Tools and Techniques for the Verification of Modular Stateful Code

Mário José Parreira Pereira

université PARIS-SACLAY

NNT : 2018SACLS605

Thèse de doctorat

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

Tools and Techniques for the Verification of Modular Stateful Code

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud

Ecole doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Orsay, le 10/12/2018, par

MÁRIO JOSÉ PARREIRA PEREIRA

Composition du Jury :

Xavier Leroy Professeur, Collège de France (Inria Paris)	Président
Wolfgang Ahrendt Professor, Chalmers University of Technology (Department of Computer Science and Engineering)	Rapporteur
Jorge Sousa Pinto Professor, Universidade do Minho (HASLab, INESC TEC)	Rapporteur
Catherine Dubois Professeure, École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise (laboratoire Samovar)	Examinatrice
Laurent Fribourg Directeur de Recherche, LSV – ENS Paris-Saclay (CNRS)	Examinateur
Mihaela Sighireanu Maître de Conférences, Université Paris Diderot (IRIF)	Examinatrice
Jean-Christophe Filliâtre Directeur de Recherche, Université Paris-Sud (CNRS)	Directeur de thèse

Tools and Techniques for the Verification of Modular Stateful Code

Mário José Parreira Pereira

December 18, 2018

À Bárbara, ao Vitor, ao Padrinho e à Mãe. À toi, Papa...

Contents

List of Figures

List of Tables

If I have seen further than others, it is by standing upon the shoulders of giants.

Isaac Newton

Préface

Quelle aventure ! Me voilà arrivé, trois ans et demi après, au bout de la plus grande aventure de ma vie. Maintenant que j'écris ces dernières lignes de mon manuscrit, un drôle de mélange de sentiments m'envahit : d'une part, terminer une thèse c'est un moment d'énorme joie, aucune autre réalisation académique peut nous porter un tel sentiment ; d'autre part, la fin d'une si belle étape de la vie me donne un secret envie de tout refaire pour une deuxième fois. Je me sens un peu comme le jour d'aujourd'hui. Le matin est arrivé nuageux sur Paris. De la fenêtre de ma chambre, à la Maison du Portugal, je vois le jardin de la maison qui touche au bord du périphérique parisien. Le contraste entre ce petit coin vert et l'intense trafic de la capitale française est assez difficile à imaginer pour ceux qui n'ont pas le vrai privilège d'habiter la Cité Universitaire de Paris. Mais, en fait, je me rends compte maintenant que la vue de ma chambre représente fidèlement un petit résumé de mes années en tant que thésard : la tranquillité de la campagne sur le plateau de Saclay, mais aussi la vie agitée de cette magnifique ville qui est Paris.

Si je peux garder un si bon souvenir de ces dernières trois années et demie, c'est surtout grâce aux gens que j'ai peux rencontrer au fil de ces années. J'ai vraiment monté sur les épaules des nombreux géants et c'est pour ça que j'ai pu voir beaucoup plus loin. À tous, je souhaite d'être capable de remercier comme vous le méritez.

Paris, le 26 novembre 2018

Remerciements

Tout d'abord, je tiens à remercier profondément mon directeur de thèse, Jean-Christophe Filliâtre. C'est une tâche trop difficile pour moi d'utiliser des mots pour exprimer toute ma gratitude vers toi, en fait beaucoup plus difficile que toutes les questions et défis scientifiques que tu m'as posés tout au long de ces dernières années. Tes énormes qualités en tant que scientifique, enseignant et programmateur nous les connaissons tous. Par contre, j'appartiens maintenant au groupe des gens privilégiés qui t'ont eu comme directeur de thèse et qui ont pu témoigner tes qualités humaines. Ton encadrement toujours très attentif, tes mots d'encouragement mais aussi tes questions pointeuses sur l'arithmétique des ordinateurs (que j'ai raté la plupart du temps), feront pour toujours partie des plus beaux souvenirs de ma thèse. Merci beaucoup, Jean-Christophe. J'espère que, dans les années à venir, je puisse être à la hauteur de toi, en tant que chercheur mais aussi bien en tant qu'encadrant, enseignant et collègue.

Je remercie très chaleureusement mes deux rapporteurs, Jorge Sousa Pinto et Wolfgang Ahrendt. J'espère que mon travail soit à la hauteur de vos plus grandes exigences scientifiques. *Muito obrigado*; *vielen Dank* !

Catherine Dubois, Laurent Fribourg, Xavier Leroy et Mihaela Sighireanu me font l'honneur de faire partie de mon jury. Je vous remercie. Soutenir devant vous c'est pour moi, à la fois, une grande responsabilité mais aussi une grande honneur.

Pendant mon séjour au LRI, l'équipe VALS m'a toujours proportionné un cadre de travail exceptionnel. Les discussions au coin café, les savoureux gâteaux, les repas de Noël très animés, mais aussi l'engagement sur l'avenir professionnel des plus jeunes sont des caractéristiques très

remarquables de cette équipe de recherche. Je tiens à vous exprimer toute ma gratitude. Je serai pour toujours un vrai *VALS-ien*.

Je tiens à remercier très particulièrement à Andrei Paskevich. Tes conseils ont été précieux pour l'achèvement de ma thèse. Cela aurait été quasiment impossible pour moi de parvenir au terme de ce travail sans ton aide. Merci beaucoup pour tout, Andrei. Et merci encore pour ton engagement au développement de cet outil extraordinaire qui est Why3.

