}_!$ cannot be applied on (yy) for $\mathsf{FO}(yy) > 1$.

The light linear depth systems has the subject reduction property which is based on weakening and substitution lemmas.

Lemma 6.3.3 (Weakening)**.** *If* $\Gamma \vdash M$ *then* $\Gamma, \Gamma' \vdash M$.

Proof. By induction on the depth judgement. □

The substitution lemma comes into three flavors.

Lemma 6.3.4 (Substitution)**.**

1. *If* $\Gamma, x:\lambda \vdash M$ *and* $\Gamma \vdash N$ *then* $\Gamma \vdash M[N/x]$.
2. *If* $\Gamma, x:\S \vdash M$ *and* $\Gamma \vdash \S N$ *then* $\Gamma \vdash M[N/x]$.
3. *If* $\Gamma, x:! \vdash M$ *and* $\Gamma \vdash !N$ *then* $\Gamma \vdash M[N/x]$.

Proof. By induction on the depth judgement of M. □

Proposition 6.3.5 (Subject reduction)**.** *If* $\Gamma \vdash M$ *and* $M \longrightarrow M'$ *then* $\Gamma \vdash M'$ *and* $d(M) \geq d(M')$.

Proof. By case analysis on the reduction rules and the above lemmas. □

6.4 Shallow-first transformation

In this section we review the method to transform arbitrary reduction sequences into longer shallow-first ones. The strict linearity of $\lambda^{!\S}_{LLD}$ allows for a much simpler presentation than Terui, although we loose the possibility to discard data. Yet, this discarding power will be recovered with the help of side effects.

The transformation procedure is an iterating process where each iteration consists in commuting two consecutive reduction steps which appear in deep-first[2] order. Concretely, we want to prove the following lemma.

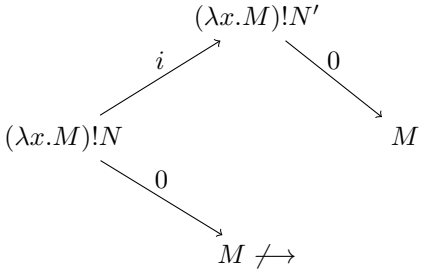
Lemma 6.4.1 (Swapping)**.** *If* M *is a well-formed term such that* $M \xrightarrow{i} M_1 \xrightarrow{j} M_2$ *and* $i > j$*, then there exists* M' *such that* $M \xrightarrow{j} M' \xrightarrow{i}{}^{+} M_2$.

[2]By deep-first we mean redexes are applied in depth-decreasing order (the contrary of shallow-first).

The key point of the swapping lemma is to return a reduction sequence which of length longer (or equal) than the initial one. We have to be careful of situations where we may obtain shorter sequences. Consider the term

$$\lambda x.M$$

where x does not occur free in M. We see in the following diagram where $i > 0$ that the deep-first sequence becomes a single reduction step if we eliminate redexes in shallow-first order:



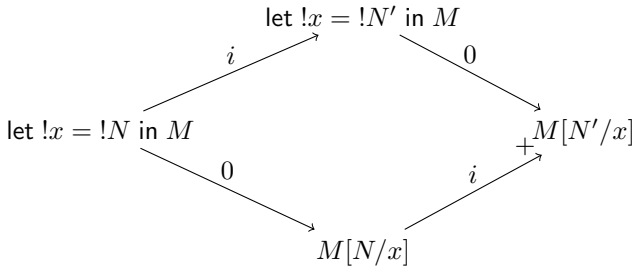
(6.9)

It turns out that this case does not need to be considered since $\lambda x.M$ is not well-formed (it is not strictly linear). As a result, a redex can never be discarded and this ensures that the swapping procedure returns longer sequences.

Let us illustrate the case where the swapping lemma returns a strictly longer sequence. Consider the well-formed term

$$\mathsf{let}\ !x = !M\ \mathsf{in}\ N$$

where x occurs free more than once in N. Clearly, the deep-first sequence of the following diagram can be transformed into a shallow-first one:



(6.10)

Since x occurs more than once in N, the redex of depth i is duplicated by the (!) reduction of depth 0 and several reduction steps are needed to eliminate all of its occurrences.

With the above observations in mind, let us now consider the proof of the swapping lemma.

Proof of Lemma 6.4.1 (swapping). We write N_c the contractum of the reduction $M \xrightarrow{i} M_1$ and N_r the redex of the reduction $M_1 \xrightarrow{j} M_2$. Assume they respectively occur at addresses w_c and w_r in M_1. For any $w \in \{0,1\}^*$ we write $w \sqsubseteq w'$ when w is a prefix of w'. We distinguish three cases:

1. N_c and N_r are separated (neither $w_c \sqsubseteq w_r$ nor $w_c \sqsupseteq w_r$);
2. N_c contains N_r ($w_c \sqsubseteq w_r$);
3. N_r strictly contains N_c ($w_c \sqsupseteq w_r$ and $w_c \neq w_r$).

We discuss them separately:

1. In this case N_c and N_r must occur in different branches of the syntax tree of M_1 and it makes no difficulty to swap the two reduction steps. Moreover it does not change the length of the reduction sequence.
2. If the contractum N_c contains the redex N_r, N_r may not exist yet in M which makes the swapping impossible. We remark that, for any well-formed term T such that $T \xrightarrow{d} T'$, both the redex and the contractum occur at depth d. As a result, N_c must occur at depth i and N_r must occur at depth j. Since $i > j$, it is clear that the contractum N_c cannot contain the redex N_r and this case is void.
3. This case would cover situations like (6.9) where the length of the reduction sequence is shortened but we have seen that the strict linearity of the depth system prevents any redex to be discarded. This case also covers situations like (6.10) where the length of the reduction sequence can be increased and which is not an issue. □

Remark 6.4.2. The swapping procedure we propose is much simpler than Terui's version. This is due to the fact that his well-formedness system is affine whereas ours is strictly linear and so that his procedure might shorten sequences by discarding redexes as in example (6.9). His solution is to introduce an auxiliary calculus with explicit discarding and to show that every discarding step can be postponed at the end of the reduction, after other steps have been swapped into shallow-first order. As a result, the reduction sequence is not shortened. This is however at the price of introducing quite a lot of extra work: once it is shown that discarding steps can be postponed, additional commutation rules come to complicate the swapping lemma. We conclude that strict linearity brings major proof simplifications. The disadvantage is of course to cause a loss of expressivity but we argue in the next chapter that side effects can be used to recover the discarding power of the language without breaking the strict linearity condition.

The swapping lemma can be generalized to sequences.

Lemma 6.4.3 (Swapping of sequences)**.** *If M is a well-formed term such that $M \xrightarrow{i}{}^{+} M_1 \xrightarrow{j}{}^{+} M_2$ and $i > j$, then there exists M' such that $M \xrightarrow{j}{}^{+}$*

$M' \overset{i}{\longrightarrow}^{+} M_2$.

Proof. By repeated application of Lemma 6.4.1. Note that the intermediate steps between M and M' (respectively M' and M_2) differ from the ones between M and M_1 (resp. M_1 and M_2). □

Finally we can show that any reduction sequence can be simulated by a shallow-first sequence.

Proposition 6.4.4 (Shallow-first transformation)**.** *To any reduction sequence $M_1 \longrightarrow^* M_n$ of a well-formed term M_1 corresponds a shallow-first reduction sequence $M_1 \longrightarrow^* M_n$ which is of length equal or longer.*

Proof. By simple application of the bubble sort algorithm: traverse the initial sequence from M_1 to M_n, compare the depth of each sequence of fixed depth, swap them by Lemma 6.4.3 if they are in deep-first order. Repeat the traversal until no swap is needed. For example, in Figure 6.4, the sequence $M \overset{2}{\longrightarrow} M' \overset{1}{\longrightarrow} M'' \overset{0}{\longrightarrow} M'''$ is transformed into $M \overset{0}{\longrightarrow}^{+} C \overset{1}{\longrightarrow}^{+} B \overset{2}{\longrightarrow}^{+} M'''$. □

M	$\overset{2}{\longrightarrow}$	M'	$\overset{1}{\longrightarrow}$	M''	$\overset{0}{\longrightarrow}$	M'''
M	$\overset{1}{\longrightarrow}$	A	$\overset{2}{\longrightarrow}^{+}$	M''	$\overset{0}{\longrightarrow}$	M'''
M	$\overset{1}{\longrightarrow}$	A	$\overset{0}{\longrightarrow}$	B	$\overset{2}{\longrightarrow}^{+}$	M'''
M	$\overset{0}{\longrightarrow}$	C	$\overset{1}{\longrightarrow}^{+}$	B	$\overset{2}{\longrightarrow}^{+}$	M'''

Figure 6.4: Transformation of $M \longrightarrow^* M'''$ into shallow-first order

6.5 Shallow-first soundness

In this section, we prove that the shallow-first strategy admits polynomial bounds for well-formed terms. As a corollary, this implies that every reduction strategy is polynomial. This section is a reformulation of Terui's combinatorial analysis.

First, for a given depth i, we define an *unfolding* transformation on terms that is intended to duplicate statically the occurrences that will be duplicated dynamically by redexes occurring at depth i.

Definition 6.5.1 (Unfolding)**.** The *unfolding* at depth i of a term M, written

$\sharp^i(M)$, is defined as follows:

$$\begin{array}{rcl}
\sharp^i(x) & = & x \\
\sharp^i(\lambda x.M) & = & \lambda x.\sharp^i(M) \\
\sharp^i(MN) & = & \sharp^i(M)\sharp^i(N)
\end{array}$$

$$\sharp^i(\dagger M) = \begin{cases} \dagger\sharp^{i-1}(M) & \text{if } i > 0 \\ \dagger M & \text{if } i = 0 \end{cases}$$

$$\sharp^i(\mathsf{let}\ \dagger x = M\ \mathsf{in}\ N) = \begin{cases} \text{if } i = 0, M = !M' \text{ and } \dagger = ! : \\ \mathsf{let}\ !x = \underbrace{MM\ldots M}_{k \text{ times}}\ \mathsf{in}\ \sharp^0(N) \\ \text{where } k = \mathsf{FO}(x, \sharp^0(N)) \\ \\ \text{otherwise:} \\ \mathsf{let}\ \dagger x = \sharp^i(M)\ \mathsf{in}\ \sharp^i(N) \end{cases}$$

For example, take the following reductions occurring at depth 0:

$$\begin{array}{l}
M = \mathsf{let}\ !x = !N\ \mathsf{in}\ (\mathsf{let}\ !y = !x\ \mathsf{in}\ \S(yy))(\mathsf{let}\ !y = !x\ \mathsf{in}\ \S(yy)) \\
\overset{0}{\longrightarrow}{}^*\ \S(NN)\S(NN)
\end{array}$$

The well-formed term M duplicates the occurrence N four times. We observe that the unfolding at depth 0 of M reflects this duplication:

$$\begin{array}{l}
\sharp^0(M) = \mathsf{let}\ !x = !N!N!N!N\ \mathsf{in} \\
\qquad\qquad (\mathsf{let}\ !y = !x!x\ \mathsf{in}\ \S(yy))(\mathsf{let}\ !y = !x!x\ \mathsf{in}\ \S(yy))
\end{array}$$

Unfolded terms are not intended to be reduced. However, the size of an unfolded term (the number of occurrences in the unfolded term) can be used as a non increasing measure in the following way.

Lemma 6.5.2. *Let M be a well-formed term such that $M \xrightarrow{i} M'$. Then $s(\sharp^i(M')) \leq s(\sharp^i(M))$.*

Proof. It is clear that (!) is the only reduction rule that can make the size of a term increase, so let us consider

$$M = E[\mathsf{let}\ !x = !N_2\ \mathsf{in}\ N_1] \xrightarrow{i} M' = E[N_1[N_2/x]]$$

We have

$$\begin{array}{l}
\sharp^i(M) = E'[\mathsf{let}\ !x = \underbrace{!N_2!N_2\ldots!N_2}_{n \text{ times}}\ \mathsf{in}\ \sharp^0(N_1)] \\
\sharp^i(M') = E'[\sharp^0(N_1[N_2/x])]
\end{array}$$

for some context E' and $n = \mathsf{FO}(x, \sharp^0(N_1))$. Therefore we are left to show

$$s(\sharp^0(N_1[N_2/x])) \leq s(\mathsf{let}\ !x = \underbrace{!N_2 !N_2 \dots !N_2}_{n \text{ times}}\ \mathsf{in}\ \sharp^0(N_1))$$

which is clear since N_2 must occur n times in $\sharp^0(N_1[N_2/x])$. □

We observe in the following lemma that the size of an unfolded term bounds quadratically the size of the original term.

Lemma 6.5.3. *If M is well-formed, then for any depth $i \leq d(M)$:*

1. $\mathsf{FO}(\sharp^i(M)) \leq s(M)$,
2. $s(\sharp^i(M)) \leq s(M) \cdot (s(M) - 1)$,

Proof. By induction on M and i. The crucial case is when $i = 0$ and $M = \mathsf{let}\ !x = !N_2\ \mathsf{in}\ N_1$.

1. We have

$$\begin{aligned} & \mathsf{FO}(\sharp^0(M)) \\ = {} & \mathsf{FO}(!N_2) \cdot \mathsf{FO}(x, \sharp^0(N_1)) + \mathsf{FO}(\sharp^0(N_1)) - \mathsf{FO}(x, \sharp^0(N_1)) \\ \leq {} & \mathsf{FO}(\sharp^0(N_1)) \\ \leq {} & s(N_1) \\ \leq {} & s(\mathsf{let}\ !x = !N_2\ \mathsf{in}\ N_1) \end{aligned}$$

   The first inequality is because $\mathsf{FO}(!N_2) \leq 1$ by well-formedness.

2. We have

$$\begin{array}{rcl} s(\sharp^0(M)) & \leq & s(!N_2) \cdot \mathsf{FO}(x, \sharp^0(N_1)) + s(\sharp^0(N_1)) \\ & \leq & s(!N_2) \cdot s(N_1) + s(N_1) \cdot (s(N_1) - 1) \\ & = & s(N_1) \cdot (s(!N_2) + s(N_1) - 1) \\ & < & s(M) \cdot (s(M) - 1) \end{array}$$

   The second inequality is by Lemma 6.5.3-1 and induction hypothesis. □

We can then bound the size of a term after reduction.

Lemma 6.5.4 (Squaring)**.** *Let M be a well-formed term such that $M \xrightarrow{i}{}^* M'$. Then:*

1. $s(M') \leq s(M) \cdot (s(M) - 1)$
2. *the length of the sequence is bounded by $s(M)$*

Proof.

1. By Lemma 6.5.2 it is clear that $s(\sharp^i(M')) \leq s(\sharp^i(M))$. Then by Lemma 6.5.3-2 we obtain $s(\sharp^i(M')) \leq s(M) \cdot (s(M) - 1)$. Finally it is clear that $s(M') \leq s(\sharp^i(M'))$ thus $s(M') \leq s(M) \cdot (s(M) - 1)$.
2. It suffices to remark $s(M')_i < s(M)_i \leq s(M)$. □

Finally we are able to bound polynomially the shallow-first strategy.

Theorem 6.5.5 (Polynomial bounds)**.** *Let M be a well-formed term such that $d(M) = d$ and $M \longrightarrow^* M'$ is shallow-first. Then:*

1. $s(M') \leq s(M)^{2^d}$
2. *the length of the reduction sequence is bounded by* $s(M)^{2^d}$

Proof. The sequence $M \longrightarrow^* M'$ can be decomposed as

$$M = M_0 \overset{0}{\longrightarrow}^* M_1 \overset{1}{\longrightarrow}^* \ldots \overset{d-1}{\longrightarrow}^* M_d \overset{d}{\longrightarrow}^* M_{d+1} = M'$$

1. We observe that by iterating Lemma 6.5.4-1 we obtain $s(M_d) \leq s(M_0)^{2^d}$. Moreover it is clear that $s(M_{d+1}) \leq s(M_d)$. Hence $s(M') \leq s(M)^{2^d}$.
2. We first prove by induction on d that $s(M_0) + s(M_1) + \ldots + s(M_d) \leq s(M_0)^{2^d}$. For $d = 0$ it is trivial. When $d > 0$:

$$\begin{aligned}
& s(M_0) + s(M_1) + \ldots + s(M_d) \\
\leq\ & s(M_0)^{2^{d-1}} + s(M_d) \quad \text{(by induction)} \\
\leq\ & s(M_0)^{2^{d-1}} + s(M_{d-1}) \cdot (s(M_{d-1}) - 1) \quad \text{(by Lemma 6.5.4)} \\
\leq\ & s(M_0)^{2^{d-1}} + s(M_0)^{2^{d-1}} \cdot (s(M_0)^{2^{d-1}} - 1) \quad \text{(by induction)} \\
=\ & s(M_0)^{2^d}
\end{aligned}$$

   By Lemma 6.5.4-2, it is clear that the length of the reduction $M \longrightarrow^* M'$ is bounded by $s(M_0) + s(M_1) + \ldots + s(M_d)$, which is in turn bounded by $s(M_0)^{2^d}$. □

Remark 6.5.6. A minor difference with Terui's result is that he bounds the length of a sequence by a polynomial of degree $d + 1$. Informally the reason is that, due to commutation rules, the size of terms is decreasing at a slower pace. Concretely, this means that in his version of Lemma 6.5.4-2, the length of the sequence is bounded by $s(M)^2$.

We conclude that any reduction strategy can be computed in polynomial time.

Corollary 6.5.7 (Strong termination in polynomial time)**.** *The evaluation of a well-formed term M of size s and depth d can be computed in time $\mathcal{O}(s^{2^{d+2}})$.*

Proof. Let $M \longrightarrow^* M'$ be the reduction sequence of the well-formed term M. By Proposition 6.4.4 we can transform the sequence into a shallow-first sequence $M \longrightarrow^* M'$ of the same length. By Theorem 6.5.5 we know that its length is bounded by $s(M)^{2^d}$ and that $s(M') \leq s(M)^{2^d}$. To conclude, it suffices to remark that each reduction step $M_i \longrightarrow M_{i+1}$ can be performed in time quadratic in the size of M_i. Therefore the running time is bounded by $\mathcal{O}(s^{2^d \cdot 2} \cdot s^{2^d}) \leq \mathcal{O}(s^{2^{d+2}})$. □

Chapter 7

A polynomial concurrent λ-calculus

In this chapter, we present a *polynomial concurrent λ-calculus*. The main contribution is the characterization of a class of *well-formed* concurrent programs that terminate in polynomial time under a peculiar call-by-value strategy. We provide a type system that captures the set well-formed programs that do not go wrong. In particular, we are able to type programs that iterate side effects over inductive data structures.

The polynomial bounds are proved by following Terui's method (see Chapter 6): first, we prove that the shallow-first strategy is polynomial by a combinatorial argument; then, we *try* to show that every call-by-value reduction sequence can be transformed into a shallow-first one of the same length, which would entail that call-by-value is polynomial. However, the transformation is made non-trivial by the presence of side effects and we proceed as follows:

(1) Let $\longrightarrow_{\mathsf{v}}$ be the call-by-value reduction. The transformation may introduce side effects which are not evaluated in call-by-value. Thus, we have to consider a larger reduction $\longrightarrow \supsetneq \longrightarrow_{\mathsf{v}}$ that is very liberal with the evaluation of side effects.

(2) It turns out that the transformation fails for sequences of an arbitrary reduction $\longrightarrow$. We identify a smaller *outer-bang* reduction $\longrightarrow_{\mathsf{ob}}$ (*i.e.* redexes are not reduced under bangs) for which the transformation succeeds. In particular, the transformation succeeds on an *outer-bang by-value* reduction $\longrightarrow_{\mathsf{obv}} \subseteq \longrightarrow_{\mathsf{ob}}$, which shows that $\longrightarrow_{\mathsf{obv}}$ is polynomial.

The above reasoning is illustrated in Figure 7.1. For any reduction $\longrightarrow_x$,$\longrightarrow_y$, we write $\longrightarrow_x \rhd \longrightarrow_y$ when $\longrightarrow_x$ simulates $\longrightarrow_y$ by reduction sequences of the same length. Therefore if $\longrightarrow_x \rhd \longrightarrow_y$ and $\longrightarrow_x \subseteq \mathbf{Ptime}$, then $\longrightarrow_y \subseteq \mathbf{Ptime}$.

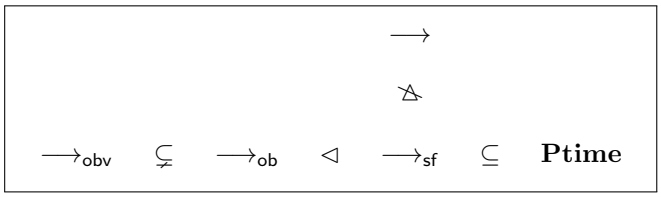


Figure 7.1: The outer-bang-by-value strategy is polynomial

Outline The chapter is organized as follows. We start by presenting the syntax and reduction of the polynomial concurrent λ-calculus ($\lambda^{!\S\|}$) in Section 7.1. Then, we introduce a *light linear depth system* ($\lambda^{!\S\|}_{LLD}$) in Section 7.2 to control the depth of program occurrences. Well-formed programs in the depth system follow Terui's discipline [Ter07] on the functional side and the *stratification of regions* by depth levels. We prove in Section 7.3 that outer-bang reduction sequences can be transformed into shallow-first ones of the same length. In particular, outer-bang *by-value* sequences can be transformed. We review the proof of polynomial soundness of the shallow-first strategy in Section 7.4, which entails that outer-bang by-value is polynomial. We provide a *light linear type system* ($\lambda^{!\S\|}_{LLT}$) in Section 7.5 which results from a simple decoration of the light linear depth system with linear types. We derive the standard subject reduction and progress properties. Finally, we illustrate the expressivity of the type system in Section 7.6 by showing that it is complete with respect to polynomial time in the extensional sense and we give a programming example of a concurrent iteration producing side effects over an inductive data structure. To conclude, we draw a quick comparison with the expressive power offered by *soft λ-calculi*.

The contributions can be summarized by the diagram of Figure 7.2.

7.1 Syntax and reduction

In this section, we present the syntax and reduction of $\lambda^{!\S\|}$. Contrary to the languages introduced previously, we need to define here a reduction which is larger than call-by-value so that we can prove shallow-first transformation in Section 7.3. Thus, we will see that side effects can be produced in non standard ways.

The syntax of $\lambda^{!\S\|}$ is presented in Figure 7.3. It is built from the syntax of $\lambda^{!\S}$ (Figure 6.2) and $\lambda^{\|}$ (Figure 2.2). The difference is that we do not define the syntactic category of values.

As usual, each program can be represented by an abstract syntax tree as in Figure 7.4.

The reduction of $\lambda^{!\S\|}$ is given in Figure 7.5. The following items highlight the

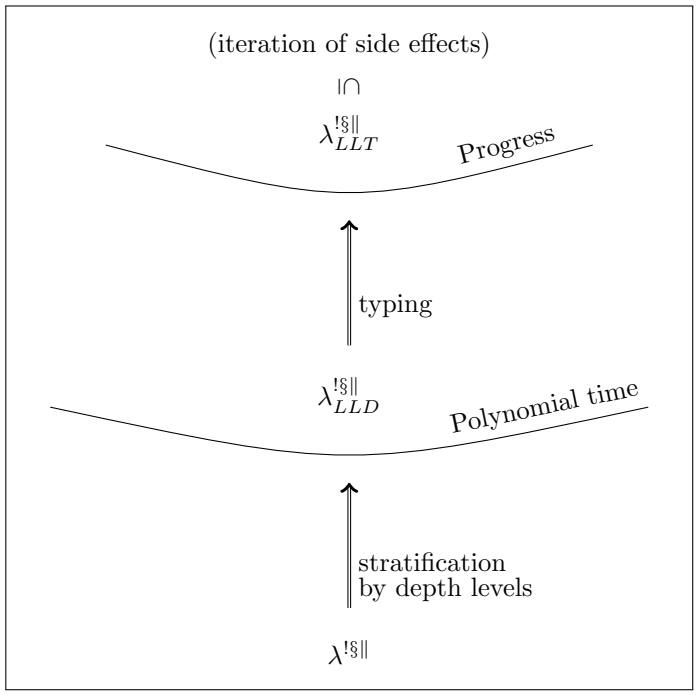


Figure 7.2: From $\lambda^{!\S\|}$ to $\lambda^{!\S\|}_{LLT}$

-variables	$x, y, \ldots$		
-regions	$r, r', \ldots$		
-terms	M	::=	$x \mid r \mid \star \mid \lambda x.M \mid MM \mid !M \mid \S M$
			$\mathsf{let}\ !x = M\ \mathsf{in}\ M \mid \mathsf{let}\ \S x = M\ \mathsf{in}\ M$
			$\mathsf{get}(r) \mid \mathsf{set}(r, M) \mid (M \parallel M)$
-stores	S	::=	$r \Leftarrow M \mid (S \parallel S)$
-programs	P	::=	$M \mid S \mid (P \parallel P)$

Figure 7.3: Syntax of $\lambda^{!\S\|}$

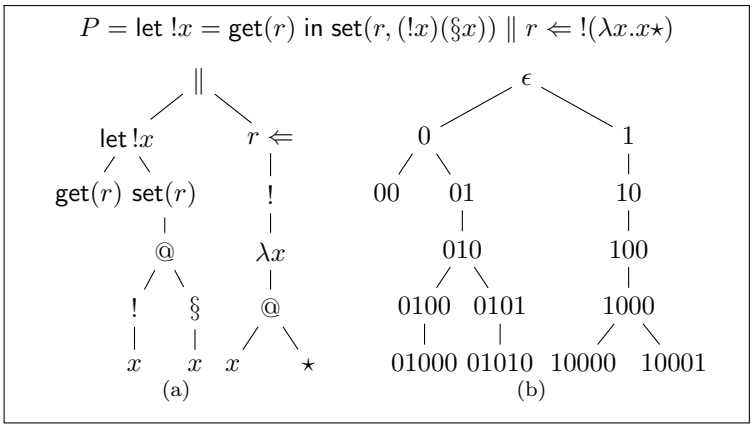


Figure 7.4: Syntax tree and addresses of P

-structural rules-

$$P \parallel P' \equiv P' \parallel P$$
$$(P \parallel P') \parallel P'' \equiv P \parallel (P' \parallel P'')$$

-evaluation contexts-

$$\begin{aligned} E ::= {} & [\cdot] \mid \lambda x.E \mid EM \mid ME \mid !E \mid \S E \\ & \mathsf{let}\ !x = E\ \mathsf{in}\ M \mid \mathsf{let}\ \S x = E\ \mathsf{in}\ M \\ & \mathsf{let}\ !x = M\ \mathsf{in}\ E \mid \mathsf{let}\ \S x = M\ \mathsf{in}\ E \\ & \mathsf{set}(r, E) \mid r \Leftarrow E \mid (E \parallel P) \mid (P \parallel E) \end{aligned}$$

-reduction rules-

(β)	$E[(\lambda x.M)N]$	$\longrightarrow$	$E[M[N/x]]$	
$(!)$	$E[\mathsf{let}\ !x = !N\ \mathsf{in}\ M]$	$\longrightarrow$	$E[M[N/x]]$	
$(\S)$	$E[\mathsf{let}\ \S x = \S N\ \mathsf{in}\ M]$	$\longrightarrow$	$E[M[N/x]]$	
(get)	$E[\mathsf{get}(r)] \parallel r \Leftarrow M$	$\longrightarrow$	$E[M]$	
(set)	$E[\mathsf{set}(r, M)]$	$\longrightarrow$	$E[\star] \parallel r \Leftarrow M$	if $\mathsf{FV}(M) = \emptyset$
(gc)	$E[\star \parallel M]$	$\longrightarrow$	$E[M]$	

Figure 7.5: Reduction of $\lambda^{!\S\|}$

notable differences with the previously defined call-by-value reductions:

- We can read/write any term to the store. Since the rule (set) generates a *global* assignment (*i.e.* out of the evaluation context), we require M to be closed such that it does not contain variables bound in E.
- The reduction can take place at any program point and is completely non-deterministic since an evaluation context can be any program with a hole $[\cdot]$. In particular, contexts of the shape $r \Leftarrow E$ allow evaluation in the store. We give an example of such reduction in Figure 7.6.
- A new reduction rule (gc) permits to *garbage collect* a terminated thread. This rule will be used to simulate the discarding of data, even though we adopt a strictly linear depth system.

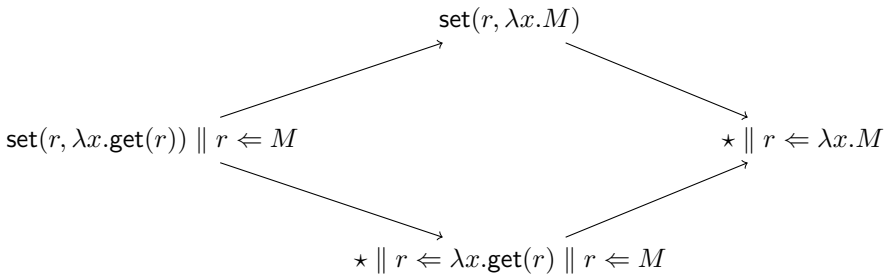


Figure 7.6: Reducing in the store

In the rules $(\beta), (!), (\S)$ and (gc), the *redex* denotes the term inside the context of the left hand-side and the *contractum* denotes the term inside the context of the right hand-side. In the rule (get), the redex is $\mathsf{get}(r)$ and the contractum is M. In the rule (set), the redex is $\mathsf{set}(r, M)$ and the contractum is M. As usual $\longrightarrow^+$ denotes the transitive closure of $\longrightarrow$ and $\longrightarrow^*$ denotes the reflexive closure of $\longrightarrow^+$.

7.2 A light linear depth system

In this section, we introduce the light linear depth system $\lambda^{!\S\parallel}_{LLD}$ that characterizes a class of *well-formed* programs. It is built from the functional system $\lambda^{!\S}_{LLD}$ (see Section 6.3) and the stratification of regions by depth levels (see Section 5.2.1). We will see that even though the depth system is strictly linear, we can simulate the discarding of data.

We first extend the revised notion of depth to count both modalities.

Definition 7.2.1 (Revised depth)**.** Let P be a program and R a region context where $dom(R)$ contains all the regions of P. The *revised depth* $rd(w)$ of an occurrence w of P is the number of '†' labels that the path leading to the end node crosses, plus $R(r)$ if the path crosses a store label '$r \Leftarrow$'. The revised depth $rd(P)$ of a program P is the maximum revised depth of its occurrences.

As in the elementary case, we will simply say 'depth' and write $d(w)$ for short.

Then we introduce variable and region context in Figure 7.7. Region contexts

$$\begin{array}{rcl} R & = & r_1 : \delta_1, \ldots, r_n : \delta_n \\ \Gamma & = & x_1 : u_1, \ldots, x_n : u_n \end{array}$$

Figure 7.7: Variable and region contexts of $\lambda^{!\S\|}_{LLD}$

come from $\lambda^{!}_{EAD}$ (Section 5.2) and variable contexts come from $\lambda^{!\S}$ (Section 6.3). A region context associates a natural number δ_i to each region r_i and the rules of the depth system will be designed so that $\mathsf{get}(r_i)$ and $\mathsf{set}(r_i, M)$ may only occur at depth δ_i. A variable context associates each variable with a usage $u \in \{\lambda, \S, !\}$ which constrains the variable to be bound by a λ-abstraction, a $\mathsf{let}\,\S$-binder or a $\mathsf{let}\,!$-binder respectively. We write Γ_u if $dom(\Gamma)$ only contains variables with usage u.

A depth judgment has the shape

$$R; \Gamma \vdash^{\delta} P$$

where δ is a natural number. It should entail the following:

- if $x : \lambda \in \Gamma$ and x occurs free in M then all free occurrences of x appear at depth 0 in M;
- if $x : ! \in \Gamma$ and x occurs free in M then all free occurrences of x appear at depth 1 in M;
- if $x : \S \in \Gamma$ and x occurs free in M then all free occurrences of x appear at depth 1 in M, and it must be in the scope of a $\S$ constructor.
- if $r : \delta' \in R$ then $\mathsf{get}(r)/\mathsf{set}(r)$ may only occur at depth δ' in $\dagger^{\delta} P$.

The inference rules of the depth system are presented in Figure 7.8. They are a combination of the rules of $\lambda^{!\S}$ (Figure 6.3) and the rules of $\lambda^{!}_{EAD}$ (Figure 5.4) dealing with side effects. In fact, we combine usages and depth indexes to control the depth of occurrences. In particular, the depth δ of the judgment is decremented by the rules $\mathsf{prom}_!$ and $\mathsf{prom}_\S$. This allows to stratify regions by depth levels by requiring $\delta = R(r)$ in the rules get and set.

Definition 7.2.2 (Well-formedness)**.** A program P is *well-formed* if a judgment $R; \Gamma \vdash^{\delta} P$ can be derived for some R, Γ and δ.

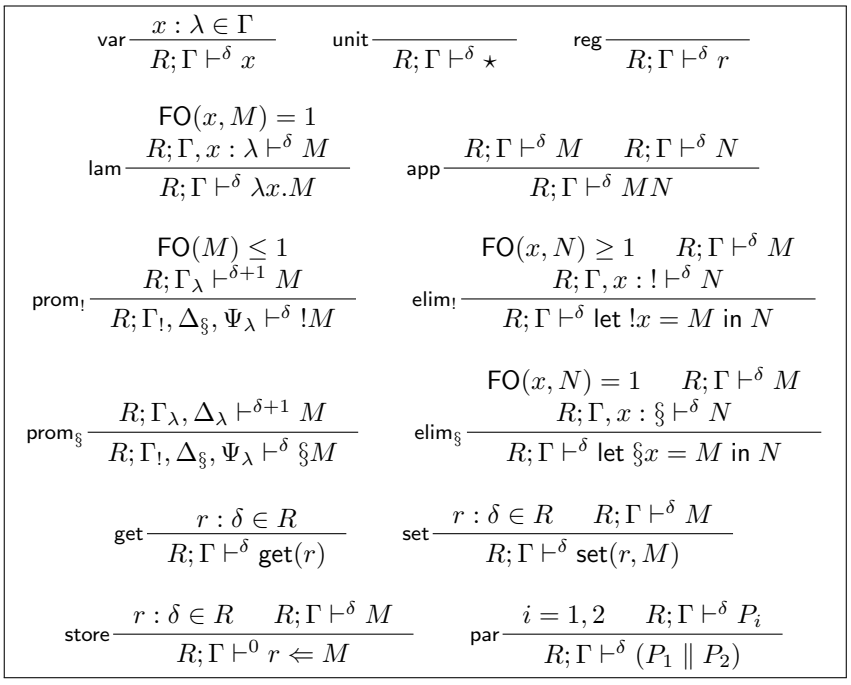


Figure 7.8: The light linear depth system $\lambda^{!\S\parallel}_{LLD}$

Example 7.2.3. The program P of Figure 7.4 is well-formed by composing the two derivation trees of Figure 7.9 with the rule par.

$$\dfrac{\dfrac{}{r:0;-\vdash^0 r}\quad\dfrac{\dfrac{\dfrac{}{r:0;x:\lambda\vdash^1 x}}{r:0;x:!\vdash^0 !x}\quad\dfrac{\dfrac{}{r:0;x:\lambda\vdash^1 x}}{r:0;x:!\vdash^0 \S x}}{r:0;x:!\vdash^0 !x\S x}}{\dfrac{r:0;-\vdash^0 \mathsf{get}(r)\qquad r:0;x:!\vdash^0 \mathsf{set}(r,!x\S x)}{r:0;-\vdash^0 \mathsf{let}\ !x=\mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r,!x\S x)}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{r:0;x:\lambda\vdash^1 x}\quad\dfrac{}{r:0;x:\lambda\vdash^1 \star}}{r:0;x:\lambda\vdash^1 x\star}}{r:0;-\vdash^1 \lambda x.x\star}}{r:0;-\vdash^0 !(\lambda x.x\star)}}{r:0;-\vdash^0 r\Leftarrow !(\lambda x.x\star)}$$

Figure 7.9: Derivation trees

The system $\lambda^{!\S\|}_{LLD}$ enjoys the subject reduction property which is based on the following weakening and substitution lemmas.

Lemma 7.2.4 (Weakening)**.** *If $R;\Gamma\vdash^\delta P$ then $R;\Gamma,\Gamma'\vdash^\delta P$.*

Proof. By induction on the derivation of P. □

Lemma 7.2.5 (Substitution)**.**

1. *If $R;\Gamma,x:\lambda\vdash^\delta M$ and $R;\Gamma\vdash^\delta N$ then $R;\Gamma\vdash^\delta M[N/x]$.*
2. *If $R;\Gamma,x:\S\vdash^\delta M$ and $R;\Gamma\vdash^\delta \S N$ then $R;\Gamma\vdash^\delta M[N/x]$.*
3. *If $R;\Gamma,x:!\vdash^\delta M$ and $R;\Gamma\vdash^\delta !N$ then $R;\Gamma\vdash^\delta M[N/x]$.*

Proof. By induction on the derivation of M. □

Proposition 7.2.6 (Subject reduction)**.** *If $R;\Gamma\vdash^0 P$ and $P\longrightarrow P'$ then $R;\Gamma\vdash^0 P'$ and $d(P)\geq d(P')$.*

Proof. By case analysis on the reduction rules and the above lemmas. □

Discarding The depth system $\lambda^{!\S\|}_{LLD}$ is strictly linear in the sense that it is not possible to bind 0 occurrences. We have seen in Section 6.4 that it greatly simplifies the proof of shallow-first transformation. However, the impossibility to discard data is quite restrictive. In call-by-value, the sequential composition $M;N$ is usually encoded by the term

$$(\lambda z.N)M$$

where $z \notin \mathsf{FV}(N)$. The argument z is used to discard the terminal value of M. We show that side effects can be used to simulate the discarding of data even though the depth system is strictly linear. Assume that we dispose of a specific region gr collecting 'garbage' values at each depth level of a program. Then $M; N$ could be encoded as the well-formed program

$$(\lambda z.\mathsf{set}(gr, z) \parallel N)M$$

By using a call-by-value semantics, we would observe the following reduction sequence

$$\begin{aligned} M; N \longrightarrow^* V; N \longrightarrow \mathsf{set}(gr, V) \parallel N &\longrightarrow \star \parallel N \parallel gr \Leftarrow V \\ &\longrightarrow N \parallel gr \Leftarrow V \end{aligned}$$

where $\star$ has been erased by (gc) and V has been garbage collected into gr. Therefore, we claim that the strict linearity condition does not cause a loss of expressivity.

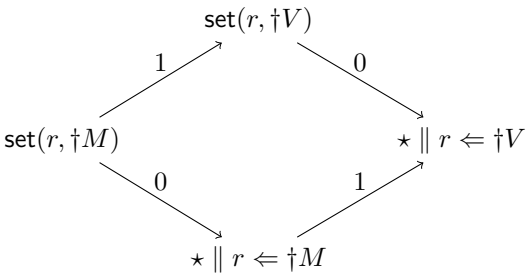
7.3 Shallow-first transformation

In this section, we show that the so-called *outer-bang by-value* reduction sequences (to be defined in Figure 7.11) can be transformed into shallow-first ones of the same length. We first explain in Section 7.3.1 why we need to identify an *outer-bang* strategy, and then we show in Section 7.3.2 how the transformation applies on outer-bang sequences, in particular on the subset of outer-bang by-value sequences.

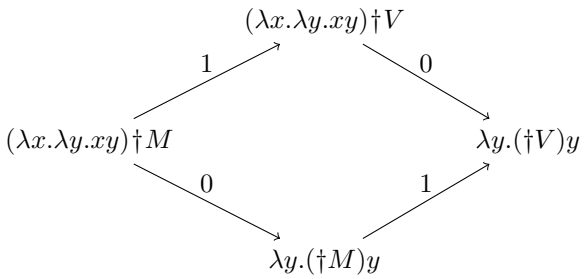
7.3.1 The outer-bang strategy

We recall that transforming a reduction sequence into a shallow-first one is an iterating process where each iteration consists in swapping two consecutive reduction steps which occur in 'deep-first' order (see Section 6.4).

First, let us show that this swapping may introduce side effects that are not executed in call-by-value style. Informally, assume $\dagger V$ denotes a value. In the following diagram,

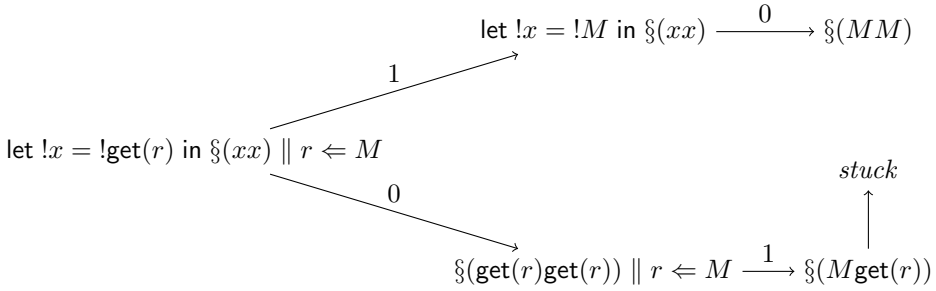


the deep-first call-by-value sequence at the top swaps into the shallow-first one at the bottom with unusual side effects: we write a non-value $\dagger M$ to the store and we reduce *in* the store! As another example, consider the diagram



where the top deep-first call-by-value sequence swaps into a shallow-first one at the bottom. This latter sequence is not in call-by-value style because it reduces inside a λ-abstraction and this is not compatible with the usual notion of value. Therefore the above two diagrams show that we need to consider a reduction larger than call-by-value.

On the other hand, sequences of an arbitrary larger reduction cannot be transformed into shallow-first ones. For instance, in the following diagram,



(7.1)

it is not possible to produce a shallow-first sequence that is confluent with the deep-first one for we try to read the region two times by duplicating the redex $\mathsf{get}(r)$.

In the presence of side effects, it is not surprising that some sequences cannot be transformed into shallow-first ones because they may require exponential time. Consider the well-formed λ-abstraction

$$M = \lambda x.\mathsf{let}\ \S x = x\ \mathsf{in}\ \S\mathsf{set}(r, x); !\mathsf{get}(r) \tag{7.2}$$

which transforms a §-term into a !-term (think of the type $\S A \multimap !A$ that would be rejected in **LAL**). Then, building on program Z given in (6.1) (page 111), take

$$Z' = \lambda x.\mathsf{let}\ !x = x\ \mathsf{in}\ M\S(xx)$$

We observe an exponential explosion of the size of the following well-formed program:

$$\begin{aligned}
&\underbrace{Z'Z'\dots Z'}_{n \text{ times}}!\star \\
\longrightarrow^* &\underbrace{Z'Z'\dots Z'}_{n-1 \text{ times}}(M\S(\star\star)) \\
\longrightarrow^* &\underbrace{Z'Z'\dots Z'}_{n-1 \text{ times}}(!(\star\star)) \parallel gr \Leftarrow \S\star \\
\longrightarrow^* &\,!(\underbrace{\star\star\dots\star}_{2^n \text{ times}}) \parallel \underbrace{gr \Leftarrow \S\star \parallel \dots \parallel gr \Leftarrow \S\star}_{n \text{ times}}
\end{aligned} \tag{7.3}$$

where gr is a region collecting the garbage produced by the sequential composition operator of M (see the encoding in Section 7.2). This previous sequence is not shallow-first since the redexes $\mathsf{set}(r, M)$ and $\mathsf{get}(r)$ occurring at depth 1 are alternatively applied with other redexes occurring at depth 0. A shallow-first strategy would produce the reduction sequence

$$\underbrace{Z'Z'\dots Z'}_{n \text{ times}}!\star \longrightarrow^* !(\star\star\underbrace{\mathsf{get}(r)\mathsf{get}(r)\dots\mathsf{get}(r)}_{n-1 \text{ times}}) \parallel S \not\longrightarrow$$

where S is the same garbage store as previously but we observe no size explosion.

By taking the above observations into account, our contribution is to identify an *outer-bang* strategy whose sequences can be transformed into shallow-first ones of the same length. This strategy is completely liberal except that it does not evaluate in the scope of bang constructors. For example, the sequence at the top of diagram (7.1) is not outer-bang since the redex $\mathsf{get}(r)$ is applied in the scope of a bang. We define the *outer-bang* evaluation contexts F in Figure 7.10. They are not decomposable in contexts of the shape $E[!E']$ and thus cannot

$$\begin{array}{rcl}
F & ::= & [\cdot] \mid \lambda x.F \mid FM \mid MF \mid \S F \\
& & \mathsf{let}\ !x = F\ \mathsf{in}\ M \mid \mathsf{let}\ !x = M\ \mathsf{in}\ F \\
& & \mathsf{let}\ \S x = F\ \mathsf{in}\ M \mid \mathsf{let}\ \S x = M\ \mathsf{in}\ F \\
& & \mathsf{set}(r, F) \mid (F \parallel M) \mid (M \parallel F) \mid r \Leftarrow F
\end{array}$$

Figure 7.10: Outer-bang evaluation contexts

be used to evaluate under bangs, but they are still decomposable in contexts of the shape $E[\S E']$. In the sequel of this chapter, $\longrightarrow_{\mathsf{ob}}$ denotes reduction modulo outer-bang evaluation contexts F instead of regular evaluation contexts E (Figure 7.5).

7.3.2 From outer-bang to shallow-first

We now show how to transform every outer-bang sequence into a shallow-first one of the same length. We also define an *outer-bang by-value* strategy which allows to program in call-by-value style sequences and for which the transformation applies.

A difficulty is that the transformation should preserve the final state of the program and thus be careful with the evaluation order of side effects. For example, assuming F_1 and F_2 do not contain any assignment to region r, the following two reduction steps do not commute:

$$F_1[\mathsf{set}(r,Q)] \parallel F_2[\mathsf{get}(r)] \xrightarrow{i} F_1[\star] \parallel F_2[\mathsf{get}(r)] \parallel r \Leftarrow Q \xrightarrow{j} F_1[\star] \parallel F_2[Q] \tag{7.4}$$

We claim that this is not an issue because the depth system enforces that side effects on a given region can only occur at fixed depth. Therefore i must be equal to j and in general we should never need to swap a read with a write on the same region.

We can prove the crucial swapping lemma.

Lemma 7.3.1 (Swapping)**.** *If P is a well-formed program such that $P \xrightarrow{i}_{\mathsf{ob}} P_1 \xrightarrow{j}_{\mathsf{ob}} P_2$ and $i > j$, then there exists P' such that $P \xrightarrow{j}_{\mathsf{ob}} P' \xrightarrow{i}_{\mathsf{ob}} P_2$.*

Proof. We write M the contractum of the reduction $P \xrightarrow{i}_{\mathsf{ob}} P_1$ and N the redex of the reduction $P_1 \xrightarrow{j}_{\mathsf{ob}} P_2$. Assume they occur at addresses w_m and w_n in P_1. As in the functional case we distinguish three cases:

1. M and N are separated (neither $w_m \sqsubseteq w_n$ nor $w_m \sqsupseteq w_n$);
2. M contains N ($w_m \sqsubseteq w_n$);
3. N strictly contains M ($w_m \sqsupseteq w_n$ and $w_m \neq w_n$).

For each of them we discuss a crucial sub-case:

1. Assume M is the contractum of a (set) rule and that N is the redex of a (get) rule related to the same region. This case has been introduced in example (7.4) where M and N are separated by a parallel node. By well-formedness of P, the redexes $\mathsf{get}(r)$ and $\mathsf{set}(r,Q)$ must occur at the same depth, that is $i = j$, and we conclude that we do not need to swap the reductions.
2. If the contractum M contains the redex N, N may not exist yet in P which makes the swapping impossible. We remark that, for any well-formed program Q such that $Q \xrightarrow{d}_{\mathsf{ob}} Q'$, both the redex and the contractum occur at depth d. In particular, this is true when a contractum occurs in the store as follows:

$$Q = F[\mathsf{set}(r,T)] \xrightarrow{d}_{\mathsf{ob}} Q' = F[\star] \parallel r \Leftarrow T$$

By well-formedness of Q, there exists a region context R such that $R(r) = d$ and the redex $\mathsf{set}(r, T)$ occurs at depth d. By the revised definition of depth, the contractum T occurs at depth d in the store. As a result of this remark, M occurs at depth i and N occurs at depth j. Since $i > j$, it is clear that the contractum M cannot contain the redex N and this case is void.

3. Let N be the redex $\mathsf{let}\ \S x = \S R\ \mathsf{in}\ Q$ and let the contractum M appears in R as in the following reduction sequence

$$\begin{aligned} & P = F[\mathsf{let}\ \S x = \S R'\ \mathsf{in}\ Q] \\ \longrightarrow_{\mathsf{ob}}^{i}\ & P_1 = F[\mathsf{let}\ \S x = \S R\ \mathsf{in}\ Q] \\ \longrightarrow_{\mathsf{ob}}^{j}\ & P_2 = F[Q[R/x]] \end{aligned}$$

By well-formedness, x occurs exactly once in Q. This implies that applying first $P \xrightarrow{j} P'$ cannot discard the redex in R'. Hence, we can produce the following shallow-first sequence of the same length:

$$\begin{aligned} & P = F[\mathsf{let}\ \S x = \S R'\ \mathsf{in}\ Q] \\ \longrightarrow_{\mathsf{ob}}^{j}\ & P' = F[Q[R'/x]] \\ \longrightarrow_{\mathsf{ob}}^{i}\ & P_2 = F[Q[R/x]] \end{aligned}$$

Moreover, the reduction $P' \longrightarrow_{\mathsf{ob}}^{i} P_2$ must be outer-bang for x cannot occur in a !-term in Q. □

Remark 7.3.2. There is a notable difference with the functional version of the swapping procedure (Lemma 6.4.1) which is that the latter may return a longer sequence than the initial one while the present procedure may only return a sequence of the exact same length. The reason is that the outer-bang strategy already duplicates every redex that occurs in the scope of a bang, as in the bottom sequence of diagram (7.1). Therefore the swapping procedure cannot duplicate more redexes. Note also that this explains why we do not generalize the swapping lemma to sequences as in Lemma 6.4.3.

Eventually, we show that any outer-bang sequence can be transformed into a shallow-first one of the same length.

Proposition 7.3.3 (Outer-bang to shallow-first)**.** *To any reduction sequence $P_1 \longrightarrow_{\mathsf{ob}}^{*} P_n$ of a well-formed program P_1 corresponds a shallow-first reduction sequence $P_1 \longrightarrow_{\mathsf{ob}}^{*} P_n$ of the same length.*

Proof. As in the functional case we can transform the sequence into a shallow-first one by using a 'bubble sort' like procedure (see Proposition 6.4.4). □

Outer-bang by-value It is convenient to define an *outer-bang by-value* strategy for programming. The idea is simple: the evaluation is outer-bang and follows a left-to-right call-by-value strategy.

The syntax and reduction of the outer-bang by-value strategy is given in Figure 7.11. We highlight the differences with respect to the arbitrary strategy

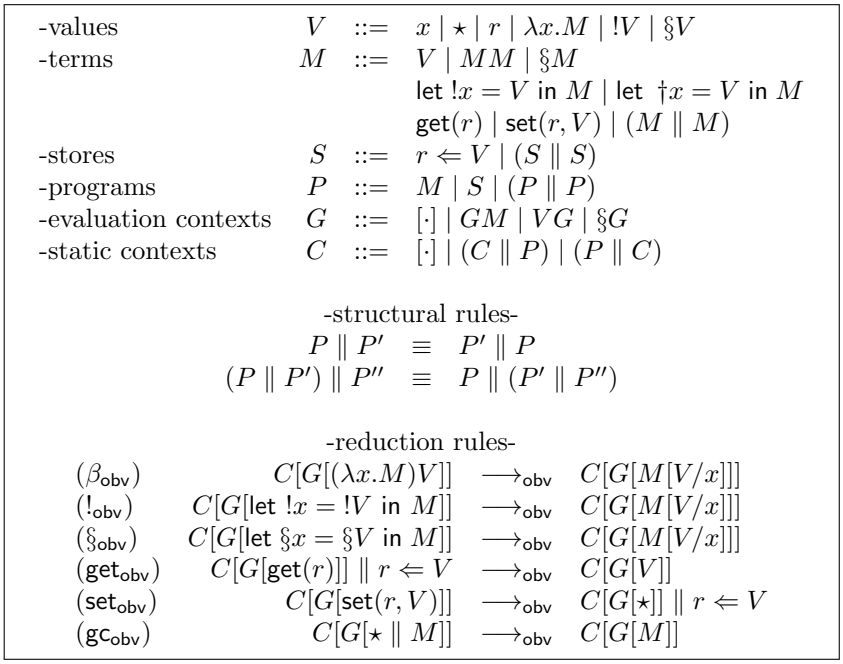


Figure 7.11: Syntax and reduction of the outer-bang by-value strategy

(Figure 7.3 and Figure 7.5):

- we integrate a category of values V, in particular, $!V$ and $\S V$ are values;
- terms M are as expected except that we cannot construct $!M$ if M is not a value;
- store assignments are restricted to values;
- evaluation contexts are outer-bang and follow a left-to-right call-by-value discipline (obviously we do not evaluate in stores); static contexts permit to separate the activation of a thread from the proper evaluation.