Ma gratitude envers mes collègues Léon Gondelman et Martin Clochard est énorme. Non seulement j'ai pu avoir le bonheur de travailler directement avec vous deux, comme j'ai eu la grande chance de partager mes découvertes scientifiques avec les meilleurs co-bureaux possibles. Je te remercie, Léon, pour toute ton amitié ; à Martin, je te remercie pour toutes tes critiques, toujours très constructives. Vous êtes, pour moi, une grande source d'inspiration.

Aussi important que de s'investir dans la recherche et dans les travaux de thèse, c'est de savoir se détendre. Moi, j'ai eu le grand privilège de porter le maillot de Midi Trente. À tous mes coéquipiers de cette belle association, un grand merci pour tous les moments conviviales au tour du match de football de chaque mercredi. Et si vous me permettez, voici un « et pour Midi Trente, hip hip » !

Je tiens à remercier profondément à ma famille de la Maison du Portugal. Avec vous, je me sentais toujours chez moi, toujours dans une ambiance très accueillante. C'est aussi grâce à vous que mes années à Paris sont remplies de bons moments et de très bons souvenirs.

De facto, nascer-se português é um privilégio que só compreendemos realmente quando saímos para lá das nossas fronteiras. E é graças às nossas gentes que aprendemos o valor da alma *lusíada*. Às minhas gentes portuguesas, o meu sentimento de gratidão é imensurável. Este será para sempre o vosso cantinho na minha tese. Aos meus amigos da Juventude Social-Democrata, pelo companheirismo e boa-disposição com que sempre me acolheram nas minhas visitas esporádicas a Portugal. Aos colegas do Release-UBI, aquela que foi a minha primeira "casa fora de casa", pois foi aqui que descobri que, afinal, gostava tanto de informática. Um abraço muito especial ao Professor Simão; hoje mais que nunca tenho a certeza que foi ele a lançar a primeira pedra desta tese. Aos meus Professores e colegas do DCC-FCUP, por me acolherem sempre tão bem. Finalmente, e projectando já o futuro, a todos os membros da equipa PLASTIC do centro NOVA-LINCS, o meu mais sincero agradecimento pela confiança que depositam em mim.

Quando passamos tanto tempo no mundo académico e no ensino superior, temos tendência a rapidamente esquecer aqueles que tanto nos ensinaram, numa altura da vida em que estar sentado numa sala de aula era tudo menos uma actividade entusiasmante. Tenho de deixar um agradecimento muito especial à Escola Secundária Quinta das Palmeiras, a casa onde mais tempo vivi até hoje. São 6 anos repletos de boas memórias. Sem querer magoar ninguém, o agradecimento mais emocionado vai para a Professora Albertina Leitão. Hoje, só desejo poder estar à altura de tudo aquilo que me ensinou. Et voilà, « tout vient à point à qui sait attendre ».

À minha família, as palavras são poucas para vos agradecer. Aos meus de Nogueira, de Cabeceiras de Basto e do Porto. À minha querida avó Alice, um obrigado muito especial e um abraço tão *arrochadinho*. Mas também à família *da* Fátima, a todos o meu muito obrigado. Aos pequeninos José e Tomás, que me mesmo antes de eu chegar ao fim desta tese já me conferiram o grau de « Tio ».

Às minhas primas-*irmãs* Alexandra e Luísa, o vosso primo tem muita dificuldade em vos agradecer como merecem. À Xana, porque me acolheu tão bem quando eu cheguei, com tantas dúvidas e incertezas à cidade de Paris e porque nunca me deixa esquecer a sua boa disposição e amizade. Porque seres a mais velha é mesmo assim, saber que podemos sempre contar contigo. À Luísa, que nunca me deixa esquecer que um dia também já fui adolescente e que são os pequenos gestos que têm um grande impacto na vida dos mais novos. Cá estarei sempre para te explicar matemática, mas também para me rir contigo e com a tua maneira tão particular de ver a vida.

Ao meu querido amigo João Machado, são tantas as saudades. O teu exemplo ajudou-me sempre a nunca desistir, a acreditar que poderia sempre alcançar aquilo que sonhamos. Que coragem a tua, *mermão*! Um grande abraço de força e esperança para ti e para os muitos anos que estão para vir.

Ao Paulo Fiadeiro, a quem tenho a sorte de chamar Padrinho. São tantos anos de aprendizagem e coisas boas. Pensar que tudo começou há mais de 10 anos, num computador velhinho e com o MatLab. Eis-me agora aqui, no fim do meu doutoramento em programação, a reflectir sobre as pequenas coisas que acabam por se revelar decisivas para a nossa caminhada. Muito obrigado, Padrinho, por tudo o que já passou e o que ainda está para vir.

Ao meu irmão Vitor Pereira, o meu melhor companheiro. Neste momento, em que há mais de 8000 quilómetros e um fuso horário de 9 horas a separar-nos, temos mais tendência a olhar para o que já passou. Tantas são as recordações boas de todos estes anos. E em todas, sou sempre o irmão mais velho, cheio de ilusões e pretensões para mudar o Mundo, sempre ao lado do seu irmão *Tó*. Hoje, continuo a querer mudar esse mesmo Mundo, ao lado do grande especialista em prova de programas e criptografia.