The outer-bang by-value reduction is denoted by $\longrightarrow_{\mathsf{obv}}$.

Remark 7.3.4. By not considering $!M$ in the syntax of terms if M is not a value, we prevent the reduction to get stuck on terms of the shape $!M$. This will allow us to prove a progress property for the outer-bang by-value reduction (Proposition 7.5.7). It does not cause a loss of expressivity since we can still prove the completeness of our type system and type interesting programs in the next Section 7.6. This is due to the fact that we can still evaluate in the scope

of paragraph constructors. In the elementary case where ! is the only modality, such a restriction breaks the completeness Theorem 5.5.3 and the outer-bang strategy is way too severe since it only allows to reduce at depth 0.

The contexts G are obviously outer-bang because they cannot be decomposed as $E[!E']$. Therefore the reduction $\longrightarrow_{\mathsf{ob}}$ contains the reduction $\longrightarrow_{\mathsf{obv}}$ and we obtain the following corollary.

Corollary 7.3.5 (From outer-bang by-value to shallow-first)**.** *To any reduction sequence $P_1 \longrightarrow_{\mathsf{obv}}^* P_n$ of a well-formed program P_1 corresponds a shallow-first reduction sequence $P_1 \longrightarrow_{\mathsf{ob}}^* P_n$ of the same length.*

Remark 7.3.6. The shallow-first sequence obtained by transformation is not 'by-value'. Indeed, after some iterations of the swapping lemma, one needs to reduce in the store and in the scope of binders. This justifies the fact that we needed to consider a reduction $\longrightarrow_{\mathsf{ob}}$ larger than $\longrightarrow_{\mathsf{obv}}$.

7.4 Shallow-first is polynomial

In this section, we prove that well-formed programs admit polynomial bounds with a shallow-first strategy. As a corollary, we obtain that the outer-bang by-value strategy can be computed in polynomial time. We precise that this section closely follows the functional case (Section 6.5). The main difficulty has been to extend the light linear depth system so that the combinatorial analysis can be easily adapted.

We recall that, for any depth i, an *unfolding* transformation on terms is intended to duplicate statically the occurrences that will be duplicated by redexes occurring at depth i. The definition is extended to $\lambda^{!\S\|}$ in a straightforward way.

Definition 7.4.1 (Unfolding)**.** The *unfolding* at depth i of a program P, written

$\sharp^i(P)$, is defined as follows:

$$
\begin{array}{rcl}
\sharp^i(x) & = & x \\
\sharp^i(r) & = & r \\
\sharp^i(\star) & = & \star \\
\sharp^i(\lambda x.M) & = & \lambda x.\sharp^i(M) \\
\sharp^i(MN) & = & \sharp^i(M)\sharp^i(N) \\
\sharp^i(\dagger M) & = & \begin{cases} \dagger\sharp^{i-1}(M) & \text{if } i > 0 \\ \dagger M & \text{if } i = 0 \end{cases} \\
\sharp^i(\mathsf{let}\ \dagger x = M\ \mathsf{in}\ N) & = & \begin{cases} \text{if } i = 0, M = !M' \text{ and } \dagger = ! : \\ \mathsf{let}\ !x = \underbrace{MM\ldots M}_{k \text{ times}}\ \mathsf{in}\ \sharp^0(N) \\ \text{where } k = \mathsf{FO}(x, \sharp^0(N)) \\ \\ \text{otherwise:} \\ \mathsf{let}\ \dagger x = \sharp^i(M)\ \mathsf{in}\ \sharp^i(N) \end{cases} \\
\sharp^i(\mathsf{get}(r)) & = & \mathsf{get}(r) \\
\sharp^i(\mathsf{set}(r, M)) & = & \mathsf{set}(r, \sharp^i(M)) \\
\sharp^i(r \Leftarrow M) & = & r \Leftarrow \sharp^i(M) \\
\sharp^i(P_1 \parallel P_2) & = & \sharp^i(P_1) \parallel \sharp^i(P_2)
\end{array}
$$

We refer the reader to Section 6.5 for a concrete illustration of unfolding. Unfolded programs are not intended to be reduced but the size of an unfolded program can be used as a non increasing measure in the following way.

Lemma 7.4.2. *Let P be a well-formed program such that $P \xrightarrow{i} P'$. Then $s(\sharp^i(P')) \leq s(\sharp^i(P))$.*

Proof. First we assume the occurrences labelled with '$\parallel$' and '$r \Leftarrow$' do not count in the size of a program and that '$\mathsf{set}(r)$' counts for two occurrences, such that the size strictly decreases by the rule (set). Then, it is clear that (!) is the only reduction rule that can make the size of a program increase. We refer to the functional case for details (Lemma 6.5.2). □

We observe in the following lemma that the size of an unfolded program bounds quadratically the size of the original program.

Lemma 7.4.3. *If P is well-formed, then for any depth $i \leq d(P)$:*

1. $\mathsf{FO}(\sharp^i(P)) \leq s(P)$,
2. $s(\sharp^i(P)) \leq s(P) \cdot (s(P) - 1)$,

Proof. By induction on P and i. See Lemma 6.5.3. □

We can then bound the size of a program after reduction.

Lemma 7.4.4 (Squaring)**.** *Let P be a well-formed program such that $P \xrightarrow{i}{}^* P'$. Then:*

1. $s(P') \leq s(P) \cdot (s(P) - 1)$
2. *the length of the sequence is bounded by* $s(P)$

Proof. We proceed as in the functional case (Lemma 6.5.4). □

Finally we obtain the following theorem for a shallow-first reduction.

Theorem 7.4.5 (Polynomial bounds)**.** *Let P be a well-formed program such that $d(P) = d$ and $P \longrightarrow^* P'$ is shallow-first. Then:*

1. $s(P') \leq s(P)^{2^d}$
2. *the length of the reduction sequence is bounded by* $s(P)^{2^d}$

Proof. The proof proceeds exactly as in the functional case (see Theorem 6.5.5). □

It is worth noticing that the first bound takes the size of all the threads into account and that the second bound is valid for any thread interleaving.

Corollary 7.4.6 (Outer-bang by-value is polynomial)**.** *The outer-bang by-value evaluation of a well-formed program P of size s and depth d can be computed in time $\mathcal{O}(s^{2^{d+2}})$.*

Proof. Let $P \longrightarrow^*_{\mathsf{obv}} P'$ be the outer-bang by-value reduction sequence of the well-formed program P. By Corollary 7.3.5 we can transform the sequence into a shallow-first sequence $P \longrightarrow^*_{\mathsf{ob}} P'$ of the same length. By Theorem 7.4.5 we know that its length is bounded by s^{2^d} and that $s(P') \leq s^{2^d}$. To conclude, it suffices to remark that each reduction step $P_i \longrightarrow P_{i+1}$ can be performed in time quadratic in the size of P_i. Therefore the running time is bounded by $\mathcal{O}(s^{2^d \cdot 2} \cdot s^{2^d}) \leq \mathcal{O}(s^{2^{d+2}})$. □

7.5 A light linear type system

As in the elementary case, the light linear depth system cannot guarantee that programs do not go wrong. We recall that a well-formed program like

$$\mathsf{let}\ !y = (\lambda x.x)\ \mathsf{in}\ \S(yy) \tag{7.5}$$

stops on a non-value. In this section we propose a polymorphic *light linear type system* $\lambda^{!\S\|}_{LLT}$ which is intended to be used with the outer-bang by-value reduction. We obtain a progress property which states that every well-typed program reduces on a value or gets stuck when it tries to read an empty region.

The light linear type system can be seen as a simple decoration of the light linear depth system with linear types plus additional rules for polymorphism. The following points should give some primary intuitions on the modal types:

- $!A$ is the type of data that can be duplicated;

- $\S A$ is the type of data that has been duplicated and cannot be duplicated further;
- $A \multimap B$ where $A \neq !A'$ is the type of a function that uses its argument only once;
- $!A \multimap B$ is the type of a function that uses its argument more than once.

7.5.1 Types and contexts

We define the syntax of types and contexts in Figure 7.12. The difference with

-type variables	$t, t', \ldots$		
-types	α	$::=$	$\mathbf{B} \mid A$
-result types	A	$::=$	$t \mid \mathsf{Unit} \mid A \multimap \alpha \mid !A \mid \S A \mid \forall t.A \mid \mathsf{Reg}_r A$
-variable contexts	Γ	$::=$	$x_1 : (u_1, A_1), \ldots, x_n : (u_n, A_n)$
-region contexts	R	$::=$	$r_1 : (\delta_1, A_1), \ldots, r_n : (\delta_n, A_n)$

Figure 7.12: Syntax of types and contexts of $\lambda^{!\S\|}_{LLT}$

types and contexts of $\lambda^{!\|}_{EAT}$ (Figure 5.5) is that we added the modal type $\S A$ and that variable contexts have usages instead of depths. Usages play the same role as in $\lambda^{!\S}_{LLD}$ (Section 6.3). Writing $x : (u, A)$ means that the variable x ranges on terms of type A and can be bound according to u. Writing $r : (\delta, A)$ means that the region r contain terms of type A and that $\mathsf{get}(r)$ and $\mathsf{set}(r, M)$ may only occur at depth δ.

As usual, we state in Figure 7.13 when a type is well-formed with respect to a region context.

$$\frac{}{R \downarrow t} \qquad \frac{}{R \downarrow \mathsf{Unit}} \qquad \frac{}{R \downarrow \mathbf{B}} \qquad \frac{R \downarrow A \quad R \downarrow \alpha}{R \downarrow (A \multimap \alpha)}$$

$$\frac{R \downarrow A}{R \downarrow \dagger A} \qquad \frac{r : (\delta, A) \in R}{R \downarrow \mathsf{Reg}_r A} \qquad \frac{R \downarrow A \quad t \notin R}{R \downarrow \forall t.A}$$

$$\frac{\forall r : (\delta, A) \in R \quad R \downarrow A}{R \vdash} \qquad \frac{R \vdash \quad R \downarrow \alpha}{R \vdash \alpha} \qquad \frac{\forall x : (\delta, A) e \in \Gamma \quad R \vdash A}{R \vdash \Gamma}$$

Figure 7.13: Formation of types and contexts

We still notice the following substitution property on types.

Proposition 7.5.1. *If $R \vdash \forall t.A$ and $R \vdash B$ then $R \vdash A[B/t]$.*

Proof. By induction on A. □

7.5.2 Rules

A typing judgment takes the form

$$R; \Gamma \vdash^{\delta} P : \alpha$$

It attributes a type α to the program P occurring at depth δ, according to region context R and variable context Γ.

Figure 7.14 introduces the rules of $\lambda^{!\S\|}_{LLT}$. The skeleton of the rules is the light linear depth system $\lambda^{!\S\|}_{LLD}$. We comment on some of the rules:

- In lam, a λ-abstraction may only take a term of result-type as argument, that is two threads in parallel are not considered an argument.
- The typing of !-terms in the rule $\mathsf{prom}_!$ is limited to values. Indeed, the type system is intended to be used with an outer-bang by-value syntax which does not allow to put bangs on non-values (see Figure 7.11).
- There exists two rules for typing parallel programs. The rule par_1 indicates that a program P_2 in parallel with a store or a thread producing a terminal value should have the type of P_2 since we might be interested in its result (note that we omit the symmetric rule for the program $(P_2 \parallel P_1)$). The rule par_2 indicates that two programs in parallel cannot reduce to a single result.

Example 7.5.2. The program of Figure 7.4 (page 126) is well-typed according to the following derivable judgment:

$$R; - \vdash^{\delta} \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, (!x)(\S x)) \parallel r \Leftarrow\ !(\lambda x.x\star) : \mathsf{Unit}$$

where $R = r : (\delta, \forall t.!((\mathsf{Unit} \multimap t) \multimap t))$. Whereas the program in Equation (7.5) (page 139) is not.

Remark 7.5.3. We easily see that a well-typed program is also well-formed.

Sequential composition We proposed in Section 7.2 to encode the sequential composition as

$$M; N = (\lambda z.\mathsf{set}(r, z) \parallel N)M$$

where r is a 'garbage collecting' region. Assuming that N is of type α, we would like to show that $M; N$ can be given type α. By using the rule par_1 we can derive

$$\frac{R; \Gamma, z : (\lambda, A) \vdash^{\delta} \mathsf{set}(r, z) : \mathsf{Unit} \qquad R; \Gamma, z : (\lambda, A) \vdash^{\delta} N : \alpha}{R; \Gamma, z : (\lambda, A) \vdash^{\delta} \mathsf{set}(r, z) \parallel N : \alpha}$$

$$\mathsf{var}\frac{R \vdash \Gamma \qquad x : (\lambda, A) \in \Gamma}{R; \Gamma \vdash^{\delta} x : A} \qquad \mathsf{unit}\frac{R \vdash \Gamma}{R; \Gamma \vdash^{\delta} \star : \mathsf{Unit}}$$

$$\mathsf{reg}\frac{R \vdash \Gamma}{R; \Gamma \vdash^{\delta} r : \mathsf{Reg}_r A}$$

$$\mathsf{lam}\frac{\mathsf{FO}(x, M) = 1 \qquad R; \Gamma, x : (\lambda, A) \vdash^{\delta} M : \alpha}{R : \Gamma \vdash^{\delta} \lambda x.M : A \multimap \alpha}$$

$$\mathsf{app}\frac{R; \Gamma \vdash^{\delta} M : A \multimap \alpha \qquad R; \Gamma \vdash^{\delta} N : A}{R; \Gamma \vdash^{\delta} MN : \alpha}$$

$$\mathsf{prom}_!\frac{\mathsf{FO}(M) \leq 1 \qquad R; \Gamma_\lambda \vdash^{\delta+1} V : A}{R; \Gamma_!, \Delta_\S, \Psi_\lambda \vdash^{\delta} !V : !A}$$

$$\mathsf{elim}_!\frac{\mathsf{FO}(x, N) \geq 1 \qquad R; \Gamma \vdash^{\delta} V : !A \qquad R; \Gamma, x : (!, A) \vdash^{\delta} M : \alpha}{R; \Gamma \vdash^{\delta} \mathsf{let}\ !x = V\ \mathsf{in}\ M : \alpha}$$

$$\mathsf{prom}_\S\frac{R; \Gamma_\lambda, \Delta_\lambda \vdash^{\delta+1} M : A}{R; \Gamma_\S, \Delta_!, \Psi_\lambda \vdash^{\delta} \S M : \S A}$$

$$\mathsf{elim}_\S\frac{\mathsf{FO}(x, N) = 1 \qquad R; \Gamma \vdash^{\delta} V : \S A \qquad R; \Gamma, x : (\S, A) \vdash^{\delta} M : \alpha}{R; \Gamma \vdash^{\delta} \mathsf{let}\ \S x = V\ \mathsf{in}\ M : \alpha}$$

$$\mathsf{forall}\frac{t \notin (R; \Gamma) \qquad R; \Gamma \vdash^{\delta} M : A}{R; \Gamma \vdash^{\delta} M : \forall t.A} \qquad \mathsf{inst}\frac{R; \Gamma \vdash^{\delta} M : \forall t.A \qquad R \vdash B}{R; \Gamma \vdash^{\delta} M : A[B/t]}$$

$$\mathsf{get}\frac{R \vdash \Gamma \qquad r : (\delta, A) \in R}{R; \Gamma \vdash^{\delta} \mathsf{get}(r) : A}$$

$$\mathsf{set}\frac{r : (\delta, A) \qquad R; \Gamma \vdash^{\delta} M : A}{R; \Gamma \vdash^{\delta} \mathsf{set}(r, M) : \mathsf{Unit}}$$

$$\mathsf{store}\frac{r : (\delta, A) \qquad R; \Gamma \vdash^{\delta} M : A}{R; \Gamma \vdash^{0} r \Leftarrow M : \mathbf{B}}$$

$$\mathsf{par}_1\frac{R; \Gamma \vdash^{\delta} P_1 : \mathsf{Unit}\ \text{or}\ P_1 = S \qquad R; \Gamma \vdash^{\delta} P_2 : \alpha}{R; \Gamma \vdash^{\delta} (P_1 \parallel P_2) : \alpha}$$

$$\mathsf{par}_2\frac{R; \Gamma \vdash^{\delta} P_i : \alpha_i}{R; \Gamma \vdash^{\delta} (P_1 \parallel P_2) : \mathbf{B}}$$

Figure 7.14: The light linear type system $\lambda^{!\S\parallel}_{LLT}$

with $r : (\delta, A) \in R$. It is then straightforward to see that the rule

$$\mathsf{seq}\,\frac{R;\Gamma \vdash^{\delta} M : A \qquad R;\Gamma \vdash^{\delta} N : \alpha}{R;\Gamma \vdash^{\delta} M;N : \alpha}$$

can be derived.

7.5.3 Properties

The light linear type system $\lambda^{!\S\|}_{LLT}$ enjoys the subject reduction and progress properties for the outer-bang by-value reduction $\longrightarrow_{\mathsf{obv}}$. The usual weakening and substitution properties need to be established first.

Lemma 7.5.4 (Weakening)**.** *If $R;\Gamma \vdash^{\delta} P : \alpha$ and $R, R' \vdash \Gamma, \Gamma'$ then $R, R';\Gamma, \Gamma' \vdash^{\delta} P : \alpha$.*

Proof. By induction on the typing of P. □

Lemma 7.5.5 (Substitution)**.**

1. *If $R;\Gamma, x : (\lambda, A) \vdash^{\delta} M : B$ and $R;\Gamma \vdash^{\delta} V : A$ then $R;\Gamma \vdash^{\delta} M[V/x] : B$.*
2. *If $R;\Gamma, x : (\S, A) \vdash^{\delta} M : B$ and $R;\Gamma \vdash^{\delta} \S V : \S A$ then $R;\Gamma \vdash^{\delta} M[V/x] : B$.*
3. *If $R;\Gamma, x : (!, A) \vdash^{\delta} M : B$ and $R;\Gamma \vdash^{\delta} !V : !A$ then $R;\Gamma \vdash^{\delta} M[V/x] : B$.*

Proof. By induction on the typing of M. □

Proposition 7.5.6 (Subject reduction)**.** *If $R;\Gamma \vdash^{\delta} P : \alpha$ and $P \longrightarrow_{\mathsf{obv}} P'$ then $R;\Gamma \vdash^{\delta} P' : \alpha$.*

Proof. By case analysis on the reduction rules and Lemma 7.5.4 and 7.5.5. □

Proposition 7.5.7 (Progress)**.** *Suppose P is a closed typable outer-bang by-value program which cannot reduce. Then P is structurally equivalent to a program*

$$M_1 \parallel \cdots \parallel M_m \parallel S_1 \parallel \cdots \parallel S_n \quad m, n \geq 0$$

where M_i is either a value or can only be decomposed as a term $F_{\mathsf{v}}[\mathsf{get}(r)]$ such that no value is associated with the region r in the stores $S_1, \ldots, S_n$.

Proof. The proof is similar to that of $\lambda^{!\|}_{EAT}$ (Proposition 5.4.8) and also crucially relies on the fact that $!M$ is not typable if M is not a value. Indeed this term locks the outer-bang by-value reduction. □

7.6 Expressivity

In this section we evaluate the expressivity of $\lambda^{!\S\|}_{LLT}$. First, in Section 7.6.1 we recall briefly that every polynomial time function is representable in $\lambda^{!\S\|}_{LLT}$. Then, in Section 7.6.2 we test the programming capabilities offered by the type

system (iteration of side effects) and highlight the loss of expressivity with respect to the elementary case. To conclude, we draw a quick comparison with the expressive power offered by *soft λ-calculi*.

7.6.1 Completeness

The representation of polynomial functions relies on the representation of binary natural numbers whose encoding is given in Figure 7.15.

$$\begin{array}{rcl} \mathsf{BNat} & = & \forall t.!(t \multimap t) \multimap !(t \multimap t) \multimap \S(t \multimap t) \\ \\ \text{for } w = i_0 \dots i_n \in \{0,1\}^*, & & \\ \\ \quad \overline{w} & : & \mathsf{BNat} \\ \quad \overline{w} & = & \lambda^! x_0.\lambda x_1^!.\S(\lambda z.x_{i_0}(\dots(x_{i_n} z))) \end{array}$$

Figure 7.15: Representation of binary natural numbers

The precise notion of representation is spelled out in the following definitions (we omit region contexts since we only rely on functional terms).

Definition 7.6.1 (Binary natural number representation)**.** Let $- \vdash^\delta M : \S^p\mathsf{BNat}$ for some $\delta, p \in \mathbb{N}$. We say M *represents* $w \in \{0,1\}^*$, written $M \Vdash w$, if $M \longrightarrow^* \S^p\overline{w}$.

Definition 7.6.2 (Function representation)**.** Let $- \vdash^\delta F : \mathsf{BNat} \multimap \S^d\mathsf{BNat}$ where $\delta, d \in \mathbb{N}$ and $f : \{0,1\}^* \to \{0,1\}^*$. We say F *represents* f, written $F \Vdash f$, if for any M and $w \in \{0,1\}^*$ such that $- \vdash^\delta M : \mathsf{BNat}$ and $M \Vdash w$, $FM \Vdash f(w)$.

The following theorem is a restatement of Girard [Gir98] and Asperti [Asp98].

Theorem 7.6.3 (Polynomial completeness)**.** *Every function $f : \{0,1\}^* \to \{0,1\}^*$ which can be computed by a Turing machine in time bounded by a polynomial of degree d can be represented by a term of type $\mathsf{BNat} \multimap \S^d\mathsf{BNat}$.*

Remark 7.6.4. Note that the type of the representing term depends on the degree of the polynomial. This lack of uniformity complicates the programming and implies to go through various typing coercions.

7.6.2 Polynomial programming

We now examine the programming capabilities offered by $\lambda^{!\S\|}_{LLT}$. We first quickly show that the operational semantics of references can be implemented like in the elementary type system $\lambda^{!\|}_{EAT}$. Then, we show how to program the iteration of operations producing side effects on an inductive data structure, and remark

that several restrictions arise compared to the elementary case. Eventually, we show that we can still exchange data between regions of different depths, but in a restricted way.

Implementing references

The primitive read operation (get) of $\lambda^{!\S\|}_{LLD}$ is to consume the value from the store, that is the operational semantics of communication channels. Copying values from the store, in the style of references, can be implemented and typed the same way as in $\lambda^{!\|}_{EAT}$, by using the *consume-and-rewrite* mechanism. See Section 5.5.2 for details.

Iteration with side effects

We want to program the concurrent iteration of operations producing side effects on an inductive data structure. In order to compare with the expressive power of the elementary case, we will try to write the same programming example.

We assume the language has dynamic locations rather than constant regions names. For a more detailed correspondence between locations and regions see Section 2.4.

We define the representation of natural number, some arithmetic functions and lists in Figure 7.16 where we abbreviate $\lambda x.\mathsf{let}\ \dagger x = x\ \mathsf{in}\ M$ by $\lambda^{\dagger}x.M$. We see that the multiplying function mult has a type $!\mathsf{Nat} \multimap \mathsf{Nat} \multimap \S\mathsf{Nat}$ whereas it had type $\mathsf{Nat} \multimap \mathsf{Nat} \multimap \mathsf{Nat}$ in $\lambda^{!\|}_{EAT}$ (see the encodings in Figure 5.8). Therefore, it is no more possible to iterate *e.g.* the function

$$\mathsf{mult}_2 = (\lambda x.\mathsf{mult}\ !\overline{2}\ x)$$

since it has type $\mathsf{Nat} \multimap \S\mathsf{Nat}$ and the iterator int_it may only iterate functions with a type of the shape $A \multimap A$. This is not surprising because it prevents to represent the exponential function $f(n) = 2^n$. We will see that it has some consequences on programming.