À minha Bárbara, os meus agradecimentos são eternos e difíceis de exprimir. Pensar no que este período de afastamento significou é agora ter a certeza que vale sempre a pena lutar por aquilo em que acreditamos, quando temos as pessoas certas junto a nós. Obrigado por fazeres esta caminhada ao meu lado. Obrigado por todo o teu apoio e carinho. *I love you.*

Mãe, obrigado. Obrigado por tudo, por tudo mesmo. Pela vida, pelo teu colo, pelo teu abraço, pelo teu olhar forte e pela tua força infinita ao longo destes últimos anos. Sei que te vais rir ao ler esta frase, mas se há aqueles que sonham em ser sábios é porque cresceram ao lado de quem já é sábio há muito tempo. Obrigado, Mãe, pessoa mais sábia que eu conheço.

Et maintenant, au plus grand des géants, je ne peux faire que te dédier ce manuscrit. Papa, je te parle en français parce que je t'imagine en ce moment à rigoler de mon accent, de ma difficulté de conjugaison des verbes irréguliers, toi qui parlais tellement bien le français. Je te parle deux jours après ton cinquante-cinquième anniversaire, et c'est à la fois émouvant mais aussi très apaisant pour moi de t'imaginer à cette âge-là. Ton regard tendre, toujours très haut, toujours avec ta douceur unique. Laisse-moi oser le dire, le regard d'un vieux père que deviendrait un jour un jeune grand-père. Plus que jamais, ça me paraît irréel que tu sois parti il y a 10 ans. Tu me manques tellement. Cette thèse, elle est pour toi, tout mon travail est pour toi. Laisse-moi imaginer ton sourire, je suis certain que tu es très heureux. Merci infiniment, papa ; *obrigado por tudo, Pai.* Je t'aime.

Conventions

This manuscript uses the following conventions.

Code snippets. Whenever we introduce code snippets, we make explicit the programming language we use for implementation via a label on the top-right corner of the snippet. For instance, the following presents an OCaml implementation of a function computing the height of a binary tree[1]:

OCaml

```
let rec height = function
  | Empty          -> 0
  | Node (l, _, r) -> 1 + max (height l) (height r)
```

Definitions. We surround new definitions of symbols/relations with two horizontal rules, one at the top and the other at the bottom, in the style of the *Types and Programming Languages* textbooks. For instance, the following is the definition of the Fibonnaci sequence for natural numbers:

$n \geq 0$ $\boxed{\mathcal{F}(n)}$

$$\begin{aligned} \mathcal{F}(0) &= 0 \\ \mathcal{F}(1) &= 1 \\ \mathcal{F}(n) &= \mathcal{F}(n-1) + \mathcal{F}(n-2) \text{ if } n \geq 2 \end{aligned}$$

On the top-left corner, we introduce a list of pre-conditions that we must satisfy to apply the defined symbol/relation. In the case above, we can only apply function $\mathcal{F}$ to a non-negative value. On the top-right corner, we give the defined symbol/relation and arguments, for readability purposes.

Sequences. Throughout this thesis, we use a horizontal bar to denote sequences of symbols. For instance, $\bar{x}$ denotes the sequence of variables x_1 to x_n, and $\bar{v}$ denotes the sequence of values v_1 to v_n.

Substitution. We use a standard, capture-avoiding definition of substitution. We denote substitutions as $e[\bar{x} \mapsto \bar{v}]$, meaning that every free occurrence of variable x_i in e is replaced by v_i.

[1] Even if this is an elegant, compact, and above all *correct* implementation, it can trigger a stack-overflow exception for very ill-balanced trees. Can the reader think of an implementation that does not present such a pitfall?

These machines have no common sense; they do exactly as they are told, no more and no less. This fact is the hardest concept to grasp when one first tries to use a computer.

Donald Knuth

1 Introduction

I am a programmer since the age of 15. This is one of those sentences that makes me feel truly happy. During these last years, I have experienced countless times the joys of writing computer programs. In particular, I keep a very sweet memory of the first program that I ever wrote: it was a Python script that printed to a terminal (how enthusiastic I felt in that moment using a computer terminal, such a mysterious object for me back then) a table containing integer distances from 1 to a certain n, expressed in the different metric units. But the most amazing part for me was that I could interact with this program. I could specify, as an input, how many lines of the table were to be printed (I was way far from knowing that I was just changing the upper bound of iteration). This little piece of code now makes me laugh, but back then I felt like I had just accomplished the most difficult task in the world.

In my first years as a programmer, I explored many different languages and programming approaches. After my first (very brief) experiences with Python, I spent some time programming in C and under the MatLab system. After that, I spent a long time around the new shiny feature of Microsoft's Visual Studio, the C# language. It is only during my first year as an undergraduate student at *Universidade da Beira Interior* that I discovered OCaml and, more generally, the functional paradigm. Now, when I look back, I am absolutely sure that this was a turning point in my life as a programmer. I felt like I was just rediscovering computer programming once again, but this time everything was so different. This was a much more *elegant* and *attractive* way of writing computer programs. Nowadays, I am using OCaml in a daily basis, either for work and personal projects.

Despite all the profound technical differences between programming languages, their syntax, their development environments, etc., one thing was not changing: my frustration in the presence of *bugs* in my code. Every time a program of mine did not work as expected (either it computed the wrong answer, crashed spectacularly with a *segmentation fault* error), I felt like I got lost in the middle of the desert. I was so disappointed every time I had the impression of having a bullet-proof code, just to discover that it failed again on the next input. I was struggling to understand that the computer was only doing what I was telling him to do, no more and no less. Back then, I wished for a method that I could use to convince myself, without any doubt, that the program I had spend so much time writing was, indeed, *correct*.

The need for reliable software. It is not only young programmers who struggle to write error-free programs. In real-life, defective software has been the cause of embarrassing, some times even dramatic, situations.

Recently, a bug was found in the sorting algorithm used by default in platforms such as Android SDK or OpenJDK. Such an algorithm, called *Timsort*, was initially proposed by Tim Peters in 2002 for Python, but was later ported to Java. In 2015, a team of researchers was able to demonstrate that the Timsort was, in fact, broken [52]. Interestingly, the authors of this work also implemented a test generator which is able to produce input values that break Timsort. With millions of Android devices being used every day all-around the globe, one easily can imagine the repercussions of such a bug.

The year of 2015 seems to have been particularly *intense* for software bugs. On April 2015, a failure during a daily system refresh shut down the point-of-sales systems of 7000 Starbucks stores in the United States and 1000 in Canada. Employees were forced to hand free coffee and tea to costumers, since they could not accept payment card neither register change. The problem was fixed a few hours later, nonetheless, the company is reported to have lost between 3 or 4 millions dollars that day[1]. It was certainly not a very good day for Starbucks, but a very happy one for *Latte Macchiato* lovers.

The opening of the brand new Heathrow Terminal 5, in 2008, came with the promise of a modern, efficient and *working* baggage handle system. This was built with the intention of carrying around the huge number of costumers luggage checked-in every day. Prior to the opening day, the system was tested with over 12000 pieces of luggage. It worked perfectly. Unfortunately, "real-life" scenarios stroke, and the system was not able to cope after Terminal's opening to the public. Ten days after inauguration, around 42000 bags were lost and over 500 flights were cancelled[2].

When it comes to privacy and security, everyone demands the highest guarantees from software. The OpenSSL cryptography library is widely used to implement the Transport Layer Security (TLS) encryption protocol that secures communications over the Internet. On April 2014, a security bug in the Heartbeat Extension of the OpenSSL implementation of TLS was reported. This bug was named *Heartbleed*. This bug is caused by a buffer over-read vulnerability in the implementation, and can be exploited by external attackers to get access to the memory of systems protected by the defective OpenSSL implementation. This means theft of server's private keys, users' session data, and passwords. At the time of disclosure of the bug, around half a million of authority-certified Internet's web servers were vulnerable to the attack[3].

Formal methods. The aforementioned examples are only a few among the many cases of problems caused by software bugs. A first approach to raise our confidence in a piece of code is to test it [115]. A carefully chosen battery of tests allows to detect many of the existing bugs. Nonetheless, the set of possible input states of a program is normally infinite, which makes it impossible to conceive a fully-exhaustive test suite. To quote Edsger Dijkstra, "testing shows the presence, not the absence of bugs". Even if a program responds correctly to all the supplied tests, we cannot state that this is a completely bug-free program.

When we seek higher confidence over a program, we must turn ourselves to the use of *formal methods* [111]. Such techniques employ mathematical reasoning in order to demonstrate that a program behaves as expected. The use of formal methods normally involves two distinct phases. First, one must describe the expected behavior of a program in a chosen formal language. This is what we normally refer to as the *program specification*. Being able to correctly specify a program's behavior is already a very challenging task. Secondly, one must *mathematically* prove that the

[1] `https://qz.com/391374/the-great-starbucks-outage-was-not-caused-by-hackers/`
[2] `http://news.bbc.co.uk/2/hi/uk_news/7314816.stm`
[3] `http://heartbleed.com/`

program actually conforms to the devised specification, which normally involves some mathematical interpretation of the program semantics. The complexity of the specification and associated program can turn this into a very difficult task. Within the field of formal methods, many different approaches have been developed to verify that a program conforms to a formal specification. We can mention *abstract interpretation*, *model checking*, *type systems*, and *deductive program verification*. The last two are of particular interest for the work of this thesis. A very important common aspect to all of these methods is that they are based on the *static analysis* discipline, *i.e.*, there is never the need to execute a program in order to establish its relation to the specification.

Abstract interpretation [44, 46] is a technique used to establish the absence of run-time errors of a program, such as arithmetic overflows, memory violations, and out-of-bounds array accesses. This technique works by building an over-approximation of all the possible states of a program, using what is called an *abstract domain*. An abstract domain consists of a symbolic representation of a set of the program's states. For instance, an abstract domain can assign an integer interval to a program variable, representing the set of possible values that variable can contain at a given point of the program. This can be exploited to report on possible arithmetic overflows in the value of some variable. However, if the assigned intervals are too wide, the analyzes can trigger false alarms. A typical strike for abstract interpretation users is the balance between a fine-grained abstract domain and the computational cost associated with a very precise analyzes. An example of a working tool, with a successful application in industry, is the `Astrée` static analyzer [47] for C programs. It has been applied by the Airbus Company to establish that software running in planes is free of run-time problems.

Model checking [35] is a technique in which a program is considered as a state transition system and specification is represented in some variant of temporal logic. Finite automata are a typical representation for such systems. This method works by exhaustive state exploration and symbolic execution. A variant that limits the analyzes to the first k steps of the system execution is called *bounded model checking* [19]. This technique employs SAT solving to efficiently verify properties about the system. It solves some of the problems associated with the search space of traditional model checking, which requires large amounts of memory in order to analyze complex systems. Bounded model checking has been successfully used in industrial hardware verification, by companies such as IBM, Intel and Compaq.