In the programming example of $\lambda^{!\|}_{EAT}$, we provide a function update that takes a memory location as argument and multiply by 2 the number stored at that location. In $\lambda^{!\S\|}_{LLT}$, the type of mult_2 does not allow to do this. Indeed, once we have multiplied the number, its type has gained a paragraph modality but the type of the memory location did not change. Instead, the update function can use the encoding add to add 2 to the content of the location:

$$r : (3, !\mathsf{Nat}); - \vdash^2 \mathsf{update} : !\mathsf{Reg}_r!\mathsf{Nat} \multimap \S\mathsf{Unit} \multimap \S\mathsf{Unit}$$
$$\mathsf{update} = \lambda^{!}x.\lambda^{\S}z.\S(\mathsf{set}(x, \mathsf{let}\ !y = \mathsf{get}(x)\ \mathsf{in}\ !(\mathsf{add}\ \overline{2}\ y)) \parallel z)$$

Another difference with the elementary update function is that here the second argument z is to be garbage collected by the rule (gc).

$$
\begin{array}{rcl}
\mathsf{Nat} & = & \forall t.!(t \multimap t) \multimap \S(t \multimap t) \\
\overline{n} & : & \mathsf{Nat} \\
\overline{n} & = & \lambda^! f.\S(\lambda x.\underbrace{f(\ldots(f}_{n \text{ times}} x))) \\
 & & \\
\mathsf{int\_it} & : & \forall t.\mathsf{Nat} \multimap !(t \multimap t) \multimap \S t \multimap \S t \\
\mathsf{int\_it} & = & \lambda n.\lambda f.\lambda^{\S} x.\mathsf{let}\ \S y = nf\ \mathsf{in}\ \S(yx) \\
 & & \\
\mathsf{add} & : & \mathsf{Nat} \multimap \mathsf{Nat} \multimap \mathsf{Nat} \\
\mathsf{add} & = & \lambda m.\lambda n.\lambda^! f.\mathsf{let}\ \S y = m!f\ \mathsf{in} \\
 & & \mathsf{let}\ \S z = n!f\ \mathsf{in}\ \S(\lambda x.y(zx)) \\
 & & \\
\mathsf{mult} & : & !\mathsf{Nat} \multimap \mathsf{Nat} \multimap \S\mathsf{Nat} \\
\mathsf{mult} & = & \lambda m.\mathsf{let}\ !m = m\ \mathsf{in}\ \lambda n.\mathsf{int\_it}\ n\ !(\mathsf{add}\ m)\ \S\overline{0} \\
 & & \\
\mathsf{List}\ A & = & \forall t.!(A \multimap t \multimap t) \multimap \S(t \multimap t) \\
[u_1, \ldots, u_n] & : & \mathsf{List}\ A \\
[u_1, \ldots, u_n] & = & \lambda f^!.\S(\lambda x.fu_1(fu_2 \ldots (fu_n x))) \\
 & & \\
\mathsf{foldr} & : & \forall u.\forall t.!(u \multimap t \multimap t) \multimap \mathsf{List}\ u \multimap \S t \multimap \S t \\
\mathsf{foldr} & = & \lambda f.\lambda l.\lambda^{\S} x.\mathsf{let}\ \S y = lf\ \mathsf{in}\ \S(yx)
\end{array}
$$

Figure 7.16: Encodings of natural number, arithmetic functions and lists

We then define the program run that iterates the function update over a list $[!x, !y, !z]$ of 3 memory locations:

$$r : (3, !\mathsf{Nat}); - \vdash^{1} \mathsf{run} : §§\mathsf{Unit}$$
$$\mathsf{run} = \mathsf{foldr}\ !\mathsf{update}\ [!x, !y, !z]\ §§\star$$

The source code of run is the same as in $\lambda^{!\|}_{EAT}$, it reduces as expected when put in parallel with some store assignments:

$$\mathsf{run} \parallel x \Leftarrow !\overline{m} \parallel y \Leftarrow !\overline{n} \parallel z \Leftarrow !\overline{p}$$
$$\longrightarrow^{*} §§\star \parallel x \Leftarrow !\overline{2+m} \parallel y \Leftarrow !\overline{2+n} \parallel z \Leftarrow !\overline{2+p}$$

To write a program of three threads where each thread concurrently increments the numerals pointed by the memory locations of the list, we proceed as in the elementary case; the minor difference being that the unit value is of type §§Unit instead of !!Unit. Here is the function gen_threads taking a functional f and a value x as arguments and generating three threads where x is applied to f:

$$r : (3, !\mathsf{Nat}); - \vdash^{0} \mathsf{gen\_threads} : \forall t.\forall t'.!(t \multimap t') \multimap !t \multimap \mathbf{B}$$
$$\mathsf{gen\_threads} = \lambda^{!} f.\lambda^{!} x.§(fx) \parallel §(fx) \parallel §(fx)$$

We define the functional F like run but parametric in the list:

$$r : (3, !\mathsf{Nat}); - \vdash^{1} \mathsf{F} : \mathsf{List}\ !\mathsf{Reg}_{r}!\mathsf{Nat} \multimap §§\mathsf{Unit}$$
$$\mathsf{F} = \lambda l.\mathsf{foldr}\ !\mathsf{update}\ l\ §§\star$$

Finally, the concurrent iteration is defined in run_threads:

$$r : (3, !\mathsf{Nat}); - \vdash^{0} \mathsf{run\_threads} : \mathbf{B}$$
$$\mathsf{run\_threads} = \mathsf{gen\_threads}\ !\mathsf{F}\ ![!x, !y, !z]$$

The program is well-typed and has depth 4 with side effects occurring at depth 3. Different thread interleavings are possible and here is one of them:

$$\mathsf{run\_threads} \parallel x \Leftarrow !\overline{m} \parallel y \Leftarrow !\overline{n} \parallel z \Leftarrow !\overline{p}$$
$$\longrightarrow^{*} §§§\star \parallel x \Leftarrow !\overline{6+m} \parallel y \Leftarrow !\overline{6+n} \parallel z \Leftarrow !\overline{6+p}$$

Supposing that the size of this program is n, every thread interleaving must run in at most n^{16} steps by Theorem 7.4.5.

Remark 7.6.5. As in the elementary case, the Church representation of data types requires to reduce under binders. See Remark 5.5.4.

Exchanging data between regions

In $\lambda^{!§\|}_{LLT}$, it is possible to exchange data between regions of different depth. We recall that in $\lambda^{!\|}_{EAT}$, the trick is to add or remove bangs on the exchanged value so that its depth is preserved (see Section 5.5.2). However, as in many program

of the elementary (concurrent) λ-calculus, we have to reduce under bangs, and this is not compatible with the outer-bang reduction. Therefore, in $\lambda^{!\S\|}_{LLT}$ the trick is to add or remove paragraphs so that the depth of the exchanged value is preserved. The direct consequence is that one cannot exchange values that are duplicable (of the shape $!V$) between regions of different depth, and this is a notable restriction with respect to the elementary case.

In the following, we recall the encodings to exchange data between regions of different depth. They are as in the elementary case, except that we only manipulate paragraphs instead of bangs. Let us consider an untyped region context

$$R = r : \delta, r' : \delta'$$

with two regions r and r' such that $\delta < \delta'$. We first consider the case where we want to transfer a value from r to r' and then the converse.

1. Intuitively, if we want to transfer a value V which is stored at depth δ in r to a deeper region r', V must be of the shape $\S^{\delta'-\delta}V'$ such that the depth of V' can be preserved. Let us define the following abbreviation by induction on $i > 1$:

$$\begin{array}{rcll} \mathsf{let}\ \S^1 x = M\ \mathsf{in}\ \S^1 N & = & \mathsf{let}\ \S x = M\ \mathsf{in}\ \S N & \\ \mathsf{let}\ \S^i x = M\ \mathsf{in}\ \S^i N & = & \mathsf{let}\ \S x = M\ \mathsf{in}\ \S(\mathsf{let}\ \S^{i-1} x = x\ \mathsf{in}\ \S^{i-1} N) & i > 1 \end{array}$$

   The program

$$P = \S^{\delta}(\mathsf{let}\ \S^{\delta'-\delta} z = \mathsf{get}(r)\ \mathsf{in}\ \S^{\delta'-\delta}\mathsf{set}(r', z))$$

   is well-formed with region context R because the read on r occurs at depth δ and the write on r' occurs at depth δ'. The transfer then happens as follows:

$$P \parallel r \Leftarrow \S^{\delta'-\delta}V' \longrightarrow^{+} \S^{\delta'}\star \parallel r' \Leftarrow V'$$

2. To transfer a value V from r' to the shallower region r take the program

$$Q = \S^{\delta}((\lambda z.\mathsf{set}(r, z))\S^{\delta'-\delta}\mathsf{get}(r'))$$

   It is well-formed with region context R because the write on r occurs at depth δ and the read on r' occurs at depth δ'. The transfer happens as follows:

$$Q \parallel r' \Leftarrow V \longrightarrow^{+} \S^{\delta}\star \parallel r \Leftarrow \S^{\delta'-\delta}V$$

7.6.3 Soft linear logic

The polynomial calculi that we proposed are inspired from the light affine λ-calculus of Terui [Ter07], which is itself designed from the light logic **LAL** [Asp98]. There exists another light logic which corresponds to polynomial time, namely

SAL (originally **SLL** [Laf04]). In this last section, we quickly justify why we settled on calculi based on **LAL** instead of **SAL**.

As other light logics, **SAL** relies on the notion of depth to control the complexity of reduction. However, while **LAL** uses two modalities bang and paragraph, **SAL** is built with a single modality bang which is sufficient to ensure both soundness and completeness with respect to polynomial time. In particular, the combinatorial proof of soundness is quite simple and works for an arbitrary reduction strategy while **LAL** relies on a non-trivial transformation of reduction sequences into shallow-first ones (see Chapter 6).

Also, we could think that the less number of modalities we have to manipulate, the more easier it is to program. It turns out that the second modality paragraph of **LAL** allows to hide some complexity details which have to appear in **SAL**. This can be exemplified at the level of types. We recall that in $\lambda^{!\S}_{LLT}$, a uniform type Nat is given to the Church encoding of natural numbers and that the addition and multiplication functions are given the following types (see Figure 7.16):

$$\mathsf{add} : \mathsf{Nat} \multimap \mathsf{Nat} \multimap \mathsf{Nat}$$
$$\mathsf{mult} : !\mathsf{Nat} \multimap \mathsf{Nat} \multimap \S\mathsf{Nat}$$

In a soft calculus (see [GR09]), the type of Church natural numbers Nat_i is not uniform, it is indexed by a number $i \in \mathbb{N}_{>0}$ and the addition and multiplication functions have the following types:

$$\mathsf{add} : \mathsf{Nat}_i \multimap \mathsf{Nat}_j \multimap \mathsf{Nat}_{max(i,j)+1}$$
$$\mathsf{mult} : \mathsf{Nat}_j \multimap !^j\mathsf{Nat}_i \multimap \mathsf{Nat}_{i+j}$$

This lack of uniformity complicates the programming. In particular, if we assume a soft calculus with regions, we cannot update the content of a region with one of these functions since the type of the region is fixed. In $\lambda^{!\S\|}_{LLT}$, we are at least able to define a function update that applies the function add on a region content (see Section 7.6.2).

Considering that we wanted to favor expressivity of the type system over simplicity of the soundness proof, we decided to base our polynomial calculi on **LAL**. Yet, we are conscious that the programming style offered by **LAL** is highly constrained by the stratification by depth levels.

Part III

Quantitative Realizability

Chapter 8

Quantitative realizability for an imperative λ-calculus

This chapter is joint work with Aloïs Brunel.

In the unique chapter of this part, we propose a semantic method to control the complexity of higher-order imperative programs. The framework of *quantitative realizability* has been proposed by U. Dal Lago and M. Hofmann [LH11] to give semantic proofs of complexity soundness of various light logics. Our contribution is to extend quantitative realizability to higher-order imperative programs, taking a *light affine type system* as case study. By proving that the type system is adequate with the realizability model, we prove that every typable program terminates in polynomial time under a call-by-value strategy.

Contrary to the combinatorial methods presented in the previous part, quantitative realizability allows to give uniform proofs of complexity soundness of various type systems. They key point is to parametrize the interpretation by a so-called *quantitative monoid* which depends on the underlying type system and whose elements contain information on the running time of programs. Moreover, our interpretation is based on bi-orthogonality (*à la Krivine* [Kri09]), following A. Brunel's extension of quantitative realizability to a *classical* setting [Bru12]. This should ease the addition of *e.g.* control operators. Interestingly, the proposed realizability framework allows to deal with affine terms (that can bind 0 occurrences of free variables), but the important drawback is that it currently does not scale to multi-threaded programs. Table 8.1 summarizes the comparison between the combinatorial proofs and the realizability ones.

We want to emphasize on the fact that the adequacy of the light affine type system with respect to the realizability interpretation is partly due to the fact that

	Combinatorial	Realizability
Untyped	√	
Multi-threading	√	
Modularity		√
Affinity		√

Table 8.1: Comparison of proof features in the polynomial case

the type system is built on the depth system $\lambda^{!\S\|}_{LLD}$ provided in Chapter 7. Realizability in the presence of imperative side effects is usually difficult and raises circularity issues; here, our notion of stratification of regions by depth levels, guaranteed by the type system, allows to give a well-defined interpretation.

Outline The chapter is organized as follows. In Section 8.1, we present an imperative λ-calculus together with a light affine type system which is the sequential subset of $\lambda^{!\S\|}_{LLT}$. In Section 8.2, we present the realizability model. We start in Section 8.2.1 by introducing the quantitative monoid whose elements contain information on the execution time of programs. Then, in Section 8.2.2 we present a notion of orthogonality which takes stores into account . In Section 8.2.3, we define the interpretation and prove that the type system is adequate to the interpretation. This allows us to associate every well-typed program with an element of the quantitative monoid that guarantees its termination in polynomial time. Finally, in Section 8.3 we discuss the pros and cons of the realizability method with respect to the combinatorial ones, with respect to a monadic translation, and we highlight connections with related work.

8.1 A light affine imperative λ-calculus

In this section, we first present the syntax and semantics of an imperative λ-calculus which is based on the sequential subset of $\lambda^{!\S\|}$ (Section 7.1). Then, we introduce a light affine type system which is an affine variant of the sequential subset of $\lambda^{!\S\|}_{LLT}$ (Section 7.5).

8.1.1 An imperative λ-calculus

The syntax of terms of the imperative λ-calculus ($\lambda^{!\S\mathsf{Reg}}$) is given in Figure 8.1. It is like the syntax of the call-by-value $\lambda^{!\S\|}$ (Figure 7.11), except that there is no parallelism constructs. We denote the set of values by $\mathbb{V}$ and the set of terms by Λ. We consider terms up to α-renaming.

Contrary to the languages presented previously, we present the operational semantics of $\lambda^{!\S\mathsf{Reg}}$ by an abstract machine. This presentation allows for a clear

-values	V	::=	$x \mid \star \mid r \mid \lambda x.M \mid !V \mid \S V$
-terms	M	::=	$V \mid MM \mid \S M$
			$\mathsf{let}\ !x = V\ \mathsf{in}\ M \mid \mathsf{let}\ \S x = V\ \mathsf{in}\ M$
			$\mathsf{get}(r) \mid \mathsf{set}(r, V)$

Figure 8.1: Syntax of $\lambda^{!\S\mathsf{Reg}}$

symmetrization between terms and contexts, and this will be very useful to define an orthogonality relation later on in Section 8.2.2.

The configurations of the abstract machine, which evaluates programs with a left-to-right call-by-value strategy, are described in Figure 8.2. A configuration

-stacks	π	::=	$\diamond \mid V \odot \pi \mid M \cdot \pi \mid \S \cdot \pi$
-stores	S	::=	$r \Leftarrow V \mid (S \uplus S)$
-configurations	C	::=	$\langle M, \pi, S \rangle$

Figure 8.2: Configuration of the abstract machine of $\lambda^{!\S\mathsf{Reg}}$

of the machine is a triplet of a term, a stack and a store. We denote the set of stacks by Π and the set of stores by Σ. As usual, a store is a multiset of assignments (note that we use the operator '$\uplus$' instead of '$\|$' to combine store assignments).

A stack keeps track of the the rest of the program to be evaluated. The empty stack is denoted by '$\diamond$' and there are the following operations:

- $(V \odot \pi)$ is to push a function V on the stack π and evaluate its argument on its right;
- $(M \cdot \pi)$ is to push an argument M on the stack π and evaluate the calling function on its left;
- $(\S \cdot \pi)$ is to push a paragraph constructor on the stack π and evaluate what was under this paragraph.

Remark 8.1.1. The reduction is outer-bang since there is no stack of the shape $(! \cdot \pi)$. The reason is extensively discussed in Section 7.3.1.

To give some more intuitions, a stack may be seen as the finite composition of elementary evaluation contexts. The following translation from stacks to outer-bang by-value evaluation contexts (see Figure 7.11) illustrates the correspondence:

$$\begin{array}{rcl} \underline{\diamond} & = & [\cdot] \\ \underline{V \odot \pi} & = & \underline{\pi}[V[\cdot]] \\ \underline{M \cdot \pi} & = & \underline{\pi}[[\cdot]M] \\ \underline{\S \cdot \pi} & = & \underline{\pi}[\S[\cdot]] \end{array}$$

A configuration of the abstract machine is executed according to the rules of Figure 8.3. As usual, reading a region amounts to *consume* a value from the

$$
\begin{array}{rcll}
\langle MN, E, S\rangle & \longrightarrow & \langle M, N \cdot E, S\rangle & \\
\langle V, M \cdot E, S\rangle & \longrightarrow & \langle M, V \odot E, S\rangle & \\
\langle V, \lambda x.M \odot E, S\rangle & \longrightarrow & \langle M[V/x], E, S\rangle & \\
\langle \S M, E, S\rangle & \longrightarrow & \langle M, \S \cdot E, S\rangle & \text{if } M \notin \mathbb{V} \\
\langle V, \S \cdot E, S\rangle & \longrightarrow & \langle \S V, E, S\rangle & \\
\langle \mathsf{let}\ !x = !V\ \mathsf{in}\ M, E, S\rangle & \longrightarrow & \langle M[V/x], E, S\rangle & \\
\langle \mathsf{let}\ \S x = \S V\ \mathsf{in}\ M, E, S\rangle & \longrightarrow & \langle M[V/x], E, S\rangle & \\
\langle \mathsf{get}(r), E, r \Leftarrow V \uplus S\rangle & \longrightarrow & \langle V, E, S\rangle & \\
\langle \mathsf{set}(r, V), E, S\rangle & \longrightarrow & \langle \star, E, r \Leftarrow V \uplus S\rangle &
\end{array}
$$

Figure 8.3: Reduction rules of the abstract machine

store and writing to a region amounts to *add* the value to the store.

Example 8.1.2. Here is an example of reduction. Take the function

$$F = \lambda x.\mathsf{let}\ !y = x\ \mathsf{in}\ \mathsf{set}(r, \S y)$$

that writes its argument to region r. It can be used to transfer a value from another region r' as follows:

$$
\begin{array}{rl}
 & \langle F\mathsf{get}(r'), \diamond, r' \Leftarrow !V\rangle \\
\longrightarrow & \langle F, \mathsf{get}(r') \cdot \diamond, r' \Leftarrow !V\rangle \\
\longrightarrow & \langle \mathsf{get}(r'), F \odot \diamond, r' \Leftarrow !V\rangle \\
\longrightarrow & \langle !V, F \odot \diamond, \emptyset\rangle \\
\longrightarrow & \langle \mathsf{let}\ !y = !V\ \mathsf{in}\ \mathsf{set}(r, \S y), \diamond, \emptyset\rangle \\
\longrightarrow & \langle \mathsf{set}(r, \S V), \diamond, \emptyset\rangle \\
\longrightarrow & \langle \star, \diamond, r \Leftarrow \S V\rangle
\end{array}
$$

The following notation will be useful to measure the execution time of programs.

Definition 8.1.3 (Time)**.** We write $C\Downarrow^n$ when the configuration C terminates in n steps of the abstract machine.

8.1.2 A light affine type system

In this section, we introduce the light affine type system $\lambda^{!\S\mathsf{Reg}}_{\mathsf{LAT}}$. It differs from the light linear type system $\lambda^{!\S\|}_{LLT}$ (Section 7.5) in several ways:

- There are explicit rules for the weakening and contraction of variables; this allows for a simpler presentation of the latter adequacy theorem (Theorem 8.2.23).

- It is affine in the sense that it allows to bind 0 occurrences of free variables and thus to discard data without garbage collecting regions (see Section 7.5.2).
- It has no polymorphic types, which keeps the realizability interpretation simpler. Note that we therefore loose the completeness property (Theorem 7.6.3).

Types and contexts

The syntax of types and contexts is given in Figure 8.4. Comparing to the types

-types	A	$::=$	$\mathsf{Unit} \mid A \multimap A \mid !A \mid \S A \mid \mathsf{Reg}_r A$
-variable contexts	Γ	$::=$	$x_1 : (u_1, A_1), \ldots, x_n : (u_n, A_n)$
-region contexts	R	$::=$	$r_1 : (\delta_1, A_1), \ldots, r_n : (\delta_n, A_n)$

Figure 8.4: Syntax of types and contexts of $\lambda^{!\S\mathsf{Reg}}_{\mathsf{LAT}}$

and contexts of $\lambda^{!\S\parallel}_{LLT}$ (Figure 7.12), we note the absence of the behavior type $\mathbf{B}$ that was used for parallelism and the polymorphic type $\forall t.A$

As usual, we state in Figure 8.5 when a type is well-formed with respect to a region context.

$$\frac{}{R \downarrow t} \qquad \frac{}{R \downarrow \mathsf{Unit}} \qquad \frac{R \downarrow A \quad R \downarrow \alpha}{R \downarrow (A \multimap \alpha)}$$

$$\frac{R \downarrow A}{R \downarrow \dagger A} \qquad \frac{r : (\delta, A) \in R}{R \downarrow \mathsf{Reg}_r A}$$

$$\frac{\forall r : (\delta, A) \in R \quad R \downarrow A}{R \vdash} \qquad \frac{R \vdash \quad R \downarrow \alpha}{R \vdash \alpha} \qquad \frac{\forall x : (\delta, A) \in \Gamma \quad R \vdash A}{R \vdash \Gamma}$$

Figure 8.5: Formation of types and contexts of $\lambda^{!\S\mathsf{Reg}}_{\mathsf{LAT}}$

Typing rules

A typing judgment takes the form

$$R; \Gamma \vdash^{\delta} M : A$$

where δ is a depth. It It should entail the following:

- if $x : (\lambda, A) \in \Gamma$ then x may occur at most once in M, and it must not be in the scope of a modal constructor;
- if $x : (\S, A) \in \Gamma$ then x may occur at most once in M, and it must be in the scope of a paragraph constructor;,
- if $x : (!, A) \in \Gamma$ then x may occur arbitrarily many times in M, and it must be in the scope of a modal constructor;
- if $r : (\delta', A) \in R$ then $\mathsf{get}(r)$ and $\mathsf{set}(r)$ occur at depth δ' in $\dagger^{\delta} M$.

The rules are given in Figure 8.6.

$$\mathsf{var}\ \frac{R \vdash}{R; x : (\lambda, A) \vdash^{\delta} x : A} \qquad \mathsf{unit}\ \frac{R \vdash}{R; - \vdash^{\delta} \star : \mathsf{Unit}} \qquad \mathsf{reg}\ \frac{R \vdash \quad r : (\delta, A) \in R}{R; - \vdash^{\delta} r : \mathsf{Reg}_r A}$$

$$\mathsf{contr}\ \frac{R; \Gamma, x : (!, A), y : (!, A) \vdash^{\delta} M : B}{R; \Gamma, z : (!, A) \vdash^{\delta} M[z/x, z/y] : B} \qquad \mathsf{weak}\ \frac{R; \Gamma \vdash^{\delta} M : B \quad R \vdash \Gamma, x : (u, A)}{R; \Gamma, x : (u, A) \vdash^{\delta} M : B}$$

$$\mathsf{lam}\ \frac{R; \Gamma, x : (\lambda, A) \vdash^{\delta} M : B}{R; \Gamma \vdash^{\delta} \lambda x.M : A \multimap B} \qquad \mathsf{app}\ \frac{R; \Gamma \vdash^{\delta} M_1 : A \multimap B \quad R; \Delta \vdash^{\delta} M_2 : A}{R; \Gamma, \Delta \vdash^{\delta} M_1 M_2 : B}$$

$$\mathsf{prom}_!\ \frac{R; x : (\lambda, A) \vdash^{\delta+1} V : A}{R; x : (!, A) \vdash^{\delta} !V : !A} \qquad \mathsf{elim}_!\ \frac{R; \Gamma \vdash^{\delta} V : !A \quad R; \Delta, x : (!, A) \vdash^{\delta} M : B}{R; \Gamma, \Delta \vdash^{\delta} \mathsf{let}\ !x = V\ \mathsf{in}\ M : B}$$

$$\mathsf{prom}_{\S}\ \frac{R; \Gamma_\lambda, \Delta_\lambda \vdash^{\delta+1} M : A}{R; \Gamma_{\S}, \Delta_! \vdash^{\delta} \S M : \S A} \qquad \mathsf{elim}_{\S}\ \frac{R; \Gamma \vdash^{\delta} V : \S A \quad R; \Delta, x : (\S, A) \vdash^{\delta} M : B}{R; \Gamma, \Delta \vdash^{\delta} \mathsf{let}\ \S x = V\ \mathsf{in}\ M : B}$$

$$\mathsf{get}\ \frac{R; - \vdash^{\delta} r : \mathsf{Reg}_r A}{R; - \vdash^{\delta} \mathsf{get}(r) : A}$$

$$\mathsf{set}\ \frac{R; - \vdash^{\delta} r : \mathsf{Reg}_r A \quad R; \Gamma \vdash^{\delta} V : A}{R; \Gamma \vdash^{\delta} \mathsf{set}(r, V) : \mathsf{Unit}} \qquad \mathsf{store}\ \frac{R; - \vdash^{\delta} r : \mathsf{Reg}_r A \quad R; \Gamma \vdash^{\delta} V : A}{R; \Gamma \vdash^{0} r \Leftarrow V : \mathsf{Unit}}$$

Figure 8.6: The light affine type system $\lambda^{!\S\mathsf{Reg}}_{\mathsf{LAT}}$

Remark 8.1.4. In binary rules, we implicitly require that the contexts Γ and Δ are disjoints. There are explicit rules weak and contr for the weakening and contraction of variables. In particular, the rule contr may only be applied to variable hypotheses with usage '!' and this has the following consequences:

- in lam, $\mathsf{FO}(x, M) \leq 1$
- in $\mathsf{prom}_!$, $\mathsf{FO}(M) \leq 1$

- in $\mathsf{elim}_{\S}$, $\mathsf{FO}(x, M) \leq 1$

The above predicates which are explicitly given (in strictly linear forms) in the typing rules of $\lambda^{!\S\|}_{LLT}$ are here implicitly guaranteed by the rules of contraction and weakening. Therefore the duplication and erasure points are explicit in typing derivation and this simplifies the presentation of the adequacy theorem (Theorem 8.2.23).

We introduce a notion of well-typing that will ensure that our realizability interpretation can be defined inductively.

Definition 8.1.5 (Well-typing)**.** We say that a program M is *well-typed* if a judgment $R; \Gamma \vdash^{\delta} M : A$ can be derived for some R, Γ, δ such that if $r : (\delta', B) \in R$, then $B = \S C$.

Remark 8.1.6. The definition of well-typing is slightly more constrained than in $\lambda^{!\S\|}_{LLT}$ since region types have to be guarded by a modality. This modality is necessary to guarantee that the interpretation is well-founded. Moreover, this modality has to be a paragraph for quantitative reasons. All this will become clear when defining the interpretation in Section 8.2.3. On the other hand, we discuss the induced loss of expressivity with respect to $\lambda^{!\S\|}_{LLT}$ in Section 8.3.

We conclude this section by claiming that the subject reduction and progress properties can be adapted from $\lambda^{!\S\|}_{LLT}$ in a straightforward way (see Proposition 7.5.6 and 7.5.7).

8.2 Quantitative realizability

In this section, we present a quantitative realizability model of the light affine type system $\lambda^{!\S\mathsf{Reg}}_{\mathsf{LAT}}$ by means of bi-orthogonality *à la Krivine* [Kri09]. At the end, we prove that the type system is adequate to the interpretation and this allows us to associate every well-typed program with an element of a *quantitative monoid* that guarantees its termination in polynomial time.

The realizability method, which was introduced by Kleene [Kle73], has been used to give interpretations of many computational systems. Recently, U. Dal Lago and M. Hofmann [LH10, LH11] extended realizability with quantitative information in order to model various light logics. Their idea is to take programs running in bounded time as realizers, where bounds are represented by elements of a *resource monoid*. Later on, A. Brunel showed [Bru12] how this quantitative extension can be adapted to a framework based on bi-orthogonality, namely Krivine's classical realizability [Kri09].