Type systems. Type systems represent a vast domain of research. Here, we present only a very small sub-set of this field, with a focus on the aspects that are most relevant to our use of type systems throughout this work. For a broader presentation, we invite the reader to consult the two books in the series *Types and Programming Languages* by Benjamin Pierce [125, 126].

Type systems are the most widely used formal technique for static program analysis. The success of type systems is explained mainly by its light-weighted interaction with the programmer. Indeed, type systems have a significant impact since they can be used without a profound knowledge of the back-end machinery. We refer to languages equipped with some form of type system as *statically typed*. Examples of such languages include Java, C, or C++, which demand for a programmer to only specify the type of each program variable. Languages such as OCaml, Haskell, or Scala even feature a form of *type inference* mechanism, which means the system can automatically compute most of the needed typing information.

The main purpose of type systems is to rule out program expressions that would lead to execution-time errors. This is done by classifying program expressions according to the set of values they can produce. For instance, 42 is said to be of type `int` to express the fact that it is an integer constant. The same principle applies to the constant `true` of type `bool`. When it comes to assign a type to a function, we must make explicit the type of all its arguments, as well as the type of the returned value. If we consider, for instance, a function `even` that decides whether its

argument is an even value, we must assign to this function the type `int -> bool`. According to this type, `even` can only be applied to integer arguments. The expression `even 42` is thus of type `bool`, but expression `even true` has no computational meaning. These kind of expressions are immediately rejected by type systems at compile-time.

The research field of type systems is founded on the seminal work by Alonzo Church [33]. In this work, Church introduces the *simple types* discipline, with the purpose of giving a rigorous definition of the class of well-behaved terms of his λ-calculus [34]. Since then, type systems have been the object of extensive scientific study worldwide. A main line of research focus on the use of type systems in the design of programming languages. Given the increasing complexity of programming languages, there is also a need for more expressive type systems. Polymorphic type systems are an example of such increase in expressiveness. Using polymorphic types, one has the great flexibility to soundly re-use the same piece of code with arguments of different types. Such class of type systems are direct results of the works by J. Roger Hindley [74] and Robin Milner [110]. In 1982, Luis Damas and Robin Milner present the algorithm $\mathcal{W}$ [49], a type inference algorithm for polymorphic types. Polymorphic type systems and algorithm $\mathcal{W}$ form the basis of modern functional programming languages, such as OCaml and Haskell.

Type systems are also used as an artifact to mechanically encode mathematical proofs. In particular, such proofs can represent statements about the functional correctness of a program. With no surprise, such line of work requires very expressive type systems. An example of such is the work by Martin Löf on the theory of dependent types [107]. Very succinctly, the idea is to allow types to be parameterized by terms of a language. This takes a step forward with respect to the polymorphic theory of types, where types can only be parameterized by other types. Dependent types introduce the idea that reasoning about the behavior of programs can be seen as a type-checking problem. An example of a type system based on dependent types is the Calculus of Inductive Constructions [123], by Christine Paulin-Mohring. Such a type system forms the kernel of the Coq proof assistant [140]. During the last few decades, Coq has been successfully applied in the formalization of mathematics, as well as in the verification of computer programs. Another example of a modern proof assistant based on the theory of dependent types is the Agda [42] proof assistant.

Deductive program verification. The work of this thesis takes place in the research field of *deductive program verification*. Deductive program verification is the activity of turning the correctness of a program into a mathematical statement, and then proving it. Here, the word "deductive" stresses that the proof of such statements is done by applying deductive reasoning. Surveys of this research field are given by Jean-Christophe Filliâtre [59] and by José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa in their textbook *Rigorous Software Development: An Introduction to Program Verification* [4].

The idea of mathematically proving the correctness of a computer program is, almost, as old as the activity of programming. The very first known proof of program dates back to 1949, and is due to Alan Turing [143]. In this paper "Checking a Large Routine", the author presents a program that computes the factorial of a number by repeated additions, together with a set of local assertions that, when verified correct individually, imply the correctness of the whole program.

The proof of Alan Turing already poses many of the questions that research in deductive program verification has been tackling over the course of years. One of the most important challenges in deductive verification is to understand how one connects a mathematical behavioral specification of a program with the program itself. Turing approaches this problem by associating each instruction with a logical assertion that must hold before execution, and a second one that must be respected after executing that instruction. The behavior of the program is thus specified via the whole set of intermediate assertions. The seminal works by Robert Floyd and Tony Hoare [70, 75]

present the bases of the first working method for the deductive verification of programs, with respect to a logical representation of its behavior. Such method is normally referred to as the *Floyd-Hoare logic*, or simply *Hoare Logic*. The idea is to represent the specification of a program via two logical formulas. We use the following notation:

$$\{\mathsf{P}\}\,\mathsf{C}\,\{\mathsf{Q}\}$$

The first is called the *pre-condition*, and states that a program C must execute in a state satisfying the conditions of P. The second one, which we refer to as the *postcondition*, states that the execution of C leads to a state that satisfies Q. This corresponds exactly to the notion of *Hoare triples*. A Hoare triple is said to be valid if for every state satisfying P in which we execute C, if execution of C terminates then the final state satisfies Q.

From a practical point of view, to entail the validity of a Hoare triple of a program, we must provide each individual instruction of the program with its pre- and postcondition. Hoare logic provides a set of *inference rules* that specify how to build valid Hoare triples. The validity of the program's Hoare triple follows from the composition of these rules. One of such rules is called the *consequence rule* and it is defined as follows:

$$\frac{\mathsf{P} \Rightarrow \mathsf{P}' \qquad \{\mathsf{P}'\}\,\mathsf{C}\,\{\mathsf{Q}'\} \qquad \mathsf{Q}' \Rightarrow \mathsf{Q}}{\{\mathsf{P}\}\,\mathsf{C}\,\{\mathsf{Q}\}}$$

This rule states that we can strengthen a pre-condition and weaken the postcondition of a valid Hoare triple. This rule is useful when composing the triples of sub-expressions in which the logical conditions are syntactically equal. The two premises containing implications are called *verification conditions*. In 1971, Tony Hoare uses the described method to prove the correctness of *FIND*, a program that rearranges the elements of an array so that the k-th element ends up at position k, with smaller elements to the left and greater elements to the right [76]. In this work, the author describes how the program behaves, together with a logical specification of such a behavior. The proof of correctness boils down to proving 18 verification conditions.

Building on the foundations of Hoare logic, many program logics have been proposed throughout the years, in order to tackle the verification of a specific class of programs. A successfully example is *separation logic*, introduced by John Reynolds in 2002 [135] as an extension to traditional Hoare logic to facilitate reasoning about heap-manipulating programs. Separation logic extends Hoare triples with the notion of *separation conjunction*. A triple of the form $\{\mathsf{P}_1 \star \mathsf{P}_2\}\,\mathsf{C}\,\{\mathsf{Q}_1 \star \mathsf{Q}_2\}$ is valid if program C is executed in a state where P_1 and P_2 are valid conditions on *separated* portions of the memory, and the execution terminates in a state where Q_1 and Q_2 state valid conditions on separated portions of memory.

The way one uses Hoare logic to verify the correctness of a program makes it hard to scale to the proof of larger programs. To build a valid triple for a whole program, one must provide the intermediate triples for each and every individual expression. This is a fastidious and error-prone task. Moreover, the presence of the consequence rule makes it difficult to build a tool that automates the process. Fortunately, Edsger Dijkstra presents in 1975 his work on the *calculus of weakest pre-conditions* [55]. This work introduces the foundations for a tremendous increase of automation in the process of deductive verification. Dijkstra observed that a programmer is only required to explicitly provide logical assertions in a few points of the program. The remaining intermediate assertions can be inferred automatically. From the practical point of view, Dijkstra proposes to take a program C and a user-supplied postcondition Q to compute the *weakest pre-condition*, noted $\mathcal{wp}(\mathsf{C}, \mathsf{Q})$, over the initial state of execution such that it terminates in a state satisfying the postcondition Q. The following Hoare triple is, thus, valid:

$$\{\mathcal{wp}(\mathsf{C}, \mathsf{Q})\}\,\mathsf{C}\,\{\mathsf{Q}\}$$

In particular, the validity of a Hoare triple of the form $\{P\}\, C\, \{Q\}$ is a consequence of the following verification condition:

$$P \Rightarrow wp(C, Q)$$

Note that during the computation of $wp(\cdot)$ other verification conditions are also generated, which must also be proved valid. Several verification tools are based in the weakest pre-condition calculus approach. These tools are commonly referred to as *verification condition generators*. We can cite as examples Dafny [97], KeY [2], VeriFast [81], VCC [38], Viper [114], SPARK2014 [108], and Why3 [21,65]. The last one is the verification tool used as the working environment of this thesis. We shall later present it in more detail.

Building the mathematical statement that entails the correction of a program, with respect to a supplied specification, is only one part of the deductive verification process. It now comes to the point where one must actually prove the validity of this statement. A possible solution is to use a pen and a paper to make such proof, similar to what Hoare did for his proof of the *FIND* algorithm. A much better approach amounts to using *theorem provers*, tools that can be used to write and/or check a formal proof. Interactive proof assistants, such as Agda, Coq, Isabelle [116], PVS [119], are a possible solution. However, these demand the user to conduct manually most of the deductive reasoning. During the last decades, we have witnessed the rise of tools that can automatically search for the proof of a given mathematical formula. Even if constrained by the limits of decidability, *automatic theorem provers* have shown impressive results, and are now widely accepted as the tools of choice when it comes to discharge verification conditions. In particular, the family of SMT (*Satisfability Modulo Theory*) solvers are nowadays a crucial ingredient in automatic proof of programs. We can cite Alt-Ergo [20], CVC4 [11], and Z3 [53] as successfully examples of this family of tools.

A rather different approach to deductive program verification builds on the principle of *refinement* [8], originally introduced by Dijkstra [54]. The idea is to begin with a very high-level, abstract, non-executable description of a program serving as a specification, and successively derive more concrete versions of the same program until we get a fully executable program. Each refinement step produces verification conditions that state the equivalence between the original program and the one obtained by refinement. The chain of valid refinement steps ensures that the final result of refinement is a *correct-by-construction* program, equivalent to the initial specification. This technique is the basis of the B method [1] and is implemented in the associated tool Atelier B.