Here is an overview of our model:

- The model is *quantitative* which means that the type interpretation $\underline{A}$ is a set of pairs (M, p) where M is a term and p is a weight which is an element

of a so-called *quantitative monoid* that gives information on the running time of M. We present a particular instance of quantitative monoid in Section 8.2.1.

- The type interpretation $\underline{A}$ is defined as the *orthogonal* of a set of pairs (π, f) where π is a stack and f is a function on the light monoid. Besides, the orthogonality relation is parametrized by a set of pairs (S, s) where S is a store and s an element of the light monoid. This means roughly that $\underline{A}$ is the set of all (M, p) such that the configuration $\langle M, \pi, S\rangle$ terminates in a number of steps that depends on $f(p + s)$, where $+$ is the monoid operation. The orthogonality relation is presented in Section 8.2.2.
- The actual type interpretation $\underline{\vdash^{\delta} A}$ is *indexed* by a depth level δ which allows us to give an inductive interpretation in the presence of region types, which usually entail circularity issues. The indexed interpretation of types is given in Section 8.2.3.

Having defined the model, in Section 8.2.4 we derive a theorem of quantitative adequacy which gives us an automatic way to infer an element of the quantitative monoid from a well-typed program. By analyzing the shape of this element we conclude that the program runs in polynomial time.

8.2.1 The light monoid

The quantitative realizability framework [LH11] is parametrized by a *resource monoid* whose elements contain information on the execution time of programs. The resource monoid should have properties that match the structure of the logic that is intended to be interpreted, while the realizability structure does not change. In his work on quantitative classical realizability [Bru12], Brunel introduced *quantitative monoids* as a generalization and simplification of resource monoids. In this section, we present the *light monoid* which is a particular instance of quantitative monoid that matches the structure of $\lambda^{!\S\mathsf{Reg}}_{\mathsf{LAT}}$.

Let us introduce the light monoid step by step. The general intuition is that a program will be associated to an element p (a weight) of the monoid which gives information on its execution time. We should have a way to associate a weight to the interaction of two programs and a way to compute the concrete quantity associated to a weight. For this, we need a *quantitative monoid*.

Definition 8.2.1 (Quantitative monoid)**.** A *quantitative monoid* is a structure $(\mathcal{M}, +, \mathbf{0}, \mathbf{1}, \leq, \|\cdot\|)$ where:

- $(\mathcal{M}, +, \mathbf{0})$ is a monoid such that:
  - $(\mathcal{M}, \leq)$ is a preordered set;
  - $\forall p, q, r, s \in \mathcal{M}$ such that $p \leq q$ and $r \leq s$, we have $p + r \leq q + s$
- $\|\cdot\| : \mathcal{M} \to \mathbb{N}$ is a function such that:

  - for every $p, q \in \mathcal{M}$, we have $\|p\| + \|q\| \leq \|p + q\|$;
  - if $p \leq q$ then $\|p\| \leq \|q\|$
- $\mathbf{1} \in \mathcal{M}$ is such that $1 \leq \|\mathbf{1}\|$.

Example 8.2.2. The set $\mathbb{N}$ is a simple instance of quantitative monoid, taking $\|n\| = n$.

In the sequel, we use lower-case consonant letters $p, q, m, v, \ldots$ to denote elements of a quantitative monoid, that we may also call *weights*. Moreover, $\mathbf{n}$ denotes the element of $\mathcal{M}$ defined as $\underbrace{\mathbf{1} + \cdots + \mathbf{1}}_{n \text{ times}}$.

Remark 8.2.3. Here are some intuitions about the previous definition.

- The operation $+$ computes the weight of the interaction of two programs.
- The elements of $\mathcal{M}$ are abstract quantities, so given such an abstract quantity $p \in \mathcal{M}$, $\|p\|$ provides the *concrete* quantity associated to it.
- The inequality $\forall p, q, \|p\| + \|q\| \leq \|p + q\|$ informally represents the fact that the amount of resources consumed by the interaction of two program entities is more than the total amount of resources that they use independently.
- Assuming that the program M has weight p and that $M' \longrightarrow M$, the program M' will be given weight $p + \mathbf{1}$.

We need an operation $! : \mathcal{M} \to \mathcal{M}$ to associate the program $!M$ with the weight $!p$, given that the program M has weight p. In a similar way, we should be able to compute the weight $\S p$ of the program $\S M$. That is why we need *sensible functions*.

Definition 8.2.4 (sensible function)**.** Given a quantitative monoid, we say that a function $f : \mathcal{M} \to \mathcal{M}$ is *sensible* if whenever $p \in \mathcal{M}$ we have $f(p) \leq f(p + \mathbf{1})$ and $\|f(p)\| \neq \|f(p + \mathbf{1})\|$.

Finally, the sensible functions should satisfy a given set of properties that depend on the type system which is interpreted. We enumerate them through the structure of *light monoid*.

Definition 8.2.5 (Light monoid)**.** We call *light monoid* a quantitative monoid $\mathcal{M}$ equipped with three sensible functions $!, \S, F : \mathcal{M} \to \mathcal{M}$ such that for every $p, q \in \mathcal{M}$, the following properties hold:

- (weak dereliction) There is some p' such that $p \leq p'$ and $\S p' \leq !p$
- (monoidalness) $\S(p + q) \leq \S p + \S q$
- (distributivity) There are p', q' such that $p \leq p'$ and $q \leq q'$, that enjoy $\S p' + \S q' \leq \S(p + q)$
- (contraction) $!p + !p \leq !p + \mathbf{1}$

- (functoriality) $!(p+q) \leq F(p) + !q$

Such a light monoid exists, as witnessed by the following example.

Example 8.2.6. We define the structure $(\mathcal{M}_l, +, \mathbf{0}, \mathbf{1}, \leq, \|\cdot\|)$ where

- $\mathcal{M}$ is a set of triples $(n, m, f) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}^{\mathbb{N}}$ where f is a polynomially-bounded function
- $(n, m, f) + (l, k, g) = (n + l, max(m, k), max(f, g))$
- $\mathbf{0} = (0, 0, x \mapsto 0)$
- $\mathbf{1} = (1, 0, x \mapsto x)$
- $(n, m, f) \leq (l, k, g) \iff n \leq l \wedge n + m \leq l + k \wedge f \leq g$
- If $(n, m, f) \in \mathcal{M}$, $\|(n, m, f)\| = n \cdot f(m + n)$.

Then $\mathcal{M}_l$ is a quantitative monoid. Moreover, we can define the three following operations $!, \S, F$ on $\mathcal{M}_l$:

- $\S = (n, m, f) \mapsto (\lceil n/m \rceil, m, x \mapsto x^2 f(x^2))$
- $! = (n, m, f) \mapsto (1, n + m, x \mapsto x^3 f(x^3))$
- $F = (n, m, f) \mapsto (1 + n + m, m, x \mapsto x^3 f(x^3))$

Proposition 8.2.7. *$(\mathcal{M}_l, +, \boldsymbol{0}, \boldsymbol{1}, \leq, \|\cdot\|, !, \S, F)$ is a light monoid.*

Remark 8.2.8. In the monoid $\mathcal{M}_l$, the operations $!, \S$ and F make the degree of the third component of any element of $\mathcal{M}_l$ grow. Therefore we will be able to relate the degree of the bounding polynomial with the depth of the program.

8.2.2 Orthogonality

In a non quantitative setting, orthogonality is usually defined between a program and a stack or any kind of environment [Kri09]. Following Brunel [Bru12], we are going to define an orthogonality relation between a pair (M, p) and a pair (π, f) where M is a term, π is a stack, $p \in \mathcal{M}$ and $f \in \mathcal{M}^{\mathcal{M}}$. The key contribution of the present section is to parametrize the orthogonality relation by a set of pairs (S, s) where S is a store and $s \in \mathcal{M}$ such that s represents the sum of the weight of the values of S.

In orthogonality-based models, we first define a notion of 'correct' computation with respect to some property like *e.g.* termination, progress,... This is done by fixing a set $\perp\!\!\!\perp$ called the *observable*. Here, it contains configurations that execute in bounded time.

Definition 8.2.9 (Observable)**.** The *observable* $\perp\!\!\!\perp$ is a set of pairs (C, p) such that the configuration C terminates in at most $\|p\|$ steps:

$$\perp\!\!\!\perp = \{(C, p) \mid C{\Downarrow^n} \wedge n \leq \|p\|\}$$

Proposition 8.2.10. *This observable satisfies some important properties:*

- *($\leq$-saturation) If $(C,p) \in \perp\!\!\!\perp$ and $p \leq q$ then $(C,q) \in \perp\!\!\!\perp$.*
- *($\longrightarrow$-saturation) If $(C,p) \in \perp\!\!\!\perp$ and $C' \longrightarrow C$ then $(C', p + \mathbf{1}) \in \perp\!\!\!\perp$.*

The property of $\longrightarrow$-saturation will be extensively used to count precisely the number of reduction steps.

We define the following abbreviations:

- $\Lambda_{\mathcal{M}}$ is the set of *weighted terms* $\Lambda \times \mathcal{M}$
- $\mathbb{V}_{\mathcal{M}}$ is the set of *weighted values* $\mathbb{V} \times \mathcal{M}$
- $\Sigma_{\mathcal{M}}$ is the set of *weighted stores* $\Sigma \times \mathcal{M}$
- $\Pi_{\mathcal{M}}$ is the set of *weighted stacks* $\Pi \times \mathcal{M}^{\mathcal{M}}$

The observable induces a notion of orthogonality which, in contrast with usual realizability models, is parametrized by a set of weighted stores.

Definition 8.2.11 (Orthogonality)**.** Suppose $\mathcal{S} \subseteq \Sigma_{\mathcal{M}}$. The *orthogonality relation* $\perp_{\mathcal{S}} \subseteq \Lambda_{\mathcal{M}} \times \Pi_{\mathcal{M}}$ is defined as:

$$(M,p) \perp_{\mathcal{S}} (\pi, e) \quad \iff \quad \forall (S,s) \in \mathcal{S}, (\langle M, \pi, S\rangle, f(p+s)) \in \perp\!\!\!\perp$$

This orthogonality relation lifts to sets of weighted terms and weighted stacks:

- if $X \subseteq \Lambda_{\mathcal{M}}$, $X^{\perp_{\mathcal{S}}} = \{(E,f) \in \Pi_{\mathcal{M}} \mid \forall (M,p) \in X, (M,p) \perp_{\mathcal{S}} (E,f)\}$
- if $X \subseteq \Pi_{\mathcal{M}}$, $X^{\perp_{\mathcal{S}}} = \{(M,p) \in \Lambda_{\mathcal{M}} \mid \forall (E,f) \in X, (M,p) \perp_{\mathcal{S}} (E,f)\}$

The operation $(\cdot)^{\perp_{\mathcal{S}}}$ satisfies the usual orthogonality properties.

Lemma 8.2.12. *Suppose $X, Y \subseteq \Lambda_{\mathcal{M}}$ or $X, Y \subseteq \Pi_{\mathcal{M}}$:*

1. *$X \subseteq Y$ implies $Y^{\perp_{\mathcal{S}}} \subseteq X^{\perp_{\mathcal{S}}}$*
2. *$X \subseteq X^{\perp_{\mathcal{S}}\perp_{\mathcal{S}}}$*
3. *$X^{\perp_{\mathcal{S}}\perp_{\mathcal{S}}\perp_{\mathcal{S}}} = X^{\perp_{\mathcal{S}}}$*

Remark 8.2.13. If $X \subseteq \Lambda_{\mathcal{M}}$, then $X^{\perp_{\mathcal{S}}\perp_{\mathcal{S}}}$ captures all the programs that 'behave' the same as those of X.

The following definition will be useful to over approximate the weight of a term.

Definition 8.2.14 ($\leq$-closure)**.** If $X \subseteq \Lambda_{\mathcal{M}}$ we define its *$\leq$-closure*

$$X^{\leq} = \{(M,p) \mid \exists q \leq p, (M,q) \in X\}$$

Remark 8.2.15. Notice that for any $\mathcal{S}$, we have $X^{\leq} \subseteq X^{\perp_{\mathcal{S}}\perp_{\mathcal{S}}}$. To see this, let $(M,p) \in X^{\leq}$ and $(E,f) \in X^{\perp_{\mathcal{S}}}$. Then there exists $q \leq p$ such that $(M,q) \in X$. Thus for any $(S,s) \in \mathcal{S}$ we have $(M,E,S,f(q+s)) \in \perp\!\!\!\perp$. By $\leq$-saturation we have $(M,E,S,f(p+s)) \in \perp\!\!\!\perp$. We conclude that $(M,p) \in X^{\perp_{\mathcal{S}}\perp_{\mathcal{S}}}$.

We now define a set of $\mathcal{S}$-reducibility candidates that will be used to show the termination in polynomial time of programs.

Definition 8.2.16 ($\mathcal{S}$-reducibility candidates)**.** The set of *$\mathcal{S}$-reducibility candidates*, denoted by $\mathsf{CR}_{\mathcal{S}}$, is the set of $X \subseteq \Lambda_{\mathcal{M}}$ such that:

- $X = X^{\perp_{\mathcal{S}}\perp_{\mathcal{S}}}$
- $(\diamond, x \mapsto x) \in X^{\perp_{\mathcal{S}}}$

Remark 8.2.17. If $(M,p) \in X$, $X \in \mathsf{CR}_{\mathcal{S}}$ and $(\emptyset, \mathbf{0}) \in \mathcal{S}$, then $\langle M, \diamond, \emptyset\rangle$ terminates in at most $\|p\|$ steps. In fact our notion of reducibility candidate extends the usual notion in the non-quantitative case.

8.2.3 Interpretation

We are now going to give an interpretation of the types of $\lambda^{!\S\mathsf{Reg}}_{\mathsf{LAT}}$ by using the orthogonality machinery previously defined. In the end we will be able to prove that each type interpretation is a reducibility candidate.

The interpretation is divided into three parts defined by mutual induction, first on a depth level δ and then on the size of a type A:

- the *interpretation* $\underline{R \vdash^{\delta}} \subseteq \Sigma_{\mathcal{M}}$ of a region context R;
- the *pre-interpretation* $\underline{\underline{R \vdash^{\delta} A}} \subseteq \mathbb{V}_{\mathcal{M}}$ of a type A;
- the *interpretation* $\underline{R \vdash^{\delta} A}_{\mathcal{S}} \subseteq \Lambda_{\mathcal{M}}$ of a type A with respect to a set $\mathcal{S} \subseteq \Sigma_{\mathcal{M}}$.

Here is roughly how it works: we interpret a region context as a set of weighted stores which only contains 'safe' values. These values are said to be safe because they belong to the pre-interpretation of the type of the region in which they are contained. The pre-interpretation of a type is a set of weighted values. In order to get not only values but all the wanted realizers, the interpretation of a type is defined as the bi-orthogonal of its pre-interpretation, where the orthogonality relation is parametrized by a set $\mathcal{S}$.

Suppose R is the region context

$$R = r_1 : (\delta_1, \S A_1), \ldots, r_n : (\delta_n, \S A_n)$$

where types A_i are guarded by a modality $\S$.

Interpretation of region contexts

The interpretation of a region context is defined as follows:

$$\underline{R \vdash^{=\delta}} = \left\{ (S, \sum_{\delta_i = \delta} \sum_{1 \leq j \leq k_i} \S q^i_j) \;\middle|\; \begin{array}{l} dom(S) = \{r_i \mid \delta_i = \delta\} \\ \wedge \quad \forall r_i \in dom(S), S(r_i) = \{\S V^i_1, \S V^i_2, \ldots, \S V^i_{k_i}\} \\ \wedge \quad \forall j \in [1, k_i], (V^i_j, q^i_j) \in \underline{\underline{R \vdash^{\delta_i + 1} A_i}} \end{array} \right\}$$

$$\underline{R \vdash^{\delta}} = \{(S_1 \uplus S_2, s_1 + \S s_2) \mid (S_1, s_1) \in \underline{R \vdash^{=\delta}} \wedge (S_2, s_2) \in \underline{R \vdash^{\delta+1}}\}$$

The intuition is that $\underline{R \vdash^{=\delta}}$ only contains store assignments at level δ. Then, we assign to the region r_i a certain number (written k_i) of values $\S V^i_j$ such that (V^i_j, q^i_j) belongs to the pre-interpretation $\underline{\underline{R \vdash^{\delta_i+1} A_i}}$. The pre-interpretation will be defined such that $(\S V^i_j, \S q^i_j) \in \underline{\underline{R \vdash^{\delta_i} \S A_i}}$. Note that if for all $r_i \in dom(R)$ we have $\delta_i < \delta$, then $\underline{R \vdash^{\delta}} = \emptyset$.

Pre-interpretation of types

We define the pre-interpretation as:

$$\begin{array}{rcl} \underline{\underline{R \vdash^{\delta} \mathsf{Unit}}} & = & \{(\star, \mathbf{1})\}^{\leq} \\ \underline{\underline{R \vdash^{\delta} \mathsf{Reg}_r A}} & = & \{(r, \mathbf{1})\}^{\leq} \\ \underline{\underline{R \vdash^{\delta} A \multimap B}} & = & \{(\lambda x.M, p) \mid \forall (V, v) \in \underline{R \vdash^{\delta} A}, \forall \mathcal{S}, \underline{R \vdash^{\delta}} \sqsubseteq \mathcal{S}, (M[V/x], p + v) \in \underline{R \vdash^{\delta} B}_{\mathcal{S}}\}^{\leq} \\ \underline{\underline{R \vdash^{\delta} \S A}} & = & \{(\S V, \S v) \mid (V, v) \in \underline{\underline{R \vdash^{\delta+1} A}}\}^{\leq} \\ \underline{\underline{R \vdash^{\delta} !A}} & = & \{(!V, !v) \mid (V, v) \in \underline{\underline{R \vdash^{\delta+1} A}}\}^{\leq} \end{array}$$

Note that the pre-interpretation of a type is closed by $\leq$. This will ease the proof of adequacy (Theorem 8.2.23).

Interpretation of types

The interpretation of a type with respect to a set $\mathcal{S}$ is just defined as the bi-orthogonal of the pre-interpretation:

$$\underline{R \vdash^{\delta} A}_{\mathcal{S}} = \underline{\underline{R \vdash^{\delta} A}}^{\perp_{\mathcal{S}} \perp_{\mathcal{S}}}$$

By taking the closure by bi-orthogonal of the pre-interpretation, we capture all the terms (not only values) that realize a given type.

Remark 8.2.18. The presence of regions make the interpretation potentially circular and yet it is well defined. Indeed, the Definition 8.1.5 of well-typing entails the following: to define $\underline{\underline{R \vdash^{\delta} A}}$, we need $\underline{R \vdash^{\delta}}$ to be already defined. But, in R each type is guarded by a modality §. This implies that to define $\underline{R \vdash^{\delta}}$ we only need to know each $\underline{\underline{R \vdash^{\delta+1} A_i}}$.

The important property of the interpretation is that every interpretation of a type A with respect to a set $\mathcal{S} \subseteq \Sigma_{\mathcal{M}}$ is a $\mathcal{S}$-reducibility candidate; this will be used to prove termination in polynomial time.

Proposition 8.2.19. *For any depth δ, we have $\underline{R \vdash^{\delta} A}_{\mathcal{S}} \in \mathsf{CR}_{\mathcal{S}}$. In particular this is true for $\mathcal{S} = \underline{R \vdash^{\delta}}$.*

Proof. By properties of orthogonality (Lemma 8.2.12) we have

$$\underline{R \vdash^{\delta} A}_{\mathcal{S}} = \underline{\underline{R \vdash^{\delta} A}}^{\perp_{\mathcal{S}}\perp_{\mathcal{S}}} = \underline{\underline{R \vdash^{\delta} A}}^{\perp_{\mathcal{S}}\perp_{\mathcal{S}}\perp_{\mathcal{S}}\perp_{\mathcal{S}}}$$

It remains to show

$$(\diamond, x \mapsto x) \in (\underline{R \vdash^{\delta} A}_{\mathcal{S}})^{\perp_{\mathcal{S}}}$$

Take $(M, p) \in \underline{\underline{R \vdash^{\delta} A}}$ and $(S, s) \in \mathcal{S}$. Since $\underline{\underline{R \vdash^{\delta} A}} \subseteq \mathbb{V}_{\mathcal{M}}$, M must be a value and therefore

$$\langle M, \diamond, S, \mathbf{0}\rangle \in \perp\!\!\!\perp$$

which by $\leq$-saturation means

$$\langle M, \diamond, S, p + s\rangle \in \perp\!\!\!\perp$$

and we can conclude

$$(\diamond, x \mapsto x) \in \underline{\underline{R \vdash^{\delta} A}}^{\perp_{\mathcal{S}}} = (\underline{R \vdash^{\delta} A}_{\mathcal{S}})^{\perp_{\mathcal{S}}}$$

□

8.2.4 From adequacy to polynomial time

In this section, we prove a theorem of quantitative adequacy which states that every pair (M, p) of a well-typed program M and a weight $p \in \mathcal{M}$ belongs to the interpretation of its type. The weight p can be inferred automatically by induction on M and by looking at the shape of p we conclude that M terminates in polynomial time.

First we need to fix an ordering relation on stores.

Definition 8.2.20 (Store ordering)**.** Let $\mathcal{S} \subseteq \Sigma_{\mathcal{M}}$ and

$$R = r_1 : (\delta_1, A_1), \ldots, r_n : (\delta_n, A_n)$$

We write $\underline{R \vdash^{\delta}} \sqsubseteq \mathcal{S}$ when the following holds:

- if $(S, s) \in \mathcal{S}$ then there is a decomposition $(S, s) = (S_1 \uplus S_2, s_1 + s_2)$ such that $(S_1, s_1) \in \underline{R \vdash^{\delta}}$ and $dom(S_2) = \{r_i \mid \delta_i < \delta\}$. Moreover, if $(S_3, s_3) \in \underline{R \vdash^{\delta}}$ then $(S_3 \uplus S_2, s_3 + s_2) \in \mathcal{S}$.

Remark 8.2.21. This store ordering will make the type interpretation enjoy properties similar to the one called *extension/restriction* by Amadio [Ama09]. It gives a way to characterize the values of a store that do not impact on the reduction below a certain depth level. More concretely, if $(S, s) \in \mathcal{S}$ such that $\underline{R \vdash^{\delta}} \sqsubseteq \mathcal{S}$, then there is a decomposition $(S, s) = (S_1 \uplus S_2, \S s_1 + s_2)$ such that $(S_1, s_1) \in \underline{R \vdash^{\delta+1}}$ and $dom(S_2) = \{r_i \mid \delta_i < \delta + 1\}$. Therefore $\underline{R \vdash^{\delta+1}} \sqsubseteq \{(S_1 \uplus S_2, s_1)\}$. In other words, this means that the store S_2 does not impact on reductions happening below the depth level $\delta + 1$ since the weight s_2 does not need to be considered.

Notations We shall introduce some notations for readability.

- We use the notations $\overline{V}$, $\overline{p}$ and $\overline{y}$ to denote respectively a list of values $[V_1, \ldots, V_n]$, a list $[p_1, \ldots, p_n]$ of elements of $\mathcal{M}$ and a list of variables $[y_1, \ldots, y_n]$. If M is a term, we denote the term $M[V_1/y_1, \ldots, V_n/y_n]$ by $M[\overline{V}/\overline{y}]$. We denote the list $[\dagger p_1, \ldots, \dagger p_n]$ by $\dagger\overline{p}$ and we define $\sum \overline{p}$ to be the sum $\sum_{1 \leq i \leq n} p_i$.
- Suppose $\Gamma = x_1 : (u_1, A_1), \ldots, x_n : (u_n, A_n)$. Then the notation $(\overline{V}, \overline{p}) \Vdash^{\delta} \Gamma$ stands for $(u_i V_i, p_i) \in \underline{\underline{R \vdash^{\delta} u_i A_i}}$ where $1 \leq i \leq n$. Note that if $u_i = \lambda$ we consider that $\lambda V_i = V_i$ and $\lambda A_i = A_i$.

Example 8.2.22. If we have

$$(\overline{V}, \overline{p}) \Vdash^{\delta} x_1 : (\lambda, A_1), x_2 : (\S, A_2), x_3 : (!, A_3)$$

then $\overline{V} = [V_1, V_2, V_3]$ and $\overline{p} = [p_1, \S p_2, !p_3]$ such that $(V_1, p_1) \in \underline{\underline{R \vdash^{\delta} A_1}}$, $(\S V_2, \S p_2) \in \underline{\underline{R \vdash^{\delta} \S A_2}}$ and $(!V_3, !p_3) \in \underline{\underline{R \vdash^{\delta} !A_3}}$.

We now state the central theorem.

Theorem 8.2.23 (Quantitative adequacy)**.**
Let M be a term of depth d such that $R; \Gamma \vdash^{\delta} M : C$. By taking the light monoid $\mathcal{M}_l$, $(\overline{V}, \overline{p}) \Vdash^{\delta} \Gamma$ and any $\mathcal{S}$ such that $\underline{R \vdash^{\delta}} \sqsubseteq \mathcal{S}$, we have:

- *if M is a value then $(M[\overline{V}/\overline{x}], w(M) + \sum \overline{p}) \in \underline{\underline{R \vdash^{\delta} C}}$,*
- *otherwise $(M[\overline{V}/\overline{x}], w(M) + \sum \overline{p}) \in \underline{R \vdash^{\delta} C_{\mathcal{S}}}$,*

such that the weight $w(M) \in \mathcal{M}_l$ can be automatically computed by following the rules of Figure 8.7.

The proof goes by induction on the typing derivation of M. We first need to prove a particularly interesting lemma that will be used for the case of the rule $\mathsf{prom}_{\S}$.