Let us conclude with a small "hall-of-fame" of software projects built on top of deductive verification tools. The list we present here not only shows that it is nowadays feasible to apply deductive methods in the construction of realistic programs but, perhaps more importantly, it opens the perspective for a future where every piece of software can, in principle, be verified. CompCert [99] is a realistic, optimizing, efficient compiler for the C language, whose functional correctness was verified using the Coq proof assistant. The dimension of this system makes it a true monument of deductive verification. Not only this verified compiler shows how the use of an interactive proof assistant can scale to the proof of an industrial-sized program, but it also opens the path for complete toolchains of verified software, from verified source programs all the way down to certified low-level representations of those programs. A recently published example of a system building on CompCert is the work of José Bacelar Almeida *et al.* in a verified software stack for secure function evaluation [3]. This work also uses EasyCrypt [14], a tool for deductive reasoning on probabilistic computations with adversarial code. Another impressive verified development based on CompCert is the Verasco [82] static analyzer. Verasco is completely specified and verified in Coq. It employs the technique of abstract interpretation to establish the absence of run-time errors in programs written in CompCert subset of ISO C 1999. Specified and developed using the HOL4 interactive theorem prover, CakeML [93] is a formally verified compiler for a subset of the

Standard ML language. A major novelty of CakeML is that it is a bootstrapped development, meaning that all parts of the compiler are built inside the theorem prover. On a different domain of application, the Sel4 project [88] presents the formal verification of a complete, general-purpose operating system kernel. The proof is conducted using the Isabelle/HOL theorem prover. Finally, and to cite a remarkable application of automated verification tools, the Verisoft XT project [16] introduced a verified version of Microsoft's Hypervisor and its embedded operating system PikeOS. The proof is done using VCC and the SMT solver Z3.

The Why3 tool. Why3 is a framework that proposes a set of tools that allow the user to implement, specify, prove programs, as well as extract correct-by-construction versions of those programs. The use of Why3 is oriented towards automatic proofs, as it supports many external automatic theorem provers. Why3 can also call interactive proof assistants, such as Coq, Isabelle, or PVS, when a proof obligation cannot be automatically discharged.

Why3 comes with a programming language, WhyML, a dialect of the ML family. This language offers some features commonly found in functional languages, like pattern-matching, algebraic types and polymorphism, but also imperative constructions, like records with mutable fields and exceptions. Programs written in WhyML can be annotated with contracts, that is, pre- and post-conditions. The code itself can be annotated, for instance, with loop invariants or termination measures for loops and recursive functions. It is also possible to add intermediate assertions in the code to ease automatic proofs. The WhyML language allows the user to write ghost code [62], which represents code with no computational interest, used for specification and proof purposes only. The system uses the annotations to generate verification conditions via its implementation of a weakest precondition calculus.

WhyML is also a specification language. It features an extension of first-order logic with rank-1 polymorphic types, algebraic types, (co-)inductive predicates and recursive definitions [60], as well as a limited form of higher-order logic [36]. This logic is used to write theories for the purpose of modeling the behavior of programs. Such theories are most of the time axiomatic. Why3 standard library is formed of many logic theories of this kind, in particular for integer and floating point arithmetic, sets, dictionaries, and finite sequences. The Why3 standard library is fully available on the project web site, `http://why3.lri.fr`.

Once a program is implemented and fully specified in WhyML, the user can interact with the system via its graphical user interface [48], which is invoked via the `why3ide` command. Fig. 1.1 presents an example of a typical user session of `why3ide`. From the graphical interface, the user can select a verification condition and apply logical *transformations* (*tactics*, in the interactive theorem provers' vocabulary), in order to make such formulas easier to discharge. This is a lightweight mechanism of interactive proof inside the Why3 system. To try proving the validity of a verification condition, the user can call her prover of choice, typically a SMT solver. The prover can give the following answers upon a proof attempt: it is able to discharge the selected verification condition, in which case the graphical interface adds a green button on the left part of window; it answers "don't know", which means that either the prover is not able to complete the proof (it falls out of the prover capacities), or it simply abandoned the proof by lacking information in the context; "timeout", which means the prover is not able to complete the proof in the given time; or finally, it is able to refute the selected verification condition.

The feedback of solvers changes the way we proceed with our proof. On one hand, a successful proof of a generated VC incites us to move on to the next one, without posing many questions. On the other hand, an unsuccessful attempt can be explained by several different reasons: the specification is insufficient, the generated VC falls out of the solver capacity, or, very simply, there is a *bug* in the given code[4]. This is the reason why the automated approach to program verification

[4]Bugs in specification are also possible, but these are more trickier to notice and repair. For instance, if we