Lemma 8.2.24 (Promotion)**.** *Suppose that for any $\mathcal{S}$ such that $\underline{R \vdash^{\delta+1}} \sqsubseteq \mathcal{S}$, $(M, m) \in \underline{R \vdash^{\delta+1} A_{\mathcal{S}}}$ holds. Then for any $\mathcal{S}$ such that $\underline{R \vdash^{\delta}} \sqsubseteq \mathcal{S}$, we have $(\S M, \S m + \mathbf{4}) \in \underline{R \vdash^{\delta} \S A_{\mathcal{S}}}$.*

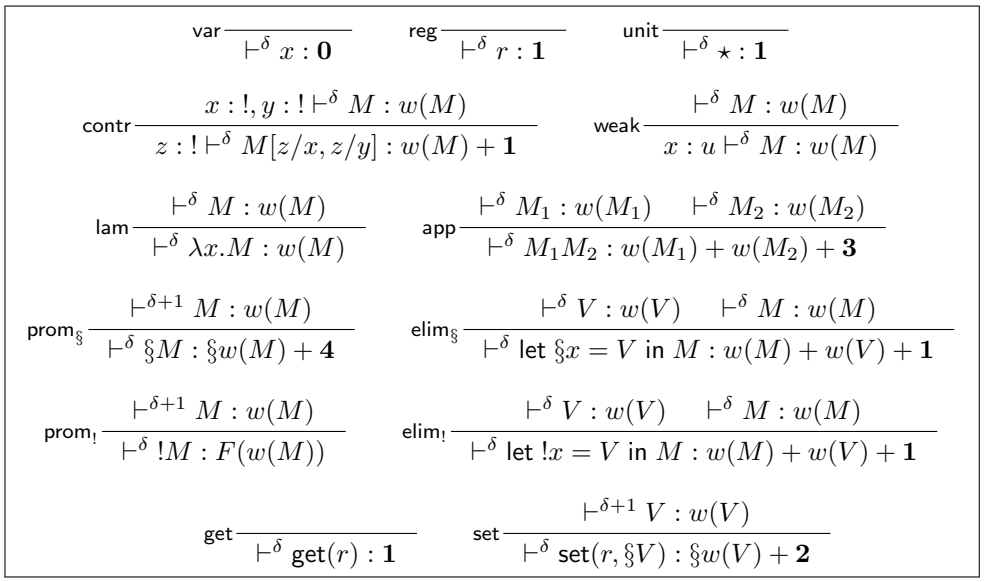


Figure 8.7: Inferred weights from the adequacy of $\lambda^{!\S\mathsf{Reg}}_{\mathsf{LAT}}$

The proof of this lemma makes use of many of the notions we have introduced:

- It requires monoidalness and distributivity of $\S : \mathcal{M} \to \mathcal{M}$ to deal with the store, but this is not true for $! : \mathcal{M} \to \mathcal{M}$ (see Definition 8.2.5). Consequently, it explains why types of regions must be guarded by a paragraph and not a bang.
- It involves the store ordering to identify the parts of the store that do not interact below a certain depth level (see Remark 8.2.21).

Proof of Lemma 8.2.24. Take $\mathcal{S}$ such that $\underline{R \vdash^\delta} \sqsubseteq \mathcal{S}$, $(E, e) \in \underline{\underline{R \vdash^\delta \S A}}^{\perp_S}$ and $(S', s') \in \mathcal{S}$. There is a decomposition $(S', s') = (S_1 \uplus S_2, \S s_1 + s_2)$ such that $(S_1, s_1) \in \underline{R \vdash^{\delta+1}}$ and $dom(S_2) = \{r_i \mid \delta_i < \delta + 1\}$. We want to show

$$(\langle \S M, E, S' \rangle, e(\S m + \mathbf{4} + \S s_1 + s_2)) \in \perp\!\!\!\perp$$

By $\longrightarrow$-saturation, $\leq$-saturation and monoidalness of $\S : \mathcal{M} \to \mathcal{M}$ it suffices to show

$$(\langle M, \S \cdot E, S' \rangle, e(\S(m + s_1) + s_2 + \mathbf{3})) \in \perp\!\!\!\perp$$

We set $\mathcal{S}' = \{(S', s_1)\}$. Clearly $\underline{R \vdash^{\delta+1}} \sqsubseteq \mathcal{S}'$ and $(S', s_1) \in \mathcal{S}'$, therefore it is sufficient to prove

$$(\S \cdot E, \lambda x.e(\S x + \mathbf{3} + s_2)) \in (\underline{R \vdash^{\delta+1} A_{\mathcal{S}'}})^{\perp_{S'}} = \underline{\underline{R \vdash^{\delta+1} A}}^{\perp_{S'}}$$

So let $(V, v) \in \underline{\underline{R \vdash^{\delta+1} A}}$. Hence $(\S V, \S v) \in \underline{\underline{R \vdash^{\delta} \S A}}$ and we know that $(S', \S s_1 + s_2) \in \mathcal{S}$. Since $(E, e) \in \underline{\underline{R \vdash^{\delta} \S A}}^{\perp_{\mathcal{S}}}$, we have

$$(\langle \S V, E, S' \rangle, e(\S v + \S s_1 + s_2)) \in \perp\!\!\!\perp$$

So by $\longrightarrow$-saturation

$$(\langle V, \S \cdot E, S' \rangle, e(\S v + \mathbf{1} + \S s_1 + s_2)) \in \perp\!\!\!\perp$$

Hence, by distributivity of $\S : \mathcal{M} \to \mathcal{M}$, we obtain

$$(\langle V, \S \cdot E, S' \rangle, e(\S(v + s_1) + \mathbf{2} + \mathbf{1} + s_2)) \in \perp\!\!\!\perp$$

□

We can finally go through the proof of quantitative adequacy.

Proof of Theorem 8.2.23. By induction on the typing of M. We highlight four interesting cases. The complete proof can be found in Appendix A.

(set) We have

$$\frac{r : (\delta, \S C) \in R \qquad R; \Gamma \vdash^{\delta+1} V : C}{R; \Gamma \vdash^{\delta} \mathsf{set}(r, \S V) : \mathsf{Unit}}$$

Without loss of generality we assume V is closed and $\Gamma = \emptyset$. Let $\mathcal{S}$ such that $\underline{R \vdash^{\delta}} \sqsubseteq \mathcal{S}$, $(E, e) \in \underline{\underline{R \vdash^{\delta} \mathsf{Unit}}}^{\perp_{\mathcal{S}}}$ and $(S, s) \in \mathcal{S}$. We want to prove

$$(\langle \mathsf{set}(r, \S V), E, S \rangle, e(\S w(V) + \mathbf{2} + s)) \in \perp\!\!\!\perp$$

Since $\langle \mathsf{set}(r, \S V), E, S \rangle \longrightarrow \langle \star, E, S \uplus r \Leftarrow \S V \rangle$, by $\longrightarrow$-saturation if suffices to show

$$(\langle \star, E, S \uplus r \Leftarrow \S V \rangle, e(\mathbf{1} + s + \S w(V)) \in \perp\!\!\!\perp$$

which amounts to prove $(S \uplus r \Leftarrow \S V, s + \S w(V)) \in \mathcal{S}$. There must be a decomposition $(S, s) = (S_1 \uplus S_2, s_1 \uplus s_2)$ such that $(S_1, s_1) \in \underline{R \vdash^{\delta}}$ and $dom(S_2) = \{r_i \mid \delta_i < \delta\}$. By induction we have $(V, w(V)) \in \underline{\underline{R \vdash^{\delta+1} C}}$ and therefore $(S_1 \uplus r \Leftarrow \S V, s_1 + \S w(V)) \in \underline{R \vdash^{\delta}}$. This allows us to conclude $(S \uplus r \Leftarrow \S V, s + \S w(V)) \in \mathcal{S}$.

(get) We have

$$\frac{r : (\delta, \S A) \in R}{R; \vdash^{\delta} \mathsf{get}(r) : \S A}$$

Let $\mathcal{S}$ be such that $\underline{R \vdash^{\delta}} \sqsubseteq \mathcal{S}$, $(E, e) \in \underline{\underline{R \vdash^{\delta} \S A}}^{\perp_{\mathcal{S}}}$ and $(S, s) \in \mathcal{S}$. There must be a decomposition $(S, s) = (S_1 \uplus S_2, s_1 \uplus s_2)$ such that $(S_1, s_1) \in \underline{R \vdash^{\delta}}$ and $dom(S_2) = \{r_i \mid \delta_i < \delta\}$. Therefore the value which is read must belong to S_1. By definition of the interpretation, any decomposition

$(S_1, s_1) = (S' \uplus r \Leftarrow \S V, s' + \S p_V)$ is such that $(V, p_V) \in \underline{\underline{R \vdash^{\delta+1} A}}$. Clearly this entails $(\S V, \S p_V) \in \underline{\underline{R \vdash^{\delta} \S A}}$ and $(S' \uplus S_2, s' + s_2) \in \mathcal{S}$, and therefore

$$(\langle \S V, E, S' \uplus S_2 \rangle, e(\S p_V + s' + s_2)) \in \perp\!\!\!\perp$$

Hence by $\longrightarrow$-saturation $(\langle get(r), E, S \rangle, e(\mathbf{1}+s))$ and we conclude $(get(r), \mathbf{1}) \in \underline{R \vdash^{\delta} \S A}_{\mathcal{S}}$.

($\mathsf{prom}_!$) We have

$$\frac{R; x : (\lambda, A) \vdash^{\delta+1} V : B}{R; x : (!, A) \vdash^{\delta} !V : !B}$$

Let $(V', p) \in \underline{\underline{R \vdash^{\delta+1} A}}$. By induction we have $(V[V'/x], w(V) + p) \in \underline{\underline{R \vdash^{\delta+1} B}}$ and therefore $((!V)[V'/x], !(w(V) + p)) \in \underline{\underline{R \vdash^{\delta} !B}}$. By functoriality of $! : \mathcal{M} \to \mathcal{M}$, $!(w(V) + p) \leq F(w(V)) + !p$ and thus by $\leq$-closure, $((!V)[V'/x], F(w(M)) + !p) \in \underline{\underline{R \vdash^{\delta} !B}}$. And this is what we want since $(!V', !p) \in \underline{\underline{R \vdash^{\delta} !A}}$.

Notice that if the rule $\mathsf{prom}_!$ had more than one variable in the context, we would need the monoidalness property $(!(p+q) \leq !p + !q)$ which is not true. Also, if it were not restricted to values, we would need the distributivity property $(!p + !q \leq !(p+q))$ to prove a lemma similar to Lemma 8.2.24 for the bang modality.

($\mathsf{prom}_\S$) We have

$$\frac{R; \Gamma_\lambda, \Delta_\lambda \vdash^{\delta+1} M : C}{R; \Gamma_\S, \Delta_! \vdash^{\delta} \S M : \S C}$$

Assume the variables associated to the context Γ and Δ are respectively noted $\overline{x}$ and $\overline{y}$. Suppose $(\overline{V}, \overline{p}) \Vdash^{\delta+1} \Gamma_\lambda$ and $(\overline{W}, \overline{q}) \Vdash^{\delta+1} \Delta_\lambda$. By induction hypothesis, for any $\mathcal{S}$ such that $\underline{R \vdash^{\delta+1}} \sqsubseteq \mathcal{S}$ we have

$$(M[\overline{V}/\overline{x}, \overline{W}/\overline{y}], w(M) + \sum \overline{p} + \sum \overline{q}) \in \underline{R \vdash^{\delta+1} C}_{\mathcal{S}}$$

Take $\mathcal{S}'$ such that $\underline{R \vdash^{\delta}} \sqsubseteq \mathcal{S}'$. By Lemma 8.2.24,

$$(\S(M[\overline{V}/\overline{x}, \overline{W}/\overline{y}]), \S(w(M) + \sum \overline{p} + \sum \overline{q}) + \mathbf{4}) \in \underline{R \vdash^{\delta} \S C}_{\mathcal{S}'}$$

By weak dereliction and monoidalness of $\S : \mathcal{M} \to \mathcal{M}$, we have

$$\S(w(M) + \sum \overline{p} + \sum \overline{q}) \leq \S w(M) + \sum \S \overline{p} + \sum !\overline{q}$$

Therefore by $\leq$-saturation we conclude

$$(\S(M[\overline{V}/\overline{x}, \overline{W}/\overline{y}]), \S w(M) + \sum \S \overline{p} + \sum !\overline{q} + \mathbf{4}) \in \underline{R \vdash^{\delta} \S C}_{\mathcal{S}'}$$

which is what we want since $(\overline{V}, \S \overline{p}) \Vdash^{\delta} \Gamma_\S$ and $(\overline{W}, !\overline{q}) \Vdash^{\delta} \Delta_!$. The case where M is a value is similar to the case of the rule $\mathsf{prom}_!$, except that we use the properties weak dereliction and monoidalness of $\S : \mathcal{M} \to \mathcal{M}$ instead of functoriality of $! : \mathcal{M} \to \mathcal{M}$. □

As a corollary of the adequacy theorem, we obtain the announced theorem of termination in polynomial time.

Theorem 8.2.25 (Termination in polynomial time)**.** *There exists a family of polynomials $\{P_d\}_{d\in\mathbb{N}}$ such that for any well-typed term M of depth d, M terminates in time bounded by $P_d(s(M))$.*

Proof. Assume $R; - \vdash^\delta M : A$. By quantitative adequacy (Theorem 8.2.23) we have $(M, w(M)) \in \underline{R \vdash^\delta A}_{\mathcal{S}}$ for any $\mathcal{S}$ such that $\underline{R \vdash^\delta} \sqsubseteq \mathcal{S}$. Since $(\emptyset, \mathbf{0}) \in \mathcal{S}$ and $(\diamond, x \mapsto x) \in \underline{\underline{R \vdash^\delta A}}^{\perp_{\mathcal{S}}}$ by Proposition 8.2.19, we can say that $\langle M, \diamond, \emptyset\rangle$ terminates in at most $\|w(M)\|$ steps. Moreover, the size of the final program must be lower or equal to $\|w(M)\|$.

It remains to analyze the shape of $\|w(M)\|$. It is easy to see that only the rules $\mathsf{prom}_§$ and $\mathsf{prom}_!$ increase the degree of the polynomial bounding the third component of $w(M)$. Therefore $\|w(M)\|$ is a function bounded by a polynomial whose degree depends on the depth of M and which is given the size of M as input. □

8.3 Discussion

Affinity The advantage of the realizability interpretation over the combinatorial method is that it can handle affine terms in an easy way. Recall that the type system $\lambda_{LLT}^{!§\|}$ is strictly linear so that the shallow-first transformation of reduction sequences can be easily proved (see Section 7.2). The realizability proof does not rely on such a transformation and so it is not necessary to have a strict linearity condition. In fact, the weakening rule weak (which allows to derive affine terms) is shown adequate to the interpretation by the fact that the interpretation of a term is a $\leq$-closure (see Definition 8.2.14), which allows to over-approximate the weight of a realizer with the weights of discarded values.

Guarded region types The condition of well-typing (Definition 8.1.5) is quite strict: it requires that every region type is guarded by a modality '§' so that the interpretation is well-defined (see Remark 8.2.18). In fact, it means that every stored value must be of the shape $§V$. This entails a loss of expressivity with respect to the light linear type system $\lambda_{LLT}^{!§\|}$ on the two following points:

- This prevents to implement the *consume-and-rewrite* mechanism (see Section 3.3.4) to simulate the operational semantics of references. Indeed, to be consumed and rewritten, a value must be duplicable, that is of the shape $!V$. The system $\lambda_{LLT}^{!§\|}$ can simulate references (see Section 7.6.2) since region types do not need to be guarded by any modality.
- The system $\lambda_{LLT}^{!§\|}$ captures some kind of side effects that are circular but do not break termination in bounded time, as long they operate on linear

values (*i.e.* of shape different than $\dagger V$). We remarked (see Section 5.5.3[1]) that this is a gain of expressivity with respect to the stratification of regions by effects. On the other hand, the system $\lambda_{\mathsf{LAT}}^{!\S\mathsf{Reg}}$ does not allow such circular side effects because the store cannot contain linear values, they must be of the shape $\S V$.

Guarded recursive types It is well-known that recursive types can be added to a light type system without breaking the time bounds [LB06]. Indeed, we have seen in the second part of this thesis that complexity soundness only relies on a so-called 'depth system'. The realizability framework that we propose can handle *guarded* recursive types. Informally, a guarded recursive type $\mu t.A$ is such that every free occurrence of t in A is in the scope of a modality. For example, the type $\mu t.\mathsf{Unit} \multimap !t$ is guarded while $\mathsf{Unit} \multimap !(\mu t.t \multimap \mathsf{Unit})$ is not. We then define

$$\underline{\underline{R \vdash^{\delta} \mu t.A}} = \underline{\underline{R \vdash^{\delta} A[\mu t.A/t]}}$$

Since the guarding modalities make the depth level of the interpretation increase, it suffices to fix a maximum level δ_{max} so that the interpretation is well-defined. The depth δ_{max} should corresponds to the depth of a program.

We preferred not to include recursive types in the interpretation to keep it relatively simple. Recursive types could be used to type *e.g.* Scott numerals [Wad80], a linear encoding of natural numbers, though the gain of expressivity is quite small because we cannot associate any iterator to them.

Nakano's modality and step-indexing Realizability in the presence of recursive structures (regions, recursive types) is usually difficult. Our realizability model is well-defined because the depth levels permit to reflect the stratification of light logics into the interpretation. We draw some connections with related works. Perhaps the closest work is about the modality '$\triangleright$' of Nakano [Nak00]. In this framework, the index k represents the number of nested '$\triangleright$' modalities. Each type variable of a recursive type or each type of a reference has to be guarded by a modality. Then the interpretation is well-defined by taking that the interpretation of the type $\triangleright A$ at level k is equal to the interpretation of the type A at level $k-1$. This work has been connected to step-indexed models which were proposed by Appel and McAllester [AM01] to interpret recursive types and reference types. Informally, the idea is to index the interpretation of a type by a natural number k and to consider terms that are 'safe' up to k reduction steps. It is then possible to define an interpretation by induction on the index k.

[1] This section deals with the elementary case but it can be safely transposed to the polynomial case.

A state-passing translation Our realizability model is indexed by depth levels. We notice that this 'depth-indexing' technique can be used to give an alternative proof of termination in polynomial time of $\lambda_{\mathsf{LAT}}^{!\S\mathsf{Reg}}$, by means of a typed translation from the imperative λ-calculus to a pure λ-calculus in state-passing style.

Due to circularity issues, it is usually difficult to provide a translation into state-passing style that preserves typing. Only recently, F. Pottier gave a typed state-passing translation [Pot11] from System F equipped with references to an extension of System F_ω that makes use of Nakano's later modality. In this work, termination *and* divergence are preserved by translation.

We should also mention the work of P. Tranquilli [Tra10] which provides a typed translation from an imperative λ-calculus to a pure λ-calculus. In particular, he shows that the stratification of regions by means of types and effects corresponds to avoiding the use of recursive types in the target language. Consequently, he is able to provide an alternative proof of termination of the stratified imperative λ-calculus.

Let us present briefly our idea of depth-indexed translation, by sketching the translation at the type level. Roughly, we want to give a translation from $\lambda_{\mathsf{LAT}}^{!\S\mathsf{Reg}}$ to its functional subset $\lambda_{\mathsf{LAT}}^{!\S}$ (with tensor product), so that the translated programs simulate the original ones. Since we know that the reduction in $\lambda_{\mathsf{LAT}}^{!\S}$ is polynomial, it must also be the case in $\lambda_{\mathsf{LAT}}^{!\S\mathsf{Reg}}$. We first define a state monad

$$T_S(A) = S \multimap A \otimes S$$

The translation is indexed by depth levels. Take the guarded region context

$$R = r_1 : (\delta_1, \S A_1), \dots, r_n : (\delta_n, \S A_n)$$

where $\delta_i \geq \delta$. We translate it as follows:

$$\underline{R \vdash^\delta} = \bigotimes_{\substack{r_i \in dom(R) \\ \wedge\ \delta_i \leq \delta}} \S^{(\delta_i + 1 - \delta)} \underline{R \vdash^{\delta_i + 1} A_i}$$

The type translation is as follows:

$$\begin{array}{rcl}
\underline{R \vdash^\delta \mathsf{Unit}} & = & \mathsf{Unit} \\
\underline{R \vdash^\delta \mathsf{Reg}_r A} & = & \mathsf{Unit} \\
\underline{R \vdash^\delta A \multimap B} & = & \underline{R \vdash^\delta A} \multimap T_{\underline{R \vdash^\delta}}(\underline{R \vdash^\delta B}) \\
\underline{R \vdash^\delta\ !A} & = & !\underline{R \vdash^{\delta+1} A} \\
\underline{R \vdash^\delta \S A} & = & \S\underline{R \vdash^{\delta+1} A}
\end{array}$$

Remark 8.3.1. The translation is well-defined for reasons that are similar to the realizability interpretation (see Remark 8.2.18). To define $\underline{R \vdash^\delta A}$, we need $\underline{R \vdash^\delta}$ to be already defined. But, in R each type is guarded by a modality $\S$. This implies that to define $\underline{R \vdash^\delta}$, we only need to know each $\underline{R \vdash^{\delta_i+1} A_i}$.

The state-passing translation has the same limitations as the realizability interpretation regarding guarded region contexts (see the second paragraph of the section). Moreover, the translation on terms, which is omitted here for the sake of conciseness, illustrates why region types must be guarded by a paragraph (and not a bang), again as in the realizability interpretation.

As future work, we would like to see if we can combine the concurrency monad with the state monad so as to recover the expressivity offered by the combinatorial methods presented in the second part of this thesis.

Chapter 9

Conclusion

In this thesis, we have presented an extension of the framework of Light Logics to higher-order concurrent programs. As a result, we have obtained new static criteria to bound the time complexity of programs. The criteria have been developed gradually: first, we examined the issue of the termination of programs (finite time); then, we considered termination of programs in elementary time; last, we considered termination of programs in polynomial time. In addition, we have introduced type systems that can be combined with the complexity criteria so that well-typed programs are guaranteed to terminate in bounded time *and* to return values. The expressivity of these static criteria has been evaluated by observing that they capture concurrent programs that can iterate functions producing side effects over inductive data structures (see Section 5.5.2 and Section 7.6.2).

These results have required to make various logical concept interact with the world of higher-order concurrent imperative programs:

- We have shown how to combine an affine-intuitionistic type system based on Linear Logic with an effect system that accounts for the way programs act on regions of the store. This allowed us to develop a discipline of region usage to ensure the confluence of programs, and to show that it can be combined with the stratification of regions ensuring the termination of programs.
- We have shown that the central notion of depth as defined in **ELL** and **LLL** can be adapted to control the duplication power of side effects and that it scales to multi-threaded programs. This allowed us to design depth systems ensuring the termination of programs in elementary time and polynomial time.
- We have provided an extension of quantitative realizability to higher-order imperative programs, building on the technique of bi-orthogonality and

introducing a so-called 'depth-indexed' interpretation. *(joint work with Aloïs Brunel)*

Future work

We have shown that Light Logics can deal with call-by-value and several high-level programming features such as imperative references, communication channels and thread generation. We identify two lines of research to get closer to a full-fledged ML language:

- The ability to define algorithms by means of recursive functions and pattern-matching on inductive data types would clearly improves on the programming based on Church-like iterators. Baillot *et al.* already endowed a higher-order functional language with such features [BGM10], but this seems to complicate very much the proof of complexity soundness, and it is not clear that the method could work for call-by-value.
- The programming languages that we propose require the programmer to write code with explicit bang constructors and destructors. In order to ease the programming, such modal information could be hidden at the level of types, as in *e.g.* **DLAL** [BT09b]. Then, the main work consists in providing an automatic type inference procedure. In this spirit, Atassi *et al.* [ABT07] propose a way to determine if a term typable in System F is typable in **DLAL**, and to ouput a typing judgment if it is so.

Discussion

Overall, the contribution in terms of programming flexibility seems quite modest. Even though we can extend languages with new high-level features, the programming experiments of this thesis show that Light Logics are very constraining to the programmer. This does not seem so much related to the complexity bounds that must be guaranteed but rather to the stratified nature of programs. Indeed, whether we program with $\lambda^{!\|}_{EAT}$ that corresponds to the very large complexity class of elementary time or with $\lambda^{!\S\|}_{LLT}$ that corresponds to the more feasible class of polynomial time, the constraints are mainly due to depth level criteria.

These mixed results about the programming expressivity of Light Logics question the applicability of the ICC approach to the static determination of resource usages of programs: is it reasonable to design a programming language from ICC criteria? On the one hand, the checking of ICC criteria can generally be computed efficiently; but on the other hand, the set of valid programs is most of the time not acceptable. The alternative seems to be to have a full-fledged programming language and then to rely on an automatic or mechanized (proof assisted) static analysis of the resource usages. The advantage of this approach is the

absence of programming constraints, but as pointed out in the introduction of this thesis, the big work then consists in providing an efficient and precise analysis. Recently though, Jost *et al.* proposed [JHLH10] an automatic *amortized analysis* of the resource usage of higher-order functional which can be efficiently computed. The obtained bounds have been compared to actual measurements and seem quite precise. Yet, it remains to explore the scalability of the method to imperative and concurrent features.

In the goal of proof assisted analysis, Gaboardi and Dal Lago recently proposed [LG11a] a system of *linear dependent types* which brings estimations on the time complexity of higher-order functional programs. In fact, this type system is quite ambitious in that it is *relatively complete*: every program which terminates in k steps can be given a typing judgment from which the number k can be recovered. Of course, the type checking problem of this system has to be undecidable, but interestingly, Dal Lago and Petit [LP12] show that this problem can be reduced to the much tractable one of checking the validity of first-order inequalities. The method seems promising in bringing interactive tools to studying the complexity of programs. However, the complexity of the type system itself appears to be a big obstacle to the integration of imperative and concurrent features. One solution is perhaps to make use of the quantitative realizability framework presented in Chapter 8 and which seems quite adaptable to various programming features. In fact, Brunel and Gaboardi already built [BG12] a quantitative realizability model of a system of linear dependent types, which may be a good source of inspiration.

Every work cited above focuses on complexity bounds which are related to high-level operational semantics. As far as computer security is concerned, it is crucial to compute as precise bounds as possible. Therefore, one question that should not be neglected is the following: does compilation to low-level code preserve the complexity of programs? Amadio and Regis-Gianas recently proposed [ARG11] a labelling method to certify cost annotations of higher-order functional programs with respect to target assembly code. Yet, the method would need to be extended to higher-order concurrent programs. An alternative solution could be to study a type-preserving translation from a high-level language to an assembly language, so that we can derive a complexity-preserving compilation chain. The work on *typed assembly language* [MWCG99] may be a good starting point.

The study of a type-preserving translation in the above spirit could perhaps reveal interesting connections between Light Logics and low-level languages. This raises the more general question of whether Light Logics have a broader application than functional programming.

Bibliography

[ABM10] Roberto M. Amadio, Patrick Baillot, and Antoine Madet. An affine-intuitionistic system of types and effects: confluence and termination. *CoRR*, abs/1005.0835, 2010. 23

[ABT07] Vincent Atassi, Patrick Baillot, and Kazushige Terui. Verification of ptime reducibility for system f terms: Type inference in dual light affine logic. *Logical Methods in Computer Science*, 3(4), 2007. 176

[AD08] Roberto M. Amadio and Mehdi Dogguy. On affine usages in signal-based communication. In G. Ramalingam, editor, *APLAS*, volume 5356 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2008. 68

[AM01] Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, 2001. 25, 172

[Ama09] Roberto M. Amadio. On stratified regions. In Zhenjiang Hu, editor, *APLAS*, volume 5904 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2009. 22, 29, 33, 41, 167

[ARG11] Roberto Amadio and Yann Régis-Gianas. Certifying and reasoning on cost annotations of functional programs. In Ricardo Peña, Marko van Eekelen, and Olha Shkaravska, editors, *FOPARA*, volume 7177 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2011. 177

[Asp98] Andrea Asperti. Light affine logic. In *LICS*, pages 300–308. IEEE Computer Society, 1998. 20, 144, 148

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical report, University of Edinburgh, 1996. 47

[BBdPH93] P. N. Benton, Gavin M. Bierman, Valeria de Paiva, and Martin Hyland. A term calculus for intuitionistic linear logic. In Marc

Bezem and Jan Friso Groote, editors, *TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 1993. 47

[BC92] Stephen Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992. 17

[Ben94] P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In Leszek Pacholski and Jerzy Tiuryn, editors, *CSL*, volume 933 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1994. 47

[BG12] Aloïs Brunel and Marco Gaboardi. Quantitative reducibility candidates for dlpcf. Third International Workshop on Developments in Implicit Complexity, Tallinn, Estonia, 2012. 177

[BGM10] Patrick Baillot, Marco Gaboardi, and Virgile Mogbil. A polytime functional language from light linear logic. In *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 104–124. Springer, 2010. 21, 176

[BM04] Patrick Baillot and Virgile Mogbil. Soft lambda-calculus: A language for polynomial time computation. In Igor Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2004. 20

[BM12] Aloïs Brunel and Antoine Madet. Indexed realizability for bounded-time programming with references and type fixpoints. In Ranjit Jhala and Atsushi Igarashi, editors, *APLAS*, volume 7705 of *Lecture Notes in Computer Science*, pages 264–279. Springer, 2012. 25

[Bou10] Gérard Boudol. Typing termination in a higher-order concurrent imperative language. *Information and Computation*, 208(6):716–736, 2010. 23, 29, 40, 41

[Bru12] Aloïs Brunel. Quantitative classical realizability. *submitted*, 2012. 25, 153, 159, 160, 162

[BT09a] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda calculus. *Information and Computation*, 207(1):41 – 62, 2009. 21

[BT09b] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda calculus. *Information and Computation*, 207(1):41–62, 2009. 176

[CDLRDR08] Paolo Coppola, Ugo Dal Lago, and Simona Ronchi Della Rocca. Light logics and the call-by-value lambda calculus. *Logical Methods in Computer Science*, 4(4), 2008. 21

[Chu33] Alonzo Church. A set of postulates for the foundation of logic (2nd paper). *Annals of Mathematics*, 34(4):839–864, October 1933. 15

[DBL11] *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. IEEE Computer Society, 2011. 182, 183

[DJ03] Vincent Danos and Jean-Baptiste Joinet. Linear logic and elementary time. *Information and Computation*, 183(1):123 – 137, 2003. 20, 77, 101, 187

[DJS95] Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. On the linear decoration of intuitionistic derivations. *Archive for Mathematical Logic*, 33:387–412, 1995. 10.1007/BF02390456. 52

[FMA06] Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. Linear regions are all you need. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 7–21. Springer, 2006. 48

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987. 18, 47, 51

[Gir98] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998. 18, 20, 109, 144

[GMR12] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. An implicit characterization of pspace. *ACM Transactions on Computational Logic*, 13(2):18, 2012. 20

[GR09] Marco Gaboardi and Simona Ronchi Della Rocca. From light logics to type assignments: a case study. *Logic Journal of the IGPL*, 17(5):499–530, 2009. 21, 149

[IK05] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, March 2005. 68

[JHLH10] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL*, pages 223–236. ACM, 2010. 177

[Kle73] S. Kleene. Realizability: A retrospective survey. In A. Mathias and H. Rogers, editors, *Cambridge Summer School in Mathematical Logic*, volume 337 of *Lecture Notes in Mathematics*, pages 95–112. Springer, 1973. 159

[Kob02] Naoki Kobayashi. Type systems for concurrent programs. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th*

Anniversary Colloquium of UNU/IIST, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453. Springer, 2002. 68

[KPT99] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999. 68

[Kri09] Jean-Louis Krivine. Realizability in classical logic. *Panoramas et synthèses*, 27:197–229, 2009. 25, 153, 159, 162

[Laf04] Yves Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1-2):163–180, 2004. 20, 149

[LB06] Ugo Dal Lago and Patrick Baillot. On light logics, uniform encodings and polynomial time. *Mathematical Structures in Computer Science*, 16(4):713–733, 2006. 172

[Lei91] Daniel Leivant. A foundational delineation of computational feasiblity. In *LICS*, pages 2–11. IEEE Computer Society, 1991. 17

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In Jeanne Ferrante and P. Mager, editors, *POPL*, pages 47–57. ACM Press, 1988. 29

[LG11a] Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. In *LICS* [DBL11], pages 133–142. 177

[LG11b] Ugo Dal Lago and Paolo Di Giamberardino. Soft session types. In Bas Luttik and Frank Valencia, editors, *EXPRESS*, volume 64 of *EPTCS*, pages 59–73, 2011. 21

[LH10] Ugo Dal Lago and Martin Hofmann. A semantic proof of polytime soundness of light affine logic. *Theory of Computing Systems*, 46:673–689, 2010. 159

[LH11] Ugo Dal Lago and Martin Hofmann. Realizability models and implicit complexity. *Theoretical Computer Science*, 412(20):2029 – 2047, 2011. Girard's Festschrift. 24, 153, 159, 160

[LHMV12] Ugo Dal Lago, Tobias Heindel, Damiano Mazza, and Daniele Varacca. Computational complexity of interactive behaviors. *CoRR*, abs/1209.0663, 2012. 21

[LMS10] Ugo Dal Lago, Simone Martini, and Davide Sangiorgi. Light logics and higher-order processes. In *EXPRESS'10*, volume 41 of *EPTCS*, pages 46–60, 2010. 21

[LP12] Ugo Dal Lago and Barbara Petit. The geometry of types. `http://lideal.cs.unibo.it/GeoTypes.pdf`, 2012. 177

[MA11] Antoine Madet and Roberto M. Amadio. An elementary affine λ-calculus with multithreading and side effects. In C.-H. Luke

Ong, editor, *TLCA*, volume 6690 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2011. 24

[Mad12] Antoine Madet. A polynomial time -calculus with multithreading and side effects. In Danny De Schreye, Gerda Janssens, and Andy King, editors, *PPDP*, pages 55–66. ACM, 2012. 24

[Mar11] Jean-Yves Marion. A type system for complexity flow analysis. In *LICS* [DBL11], pages 123–132. 21

[MOTW99] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theoretical Computer Science*, 228(1-2):175–210, 1999. 47

[MP12] Jean-Yves Marion and Romain Péchoux. Complexity information flow in a multi-threaded imperative language. *CoRR*, abs/1203.6878, 2012. 21

[MWCG99] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999. 177

[Nak00] Hiroshi Nakano. A modality for recursion. In *LICS*, pages 255–266. IEEE Computer Society, 2000. 25, 172

[Plo93] Gordon D. Plotkin. Type theory and recursion (extended abstract). In *LICS*, page 374. IEEE Computer Society, 1993. 47

[Pot11] François Pottier. A typed store-passing translation for general references. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 147–158. ACM, 2011. 173

[TBE+06] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld, and Olesen Peter Sestoft. Programming with regions in the mlkit revised for version 4.3.0, 2006. 45

[TBEH04] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004. 45

[Ter07] Kazushige Terui. Light affine lambda calculus and polynomial time strong normalization. *Archive for Mathematical Logic*, 46(3-4):253–280, 2007. 20, 23, 24, 51, 77, 78, 109, 110, 111, 113, 124, 148

[Tra10] Paolo Tranquilli. Translating types and effects with state monads and linear logic. `http://www.cs.unibo.it/~tranquil/content/docs/typeseffects.pdf`, 2010. 173

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information Computation*, 132(2):109–176, 1997. 29, 45

[Wad80] C. Wadsworth. Some unusual λ-calculus numeral systems. In J.P. Seldin and J.R. Hindley, editors, *To HB Curry: Essays on combinatory logic, lambda calculus and formalism*, pages 215–230. Academic Press, 1980. 172

[Wad93] Philip Wadler. A taste of linear logic. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *MFCS*, volume 711 of *Lecture Notes in Computer Science*, pages 185–210. Springer, 1993. 47

[WW01] David Walker and Kevin Watkins. On regions and linear types. In Benjamin C. Pierce, editor, *ICFP*, pages 181–192. ACM, 2001. 48

Appendix A

Proofs

Chapter 4

Proof of Lemma 4.4.5.

1. By induction on y. The case for $y = 0$ is clear. For the inductive case, we notice:
$$(y+1)+1 \leq 2^y + 2^y = 2^{y+1} \leq x^{y+1} \ .$$

2. By induction on y. The case $y = 0$ is clear. For the inductive case, we notice:
$$\begin{array}{rcll} x \cdot (y+1) & \leq & x \cdot (x^y) & \text{(by 1)} \\ & = & x^{(y+1)} & \end{array}$$

3. By induction on z. The case $z = 0$ is clear. For the inductive case, we notice:
$$\begin{array}{rcll} (x \cdot y)^{z+1} & = & (x \cdot y)^z (x \cdot y) & \\ & \leq & x^{y \cdot z}(x \cdot y) & \text{(by inductive hypothesis)} \\ & \leq & x^{y \cdot z}(x^y) & \text{(by 2)} \\ & = & x^{y \cdot (z+1)} & \end{array}$$

4. From $z \geq 1$ we derive $y \leq y^z$. Then:
$$\begin{array}{rcll} x^z \cdot y & \leq & x^z \cdot y^z & \\ & = & (x \cdot y)^z & \\ & \leq & x^{y \cdot z} & \text{(by 3)} \end{array}$$

5. By the binomial law, we have $x^k = ((x-y)+y)^k = (x-y)^k + y^k + p$ with $p \geq 0$. Thus $(x-y)^k = x^k - y^k - p$ which implies $(x-y)^k \leq x^k - y^k$.

□

Proof of Theorem 4.4.7. Suppose $\mu_\alpha(M) = (x_0, \ldots, x_\alpha)$ so that x_i corresponds to the occurrences at depth $(\alpha - i)$ for $0 \leq i \leq \alpha$. Also assume the reduction is at depth $(\alpha - i)$. By looking at equations (4.7) and (4.8) in the proof of Proposition 4.4.2 we see that the components $i+1, \ldots, \alpha$ of $\mu_\alpha(M)$ and $\mu_\alpha(M')$ coincide. Hence, let $k = 2^{t_\alpha(x_{i+1}, \ldots, x_\alpha)}$. By definition of the tower function, $k \geq 1$.

We proceed by case analysis on the reduction rules.

- $M \equiv \mathsf{let}\ !x = !M_2\ \mathsf{in}\ M_1 \longrightarrow P' \equiv M_1[M_2/x]$
  By Inequality (4.8) we know

$$\begin{array}{rcl} t_\alpha(\mu_\alpha(M')) & \leq & t_\alpha(x_0 \cdot x_{i-1}, \ldots, x_{i-1} \cdot x_{i-1}, x_i - 2, x_{i+1}, \ldots, x_\alpha) \\ & = & t_\alpha(x_0 \cdot x_{i-1}, \ldots, x_{i-1} \cdot x_{i-1}, x_i - 2)^k \end{array}$$

  By iterating Lemma 4.4.4 we derive:

$$\begin{array}{rl} & t_\alpha(x_0 \cdot x_{i-1}, x_1 \cdot x_{i-1}, \ldots, x_{i-1} \cdot x_{i-1}, x_i - 2) \\ \leq & t_\alpha(x_0, x_1 \cdot x_{i-1}^2, \ldots, x_{i-1} \cdot x_{i-1}, x_i - 2) \\ \leq & \ldots \\ \leq & t_\alpha(x_0, x_1, \ldots, x_{i-1}^i, x_i - 2) \end{array}$$

  Renaming x_{i-1} with x and x_i with y, we are left to show

$$(\alpha x^i)^{2^{(\alpha \cdot (y-2))^k}} < (\alpha x)^{2^{(\alpha \cdot y)^k}}$$

  Since $i \leq \alpha$ the first quantity is bounded by

$$(\alpha x)^{\alpha \cdot 2^{(\alpha \cdot (y-2))^k}}$$

  We notice

$$\begin{array}{rll} & \alpha \cdot 2^{(\alpha \cdot (y-2))^k} & \\ = & \alpha \cdot 2^{(\alpha \cdot y - \alpha \cdot 2)^k} & \\ \leq & \alpha \cdot 2^{(\alpha \cdot y)^k - (\alpha \cdot 2)^k} & \text{(by Lemma 4.4.5-5)} \end{array}$$

  So we are left to show

$$\alpha 2^{(\alpha \cdot y)^k - (\alpha \cdot 2)^k)} \leq 2^{(\alpha \cdot y)^k}$$

  Dividing by $2^{(\alpha \cdot y)^k}$ and recalling that $k \geq 1$, it remains to check

$$\alpha \cdot 2^{-(\alpha \cdot 2)^k} \leq \alpha \cdot 2^{-(\alpha \cdot 2)} < 1$$

  which is obviously true for $\alpha \geq 1$.

- $M \equiv (\lambda x.M_1)M_2 \longrightarrow M' \equiv M_1[M_2/x]$
  By Equation (4.7), we have

$$t_\alpha(\mu_\alpha(M')) \leq t_\alpha(x_0, \ldots, x_{i-1}, x_i - 2, x_{i+1}, \ldots, x_\alpha)$$

  and one can check that this quantity is strictly less than

$$t_\alpha(\mu_\alpha(M)) = t_\alpha(x_0, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_\alpha)$$

□

Chapter 5

Proof of Theorem 5.5.3

Elementary functions are characterized as the smallest class of functions containing zero, successor, projection, subtraction and which is closed by composition and bounded summation/product. We will need the arithmetic functions defined in Figure A.1. We will abbreviate $\lambda^! xM$ for $\lambda x.\mathsf{let}\ !x = x\ \mathsf{in}\ M$.

In the following, we show that the required functions can be represented in the sense of Definition 5.5.2 by adapting the proofs from Danos and Joinet [DJ03].

Successor, addition and multiplication

We check that succ represents the *successor* function s:

$$s : \mathbb{N} \mapsto \mathbb{N}$$
$$s(x) = x + 1$$

Proposition A.0.2. $\mathsf{succ} \Vdash s$.

Proof. Take $\emptyset \vdash^\delta M : \mathsf{Nat}$ and $M \Vdash n$. We have $\emptyset \vdash^\delta \mathsf{succ} : \mathsf{Nat} \multimap \mathsf{Nat}$. We can show that $\mathsf{succ}\ M \longrightarrow^* \overline{s(n)}$, hence $\mathsf{succ}\ M \Vdash s(n)$. Thus $\mathsf{succ} \Vdash s$. □

We check that add represents the *addition* function a:

$$a : \mathbb{N}^2 \mapsto \mathbb{N}$$
$$a(x, y) = x + y$$

Proposition A.0.3. $\mathsf{add} \Vdash a$.

Proof. For $i = 1, 2$ take $\emptyset \vdash^\delta M_i : \mathsf{Nat}$ and $M_i \Vdash n_i$. We have $\emptyset \vdash^\delta \mathsf{add} : \mathsf{Nat} \multimap \mathsf{Nat} \multimap \mathsf{Nat}$. We can show that $\mathsf{add}\ M_1 M_2 \longrightarrow^* \overline{a(n_1, n_2)}$, hence $\mathsf{add}\ M_1 M_2 \Vdash a(n_1, n_2)$. Thus $A \Vdash a$. □

$$
\begin{array}{rcll}
\mathsf{Nat} & = & \forall t.!(t \multimap t) \multimap !(t \multimap t) & \text{(type of numerals)} \\
 & & & \\
\mathsf{zero} & : & \mathsf{Nat} & \text{(zero)} \\
\mathsf{zero} & = & \lambda f.!(\lambda x.x) & \\
 & & & \\
\mathsf{succ} & : & \mathsf{Nat} \multimap \mathsf{Nat} & \text{(successor)} \\
\mathsf{succ} & = & \lambda n.\lambda f.\mathsf{let}\ !f = f\ \mathsf{in} & \\
 & & \mathsf{let}\ !y = n!f\ \mathsf{in} !(\lambda x.f(yx)) & \\
 & & & \\
\overline{n} & : & \mathsf{Nat} & \text{(numerals)} \\
\overline{n} & = & \lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ !(\lambda x.f(\cdots(fx)\cdots)) & \\
 & & & \\
\mathsf{add} & : & \mathsf{Nat} \multimap (\mathsf{Nat} \multimap \mathsf{Nat}) & \text{(addition)} \\
\mathsf{add} & = & \lambda n.\lambda m.\lambda f.\mathsf{let}\ !f = f\ \mathsf{in} & \\
 & & \mathsf{let}\ !y = n!f\ \mathsf{in} & \\
 & & \mathsf{let}\ !y' = m!f\ \mathsf{in} \quad !(\lambda x.y(y'x)) & \\
 & & & \\
\mathsf{mult} & : & \mathsf{Nat} \multimap (\mathsf{Nat} \multimap \mathsf{Nat}) & \text{(multiplication)} \\
\mathsf{mult} & = & \lambda n.\lambda m.\lambda f.\mathsf{let}\ !f = f\ \mathsf{in} & \\
 & & n(m!f) & \\
 & & & \\
\mathsf{int\_it} & : & \mathsf{Nat} \multimap \forall t.!(t \multimap t) \multimap !t \multimap !t & \text{(iteration)} \\
\mathsf{int\_it} & = & \lambda n.\lambda g.\lambda x.\mathsf{let}\ !y = ng\ \mathsf{in} & \\
 & & \mathsf{let}\ !y' = x\ \mathsf{in}\ !(yy') & \\
 & & & \\
\mathsf{gen\_it} & : & \forall t.\forall t'.!(t \multimap t) \multimap (!(t \multimap t) \multimap t') \multimap \mathsf{Nat} \multimap t' & \\
\mathsf{gen\_it} & = & \lambda s.\lambda e.\lambda n.e(nts) &
\end{array}
$$

Figure A.1: Representation of arithmetic functions

We check that mult represents the multiplication function m:

$$
\begin{array}{l}
m : \mathbb{N}^2 \mapsto \mathbb{N} \\
m(x, y) = x * y
\end{array}
$$

Proposition A.0.4. $\mathsf{mult} \Vdash m$.

Proof. For $i = 1, 2$ take $\emptyset \vdash^{\delta} M_i : \mathsf{Nat}$ and $M_i \Vdash n_i$. We have $\emptyset \vdash^{\delta} \mathsf{mult} : \mathsf{Nat} \multimap \mathsf{Nat} \multimap \mathsf{Nat}$. We can show that $\mathsf{mult}\, M_1 M_2 \longrightarrow^* \overline{m(n_1, n_2)}$, hence $\mathsf{mult}\, M_1 M_2 \Vdash m(n_1, n_2)$. Thus $\mathsf{mult} \Vdash m$. □

Iteration schemes

We check that $\mathsf{int\_it}$ represents the following iteration function it:

$$\begin{array}{l} it : (\mathbb{N} \mapsto \mathbb{N}) \mapsto \mathbb{N} \mapsto \mathbb{N} \mapsto \mathbb{N} \\ it(f, n, x) = f^n(x) \end{array}$$

Proposition A.0.5. $\mathsf{int\_it} \Vdash it$.

Proof. We have $\emptyset \vdash^\delta \mathsf{int\_it} : \mathsf{Nat} \multimap \forall t.!(t \multimap t) \multimap !t \multimap !t$. Given $\emptyset \vdash^\delta M : \mathsf{Nat}$ with $M \Vdash n$, $\emptyset \vdash^\delta F : \mathsf{Nat} \multimap \mathsf{Nat}$ with $F \Vdash f$ and $\emptyset \vdash^\delta X : \mathsf{Nat}$ with $X \Vdash x$, we observe that $\mathsf{int\_it}\, M(!F)(!X) \longrightarrow^* F^n X$. Since $F \Vdash f$ and $X \Vdash x$, we get $F^n X \longrightarrow^* \overline{it(f, n, x)}$. Hence $\mathsf{int\_it} \Vdash it$. □

The function it is an instance of the more general iteration scheme git:

$$\begin{array}{l} git : (\mathbb{N} \mapsto \mathbb{N}) \mapsto ((\mathbb{N} \mapsto \mathbb{N}) \mapsto \mathbb{N}) \mapsto \mathbb{N} \mapsto \mathbb{N} \\ git(step, exit, n) = exit(\lambda x.step^n(x)) \end{array}$$

Indeed, we have:

$$git(f, \lambda f.fx, n) = (\lambda f.fx)(\lambda x.f^n(x)) = it(f, n, x)$$

Proposition A.0.6. $\mathsf{gen\_it} \Vdash git$.

Proof. Take $\emptyset \vdash^\delta M : \mathsf{Nat}$ with $M \Vdash n$, $\emptyset \vdash^\delta E : ((\mathsf{Nat} \multimap \mathsf{Nat}) \multimap \mathsf{Nat}) \multimap \mathsf{Nat}$ with $E \Vdash exit$, $\emptyset \vdash^\delta S : \mathsf{Nat} \multimap \mathsf{Nat}$ with $S \Vdash step$. Then we have $\underline{\mathsf{gen\_it} SEM \longrightarrow^* E(\lambda x.S^n x)}$. Since $S \Vdash step$ and $E \Vdash exit$ we have $E(\lambda x.S^n x) \longrightarrow^* \overline{exit(\lambda x.step^n(x)}$. Hence $\mathsf{gen\_it} \Vdash git$. □

Coercion

Let $S = \lambda n^N.S'$. For $0 \geq i$, we define S'_i inductively:

$$\begin{array}{l} S'_0 = S' \\ S'_i = \mathsf{let}\ !n = n\ \mathsf{in}\ !S'_{i-1} \end{array}$$

Let $S_i = \lambda n.S'_i$. We can derive $\emptyset \vdash^\delta S_i : !^i\mathsf{Nat} \multimap !^i\mathsf{Nat}$. For $i \geq 0$, we define C_i inductively:

$$\begin{array}{l} C_0 = \lambda x.x \\ C_{i+1} = \lambda n.\mathsf{int\_it}(!S_i)(!^{i+1}\overline{0})n \\ \emptyset \vdash^\delta C_i : \mathsf{Nat} \multimap !^i\mathsf{Nat} \end{array}$$

Lemma A.0.7 (integer representation is preserved by coercion)**.** *Let* $\emptyset \vdash^\delta M : \mathsf{Nat}$ *and* $M \Vdash n$. *We can derive* $\emptyset \vdash^\delta C_i M : !^i\mathsf{Nat}$. *Moreover* $C_i M \Vdash n$.

Proof. By induction on i. □

Lemma A.0.8 (function representation is preserved by coercion)**.** *Let*

$$\emptyset \vdash^{\delta} F : !^{i_1}\mathsf{Nat}_1 \multimap \ldots \multimap !^{i_k}\mathsf{Nat}_k \multimap !^{p}\mathsf{Nat}$$

and $\emptyset \vdash^{\delta} M_j : \mathsf{Nat}$ *with* $M_j \Vdash n_j$ *for* $1 \leq j \leq k$ *such that* $F(!^{i_1}M_1 \ldots (!^{i_k}M_k)) \longrightarrow^* \overline{f(n_1, \ldots, n_k)}$*. Then we can find a term* $\mathcal{C}(F) = \lambda \vec{x}^{\mathsf{Nat}}.F((C_{i_1}x_1)\ldots(C_{i_k}x_k))$ *such that*

$$\emptyset \vdash^{\delta} \mathcal{C}(F) : \mathsf{Nat} \multimap \mathsf{Nat} \multimap \ldots \multimap \mathsf{Nat} \multimap !^{p}\mathsf{Nat}$$

and $\mathcal{C}(F) \Vdash f$*.*

Predecessor and subtraction

We first want to represent *predecessor*:

$$\begin{array}{l} p : \mathbb{N} \mapsto \mathbb{N} \\ p(0) = 0 \\ p(x) = x - 1 \end{array}$$

We define the following terms:

$$\begin{array}{l} ST = !(\lambda z.\langle \mathsf{snd}\ z, f(\mathsf{snd}\ z)\rangle) \\ f : (\delta + 1, t \multimap t) \vdash^{\delta} ST : !(t \times t \multimap t \times t) \end{array}$$

$$\begin{array}{l} EX = \lambda g.\mathsf{let}\ !g = g\ \mathsf{in}\ !(\lambda x.\mathsf{fst}\ g\langle x, x\rangle) \\ \emptyset \vdash^{\delta} EX : !(t \times t \multimap t \times t) \multimap !(t \multimap t) \end{array}$$

$$\begin{array}{l} P = \lambda n.\lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ \mathsf{gen\_it}\ ST\ EX\ n \\ \emptyset \vdash^{\delta} P : \mathsf{Nat} \multimap \mathsf{Nat} \end{array}$$

Proposition A.0.9 (predecessor is representable)**.** $P \Vdash p$*.*

Proof. Take $\emptyset \vdash^{\delta} M : \mathsf{Nat}$ and $M \Vdash n$. We can show that $(PM)^{-} \longrightarrow^* \overline{p(n)}$, hence $PM \Vdash p(n)$. Thus $P \Vdash p$. □

Now we want to represent (positive) subtraction s:

$$\begin{array}{l} s : \mathbb{N}^2 \mapsto \mathbb{N} \\ s(x, y) = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{if } y \geq x \end{cases} \end{array}$$

Take

$$\begin{array}{l} SUB = \lambda m.\mathsf{let}\ !m = m\ \mathsf{in}\ \lambda n.\mathsf{int\_it}\ !P\ !m\ n : !\mathsf{Nat} \multimap \mathsf{Nat} \multimap !\mathsf{Nat} \\ \emptyset \vdash^{\delta} SUB : !\mathsf{Nat} \multimap \mathsf{Nat} \multimap !\mathsf{Nat} \end{array}$$

Proposition A.0.10 (subtraction is representable)**.** $\mathcal{C}(SUB) \Vdash s$*.*

Proof. For $i = 1, 2$ take $\emptyset \vdash^{\delta} M_i : \mathsf{Nat}$ and $M_i \Vdash n_i$. We can show that $(SUB(!M_1)M_2)^{-} \longrightarrow^* \overline{s(n_1, n_2)}$. Hence by Lemma A.0.8, $\mathcal{C}(SUB) \Vdash s$. □

Composition

Let g be a m-ary function and G be a term such that $\emptyset \vdash^{\delta} G : \mathsf{Nat}_1 \multimap \ldots \multimap \mathsf{Nat}_m \multimap !^{p}\mathsf{Nat}$ (where $p \geq 0$) and $G \Vdash g$. For $1 \leq i \leq m$, let f_i be a k-ary function and F_i a term such that $\emptyset \vdash^{\delta} F_i : \mathsf{Nat}_1 \multimap \ldots \multimap \mathsf{Nat}_k !^{q_i}\mathsf{Nat}$ (where $q_i \geq 0$) and $F_i \Vdash f_i$. We want to represent the composition function h such that:

$$h : \mathbb{N}^k \mapsto \mathbb{N}$$
$$h(x_1, \ldots, x_k) = g(f_1(x_1, \ldots, x_k), \ldots, f_m(x_1, \ldots, x_k))$$

For $i \geq 0$ and a term T, we define T^i inductively as:

$$T^0 = T$$
$$T^i = \lambda \vec{x}^{!^i\mathsf{Nat}}.\mathsf{let}\ !\vec{x} = \vec{x}\ \mathsf{in}\ !(T^{i-1}\vec{x})$$

Let $q = max(q_i)$. We can derive

$$\emptyset \vdash^{\delta} G^{q+1} : !^{q+1}\mathsf{Nat}_1 \multimap \ldots \multimap !^{q+1}\mathsf{Nat}_m \multimap !^{p+q+1}\mathsf{Nat}$$

We can also derive

$$\emptyset \vdash^{\delta} F_i^{q-q_i} : !^{q-q_i}\mathsf{Nat}_1 \multimap \ldots \multimap !^{q-q_i}\mathsf{Nat}_k \multimap !^{q}\mathsf{Nat}$$

Then, applying coercion we get

$$\emptyset \vdash^{\delta} \mathcal{C}(F_i^{q-q_i}) : \mathsf{Nat}_1 \multimap \ldots \mathsf{Nat}_k \multimap !^{q}\mathsf{Nat}$$

and we derive

$$x_1 : (\delta + 1, \mathsf{Nat}), \ldots, x_k : (\delta + 1, \mathsf{Nat}) \vdash^{\delta} !(\mathcal{C}(F_i^{q-q_i})x_1 \ldots x_k) : !^{q+1}\mathsf{Nat}$$

Let $F_i' \equiv !(\mathcal{C}(F_i^{q-q_i})x_1 \ldots x_k)$. By application we get

$$x_1 : (\delta + 1, \mathsf{Nat}), \ldots, x_k : (\delta + 1, \mathsf{Nat}) \vdash^{\delta} G^{q+1}F_1' \ldots F_m' : !^{p+q+1}\mathsf{Nat}$$

We derive

$$\emptyset \vdash^{\delta} \lambda\vec{x}.\mathsf{let}\ !\vec{x} = \vec{x}\ \mathsf{in}\ G^{q+1}F_1' \ldots F_m' : !\mathsf{Nat}_1 \multimap \ldots \multimap !\mathsf{Nat}_m \multimap !^{p+q+1}\mathsf{Nat}$$

Applying coercion we get

$$\emptyset \vdash^{\delta} \mathcal{C}(\lambda\vec{x}^{!\mathsf{Nat}}.\mathsf{let}\ !\vec{x} = \vec{x}\ \mathsf{in}\ G^{q+1}F_1' \ldots F_m') : \mathsf{Nat}_1 \multimap \ldots \multimap \mathsf{Nat}_m \multimap !^{p+q+1}\mathsf{Nat}$$

Take

$$H = \mathcal{C}(\lambda\vec{x}^{!\mathsf{Nat}}.\mathsf{let}\ !\vec{x} = \vec{x}\ \mathsf{in}\ G^{q+1}F_1' \ldots F_m')$$

Proposition A.0.11 (composition is representable)**.** *$H \Vdash h$.*

Proof. We now have to show that for all M_i and n_i where $1 \leq i \leq k$ such that $M_i \Vdash n_i$ and $\emptyset \vdash^{\delta} M_i : \mathsf{Nat}$, we have $HM_1 \ldots M_k \Vdash h(n_1, \ldots, n_k)$. Since $F_i \Vdash f_i$, we have $F_iM_1 \ldots M_k \Vdash f_i(n_1, \ldots, n_k)$. Moreover $G \Vdash g$, hence

$$G(F_1M_1 \ldots M_k) \ldots (F_mM_1 \ldots M_k) \Vdash g(f_1(n_1, \ldots, n_k), \ldots, f_m(n_1, \ldots, n_k))$$

We can show that $HM_1 \ldots M_k \longrightarrow^* G(F_1M_1 \ldots M_k) \ldots (F_mM_1 \ldots M_k)$, hence

$$HM_1 \ldots M_k \Vdash g(f_1(n_1, \ldots, n_k), \ldots, f_m(n_1, \ldots, n_k))$$

Thus $H \Vdash h$. □

Bounded sums and products

Let f be a $k+1$-ary function $f : \mathbb{N}^{k+1} \to \mathbb{N}$, where

$$\emptyset \vdash F : \mathsf{Nat}_i \multimap \mathsf{Nat}_1 \multimap \ldots \multimap \mathsf{Nat}_k \multimap !^p\mathsf{Nat}$$

with $p \geq 0$ and $F \Vdash f$. We want to represent

- bounded sum: $\sum_{1 \leq i \leq n} f(i, x_1, \ldots, x_k)$
- bounded product: $\prod_{1 \leq i \leq n} f(i, x_1, \ldots, x_k)$

For this we are going to represent $h : \mathbb{N}^{k+1} \to \mathbb{N}$:

$$\begin{array}{l} h(0, x_1, \ldots, x_k) = f(0, x_1, \ldots, x_k) \\ h(n+1, x_1, \ldots, x_k) = g(f(n+1, x_1, \ldots, x_k), h(n, x_1, \ldots, x_k)) \end{array}$$

where g is a binary function standing for addition or multiplication, thus representable. More precisely we have $g : \mathbb{N}^2 \to \mathbb{N}$ such that $\emptyset \vdash^\delta G : \mathsf{Nat} \multimap \mathsf{Nat} \multimap \mathsf{Nat}$ and $G \Vdash g$.

For $i \geq 0$ and a term T we define T^i inductively:

$$\begin{array}{l} T^0 = Tx_1 \ldots x_k \\ T^i = \mathsf{let}\ !x_1 = x_1\ \mathsf{in}\ \ldots \mathsf{let}\ !x_k = x_k\ \mathsf{in}\ !T^{i-1} \end{array}$$

We define the following terms:

$$\begin{array}{l} ST = \lambda z.\langle S(\mathsf{fst}\ z), G^p(Fx_1 \ldots x_k(S(\mathsf{fst}\ z)))(\mathsf{snd}\ z)\rangle \\ \emptyset; x_1 : (\delta, \mathsf{Nat}), \ldots, x_k : (\delta, \mathsf{Nat}) \vdash^\delta ST : \mathsf{Nat} \times !^p\mathsf{Nat} \multimap \mathsf{Nat} \times !^p\mathsf{Nat} \end{array}$$

$$\begin{array}{l} EX = \lambda h.\mathsf{let}\ !h = h\ \mathsf{in}\ !\mathsf{snd}\ h\langle\overline{0}, Fx_1 \ldots x_k\overline{0}\rangle \\ \emptyset; x_1 : (\delta+1, \mathsf{Nat}), \ldots, x_k : (\delta+1, \mathsf{Nat}) \vdash^\delta EX : !(\mathsf{Nat} \times !^p\mathsf{Nat} \multimap \mathsf{Nat} \times !^p\mathsf{Nat}) \multimap !^{p+1}\mathsf{Nat} \end{array}$$

We derive

$$n : (\,\mathsf{Nat}), \vec{x} : (\delta, \mathsf{Nat}) \vdash^\delta \mathsf{let}\ !\vec{x} = \vec{x}\ \mathsf{in}\ \mathsf{let}\ !n = n\ \mathsf{in}\ \mathsf{gen\_it}\ !ST\ EX\ n : !^{p+1}\mathsf{Nat}$$

Let $R = \mathsf{let}\ !\vec{x} = \vec{x}\ \mathsf{in}\ \mathsf{let}\ !n = n\ \mathsf{in}\ \mathsf{gen\_it}\,!ST\,EX\,n$. By coercion and abstractions we get

$$\emptyset \vdash^\delta \mathcal{C}(\lambda n.\lambda\vec{x}.R) : \mathsf{Nat}_i \multimap \mathsf{Nat}_1 \multimap \ldots \multimap \mathsf{Nat}_k \multimap !^{p+1}\mathsf{Nat}$$

Take $H = \mathcal{C}(\lambda n.\lambda\vec{x}.R)$.

Proposition A.0.12 (bounded sum/product is representable)**.** *$H \Vdash h$.*

Proof. Given $M_i \Vdash i$ and $M_j \Vdash n_j$ with $1 \leq j \leq k$ and taking G for addition, we remark that

$$HM_iM_1 \ldots M_k \longrightarrow^* \overline{f(i, n_1, \ldots, n_k) + \ldots + f(1, n_1, \ldots, n_k) + f(0, n_1, \ldots, n_k)}$$

Hence $H \Vdash h$. □

Chapter 8

Proof of Theorem 8.2.23 (Quantitative adequacy). By induction on the typing of M.

(var) This case is immediate by substitution.

(reg,unit,int,arith) These cases are trivial, by definition of $\underline{\underline{R \vdash^\delta \mathsf{Unit}}}$, $\underline{\underline{R \vdash^\delta \mathsf{Reg}_r A}}$ and $\underline{\underline{R \vdash^\delta \mathsf{Int}}}$.

(weak) This case is just an application of $\leq$-saturation.

(lam) We have

$$\frac{R; \Gamma, y : (\lambda, A) \vdash^\delta N : B}{R; \Gamma \vdash^\delta \lambda y.N : A \multimap B}$$

We take $(\overline{V}, \overline{p}) \Vdash^\delta \Gamma$ and $(W, q) \in \underline{\underline{R \vdash^\delta A}}$. By induction hypothesis we know that for every $\mathcal{S}$ such that $\underline{R \vdash^\delta} \sqsubseteq \mathcal{S}$ we have $(N[\overline{V}/\overline{x}, W/y], w(N) + \sum \overline{p} + q) \in \underline{\underline{R \vdash^\delta B_{\mathcal{S}}}}$. Therefore $(\lambda y.N[\overline{V}/\overline{x}], w(N) + \sum \overline{p}) \in \underline{\underline{R \vdash^\delta A \multimap B}}$ by definition of the interpretation.

(app) We have

$$\frac{R; \Gamma \vdash^\delta M : A \multimap B \qquad R; \Delta \vdash^\delta N : A}{R; \Gamma, \Delta \vdash^\delta MN : B}$$

For simplicity assume the contexts Γ and Δ are empty (it does not change the argument). Take $\mathcal{S}$ such that $\underline{R \vdash^\delta} \sqsubseteq \mathcal{S}$. By induction hypothesis we have

$$(M, w(M)) \in \underline{\underline{R \vdash^\delta A \multimap B_{\mathcal{S}}}}$$
$$(N, w(N)) \in \underline{\underline{R \vdash^\delta A_{\mathcal{S}}}}$$

Take $(E, e) \in (\underline{\underline{R \vdash^\delta B_{\mathcal{S}}}})^{\perp_{\mathcal{S}}}$ and $(S, s) \in \mathcal{S}$. We want to show that

$$(\langle MN, E, S\rangle, e(w(M) + w(N) + \mathbf{3} + s)) \in \perp\!\!\!\perp$$

Since $\langle MN, E, S\rangle \longrightarrow \langle M, N \cdot E, S\rangle$, by $\longrightarrow$-saturation it suffices to show $(N \cdot E, x \mapsto e(w(N) + \mathbf{2} + x)) \in \underline{\underline{R \vdash^\delta A \multimap B}}^{\perp_{\mathcal{S}}}$. Take $(\lambda x.P, p) \in \underline{\underline{R \vdash^\delta A \multimap B}}$. Now we have to prove

$$(\lambda x.P, N \cdot E, S, e(w(N) + \mathbf{2} + p + s)) \in \perp\!\!\!\perp$$

But $\langle \lambda x.P, N \cdot E, S\rangle \longrightarrow \langle N, \lambda x.P \odot E, S\rangle$, so by $\longrightarrow$-saturation we only have to prove $(\lambda x.P \odot E, x \mapsto e(x + p + \mathbf{1})) \in \underline{\underline{R \vdash^\delta A}}^{\perp_{\mathcal{S}}}$. Let $(V_A, v_A) \in \underline{\underline{R \vdash^\delta A}}$. Then by definition $(P[V_A/x], p + v_A) \in \underline{\underline{R \vdash^\delta B_{\mathcal{S}}}}$ and we have

$$(\langle P[V_A/x], E, S\rangle, e(p + v_A + s)) \in \perp\!\!\!\perp$$

Since $\langle V_A, \lambda x.P \odot E, S\rangle \longrightarrow \langle P[V_A/x], E, S\rangle$, we conclude by $\longrightarrow$-saturation that $(\lambda x.P \odot E, x \mapsto e(x + p + \mathbf{1})) \in \underline{\underline{R \vdash^\delta A}}^{\perp_{\mathcal{S}}}$.

(set) We have

$$\frac{r : (\delta, \S C) \in R \qquad R; \Gamma \vdash^{\delta+1} V : C}{R; \Gamma \vdash^{\delta} \mathsf{set}(r, \S V) : \mathsf{Unit}}$$

Without loss of generality we assume V is closed and $\Gamma = \emptyset$. Let $\mathcal{S}$ such that $\underline{R \vdash^{\delta}} \sqsubseteq \mathcal{S}$, $(E, e) \in \underline{\underline{R \vdash^{\delta} \mathsf{Unit}}}^{\perp_{\mathcal{S}}}$ and $(S, s) \in \mathcal{S}$. We want to prove

$$(\langle \mathsf{set}(r, \S V), E, S\rangle, e(\S w(V) + \mathbf{2} + s)) \in \perp\!\!\!\perp$$

Since $\langle \mathsf{set}(r, \S V), E, S\rangle \longrightarrow \langle \star, E, S \uplus r \Leftarrow \S V\rangle$, by $\longrightarrow$-saturation if suffices to show

$$(\langle \star, E, S \uplus r \Leftarrow \S V\rangle, e(\mathbf{1} + s + \S w(V)) \in \perp\!\!\!\perp$$

which amouts to prove $(S \uplus r \Leftarrow \S V, s + \S w(V)) \in \mathcal{S}$. There must be a decomposition $(S, s) = (S_1 \uplus S_2, s_1 \uplus s_2)$ such that $(S_1, s_1) \in \underline{R \vdash^{\delta}}$ and $dom(S_2) = \{r_i \mid \delta_i < \delta\}$. By induction we have $(V, w(V)) \in \underline{\underline{R \vdash^{\delta+1} C}}$ and therefore $(S_1 \uplus r \Leftarrow \S V, s_1 + \S w(V)) \in \underline{R \vdash^{\delta}}$. This allows us to conclude $(S \uplus r \Leftarrow \S V, s + \S w(V)) \in \mathcal{S}$.

(get) We have that

$$\frac{r : (\delta, \S A) \in R}{R; \vdash^{\delta} \mathsf{get}(r) : \S A}$$

Let $\mathcal{S}$ be such that $\underline{R \vdash^{\delta}} \sqsubseteq \mathcal{S}$, $(E, e) \in \underline{\underline{R \vdash^{\delta} \S A}}^{\perp_{\mathcal{S}}}$ and $(S, s) \in \mathcal{S}$. There must be a decomposition $(S, s) = (S_1 \uplus S_2, s_1 \uplus s_2)$ such that $(S_1, s_1) \in \underline{R \vdash^{\delta}}$ and $dom(S_2) = \{r_i \mid \delta_i < \delta\}$. Therefore the value which is read must belong to S_1. By definition of the interpretation, any decomposition $(S_1, s_1) = (S' \uplus r \Leftarrow \S V, s' + \S p_V)$ is such that $(V, p_V) \in \underline{\underline{R \vdash^{\delta+1} A}}$. Clearly this entails $(\S V, \S p_V) \in \underline{\underline{R \vdash^{\delta} \S A}}$ and $(S' \uplus S_2, s' + s_2) \in \mathcal{S}$, and therefore

$$(\langle \S V, E, S' \uplus S_2\rangle, e(\S p_V + s' + s_2)) \in \perp\!\!\!\perp$$

Hence by $\longrightarrow$-saturation $(\langle get(r), E, S\rangle, e(\mathbf{1}+s))$ and we conclude $(get(r), \mathbf{1}) \in \underline{R \vdash^{\delta} \S A}_{\mathcal{S}}$.

(contr) We want to justify the contraction rule

$$\frac{R; \Gamma, z : (!, A), y : (!, A) \vdash^{\delta} M : B}{R; \Gamma, y : (!, A) \vdash^{\delta} M[z/y] : B}$$

We take $(\overline{W}, \overline{p}) \Vdash^{\delta} \Gamma$ and $(!V, !v) \in \underline{\underline{R \vdash^{\delta} !A}}$. Let $\mathcal{S}$ such that $\underline{R \vdash^{\delta}} \sqsubseteq \mathcal{S}$. By induction hypothesis we have $(M[\overline{\overline{W}/\overline{x}, V/y, V/z}], w(M) + \sum \overline{p} + !v + !v) \in \underline{R \vdash^{\delta} B}_{\mathcal{S}}$. Since $!v + !v \leq !v + \mathbf{1}$ by contraction of $! : \mathcal{M} \to \mathcal{M}$ we conclude $((M[\overline{\overline{W}/\overline{x}, z/y}])[V/z], w(M) + \sum \overline{p} + !v + \mathbf{1}) \in \underline{R \vdash^{B} R}_{\delta}\mathcal{S}$. The case where M is a value is similar.

($\mathsf{prom}_!$) We have

$$\frac{R; x : (\lambda, A) \vdash^{\delta+1} V : B}{R; x : (!, A) \vdash^{\delta} !V : !B}$$

Let $(V', p) \in \underline{R \vdash^{\delta+1} A}$. By induction we have $(V[V'/x], w(V) + p) \in \underline{R \vdash^{\delta+1} B}$ and therefore $((!V)[V'/x], !(w(V) + p)) \in \underline{R \vdash^{\delta} !B}$. By functoriality of $! : \mathcal{M} \to \mathcal{M}$, $!(w(V) + p) \leq F(w(V)) + !p$ and thus by $\leq$-closure, $((!V)[V'/x], F(w(M)) + !p) \in \underline{R \vdash^{\delta} !B}$. And this is what we want since $(!V', !p) \in \underline{R \vdash^{\delta} !A}$.

Notice that since we don't have $!(p + q) \leq !p + !q$ (the *monoidalness* property), we cannot handle more than one variable in the context as in the $\mathsf{prom}_\S$.

($\mathsf{prom}_\S$) We have

$$\frac{R; \Gamma_\lambda, \Delta_\lambda \vdash^{\delta+1} M : C}{R; \Gamma_\S, \Delta_! \vdash^{\delta} \S M : \S C}$$

Assume the variables associated to the context Γ and Δ are respectively noted $\overline{x}$ and $\overline{y}$. Suppose $(\overline{V}, \overline{p}) \Vdash^{\delta+1} \Gamma_\lambda$ and $(\overline{W}, \overline{q}) \Vdash^{\delta+1} \Delta_\lambda$. By induction hypothesis, for any $\mathcal{S}$ such that $\underline{R \vdash^{\delta+1}} \sqsubseteq \mathcal{S}$ we have

$$(M[\overline{V}/\overline{x}, \overline{W}/\overline{y}], w(M) + \sum \overline{p} + \sum \overline{q}) \in \underline{R \vdash^{\delta+1} C}_{\mathcal{S}}$$

Take $\mathcal{S}'$ such that $\underline{R \vdash^{\delta}} \sqsubseteq \mathcal{S}'$. By Lemma 8.2.24,

$$(\S(M[\overline{V}/\overline{x}, \overline{W}/\overline{y}]), \S(w(M) + \sum \overline{p} + \sum \overline{q}) + \mathbf{4}) \in \underline{R \vdash^{\delta} \S C}_{\mathcal{S}'}$$

By *weak dereliction* and *monoidalness* we have

$$\S(w(M) + \sum \overline{p} + \sum \overline{q}) \leq \S w(M) + \sum \S\overline{p} + \sum !\overline{q}$$

Therefore by $\leq$-saturation we conclude

$$(\S(M[\overline{V}/\overline{x}, \overline{W}/\overline{y}]), \S w(M) + \sum \S\overline{p} + \sum !\overline{q} + \mathbf{4}) \in \underline{R \vdash^{\delta} \S C}_{\mathcal{S}'}$$

which is what we want since $(\overline{V}, \S\overline{p}) \Vdash^{\delta} \Gamma_\S$ and $(\overline{W}, !\overline{q}) \Vdash^{\delta} \Delta_!$. The case where M is a value is similar to the case of the rule $\mathsf{prom}_!$, except that we use the properties *weak derelication* and *monoidalness* of $\S : \mathcal{M} \to \mathcal{M}$ instead of *functoriality* of $! : \mathcal{M} \to \mathcal{M}$.

($\mathsf{elim}_\dagger$) This case is similar for $\dagger \in \{!, \S\}$. We have

$$\frac{R; \Gamma \vdash^{\delta} V : \dagger B \qquad R; \Delta, x : (\dagger, B) \vdash^{\delta} M : C}{R; \Gamma, \Delta \vdash^{\delta} \mathsf{let}\ \dagger x = V\ \mathsf{in}\ M : C}$$

where the variables associated to the contexts Γ and Δ are respectively noted $\overline{y}$ and $\overline{z}$. We take $(\overline{V_\Gamma}, \overline{p}) \Vdash^{\delta} \Gamma$, $(\overline{V_\Delta}, \overline{q}) \Vdash^{\delta} \Delta$. By induction hypothesis we have

$$(V[\overline{V_\Gamma}/\overline{y}], w(V) + \sum \overline{p}) \in \underline{R \vdash^{\delta} \dagger B}$$

Therefore $V[\overline{V_\Gamma}/\overline{y}]$ must be of the shape $\dagger V'$. Let $\mathcal{S}$ be such that $\underline{R \vdash^\delta} \sqsubseteq \mathcal{S}$. By the other induction hypothesis we have

$$(M[\overline{V_\Delta}/\overline{z}, V'/x], w(M) + w(V) + \sum \overline{p} + \sum \overline{q}) \in \underline{R \vdash^\delta C_{\mathcal{S}}}$$

By $\longrightarrow$-saturation (by considering a context), we obtain

$$(\mathsf{let}\ \dagger x = V[\overline{V_\Gamma}/\overline{y}]\ \mathsf{in}\ M[\overline{V_\Delta}/\overline{z}], w(M)+w(V)+\sum \overline{p}+\sum \overline{q}+\mathbf{1}) \in \underline{R \vdash^\delta C_{\mathcal{S}}}$$

□

Abstract Controlling the resource consumption of programs is crucial: besides performance reasons, it has many applications in the field of computer security where *e.g.* mobile or embedded systems dispose of limited amounts of resources.

In this thesis, we develop static criteria to control the resource consumption of higher-order concurrent programs. Our starting point is the framework of Light Logics which has been extensively studied to control the complexity of higher-order functional programs through the proofs-as-programs correspondence. The contribution of this thesis is to extend this framework to higher-order concurrent programs. More generally, this thesis fits in the research field of Implicit Computational Complexity which aims at characterizing complexity classes by logical principles or language restrictions.

The criteria that we propose are purely syntactic and are developed gradually to control the computational time of programs in a finer and finer way: first, we show how to guarantee the termination of programs (finite time); then, we show how to guarantee the termination of programs in elementary time and last, we show how to guarantee the termination of programs in polynomial time. We also introduce type systems so that well-typed programs are guaranteed to terminate in bounded time *and* to return values. Finally, we show that the type systems capture some interesting concurrent programs that iterate functions producing side effects over inductive data structures.

In the last part, we study an alternative semantic method to control the resource consumption of higher-order imperative programs. The method is based on Dal Lago and Hofmann's quantitative realizability framework and allows to obtain various complexity bounds in a uniform way. This last part is joint work with Aloïs Brunel.

Résumé Contrôler la consommation en ressources des programmes informatiques est d'importance capitale, non seulement pour des raisons de performance, mais aussi pour des questions de sécurité quand par exemple certains systèmes mobiles ou embarqués disposent de quantités limitées de ressources.

Dans cette thèse, nous développons des critères statiques pour contrôler la consommation en ressources de programmes concurrents d'ordre supérieur. Nous prenons comme point de départ le cadre des Logiques Light qui a été étudié afin de contrôler la complexité de programmes fonctionnels d'ordre supérieur au moyen de la correspondance preuves-programmes. La contribution de cette thèse est d'étendre ce cadre aux programmes concurrents d'ordre supérieur. Plus généralement, cette thèse s'inscrit dans le domaine de la complexité implicite qui cherche à caractériser des classes de complexité par des principes logiques ou des restrictions de langage.

Les critères que nous proposons sont purement syntaxiques et sont développés graduellement afin de contrôler le temps de calcul des programmes de plus en plus finement: dans un premier temps nous montrons comment garantir la terminaison des programmes (temps fini), puis nous montrons comment garantir la terminaison des programmes en temps élémentaire, et enfin nous montrons comment garantir la terminaison des programmes en temps polynomial. Nous introduisons également des systèmes de types tels que les programmes bien typés terminent en temps borné *et* retournent des valeurs. Enfin, nous montrons que ces systèmes de types capturent des programmes concurrents intéressants qui itèrent des fonctions produisant des effets de bord sur des structures de données inductives.

Dans la dernière partie, nous étudions une méthode sémantique alternative afin de contrôler la consommation en ressources de programmes impératifs d'ordre supérieur. Cette méthode est basée sur la réalisabilité quantitative de Dal Lago et Hofmann et permet d'obtenir plusieurs bornes de complexité de manière uniforme. Cette dernière partie est un travail en collaboration avec Aloïs Brunel.