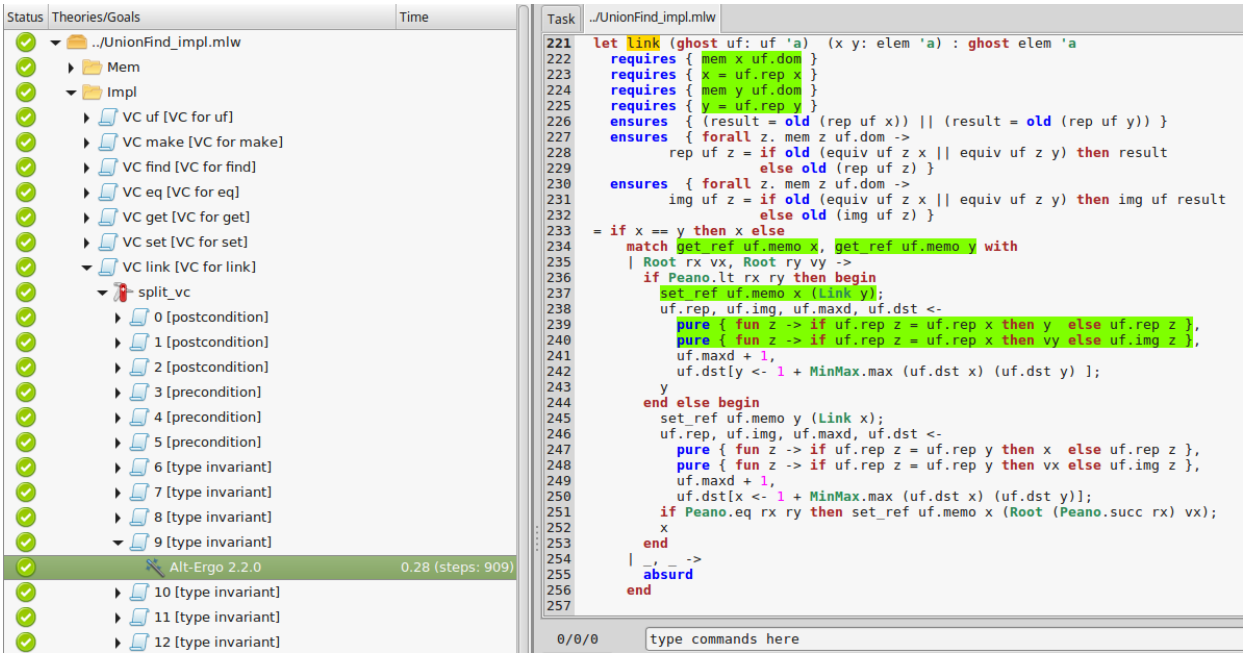


Figure 1.1: The Why3 Graphical User Interface.

is many times a *try-and-repair* process: negative feedback on a proof attempt naturally leads to some reparation of the code, reinforcement of specification, or maybe both.

Once the verification task is complete, *i.e.*, all the generated verification conditions are discharged, we can use the Why3 *code extraction mechanism* to generate a correct-by-construction executable code. The main task of the extraction phase is the removal of ghost code, without compromising the operational meaning of the verified WhyML program. Currently, Why3 supports extraction towards OCaml, CakeML, and the C language.

Why3 has been, until now, applied in the formal verification of several programs. The majority of theses examples is contained in the Why3 *Gallery of Verified Programs*, available online at `http://toccata.lri.fr/gallery/why3.en.html`. Despite its success, we feel Why3 still lacks application in a large-scale development. We believe that Why3 has reached a state of maturity that makes it feasible to be used in the development of a realistic verified software. Taking part in the development of a fully verified programming library is, perhaps, an interesting entry point for Why3 in the "revolution" that will lead us to a world of verified software.

The VOCaL project. VOCaL is an acronym for *Verified OCaml Library*. The name says it all: the great ambition of the VOCaL project[5] is to conceive a mechanically verified, general-purpose library for the OCaml language. We believe to build a "world of verified software" we must start by verifying libraries, the building blocks of any realistic software project. Even massively used and tested libraries can contain bugs, like the aforementioned cases of the mergesort and Timsort algorithms. We chose OCaml as our targeting language since this is the implementation language of systems used worldwide where stability, safety, and correctness are of utmost importance. Examples include the Coq proof assistant, the EasyCrypt proof assistant for cryptographic programs, the Astrée [47] and Frama-C [87] static analyzers, the Cubicle model-checker [40], and the Alt-Ergo theorem prover. Moreover, the verification of programming libraries has been, just until recently,

introduce a logically inconsistent specification, we might still get every generated verification condition discharged, even for a bugged program.

[5] `https://vocal.lri.fr/`

barely explored by the formal methods community, which makes the VOCaL project an interesting ground for research work. Other than the library itself, another foreseen contribution of VOCaL is the formal verification of parts of verification tools themselves.

One of the key ingredients of the VOCaL project is the design of a specification language for OCaml, independently of any verification tool. Another ingredient is the development of the verified library itself, using a combination of three tools: CFML [29], Coq, and Why3. These tools nicely complement each other: CFML implements a separation logic and targets pointer-based data structures; Coq is a tool of choice for purely applicative programs; and Why3 provides a high degree of automation using off-the-shelf SMT solvers. A consistent collaboration between these tools, keeping the benefits of each one, is one of the project's greatest challenges.

This thesis takes place right in the core of the VOCaL project. Over the course of this work, we have been developing the tools and techniques that allow Why3 to scale up to a verification framework that can successfully meet the goals of the VOCaL project. We detail on the contributions of this thesis in the following.

Contributions of this thesis. A major contribution of this thesis, with a direct impact on the VOCaL project, is the new Why3 code extraction mechanism. Our implementation effort resulted in an enhanced extraction mechanism for Why3, featuring a modular translation, up to OCaml functors. This new extraction mechanism has been successfully used to generate correct-by-construction implementations of several verified WhyML programs. In this thesis, we also present the mathematical formalization of a representative subset of the implemented extraction function. This formalization includes a proof that the extracted programs preserve the semantics behavior of the original source, either for convergent or divergent evaluations. We also show that the result of extracting a well-typed program is still a well-typed program. From a software architecture perspective, one key feature of the new extraction mechanism is that it clearly separates the code translation phase (removing logical annotations, erasing ghost code, optimizing superfluous `let..in` expressions) from code printing. Adding support for the extraction to a new language is now as simple as writing a printer from an intermediate representation to that language. The OCaml printer is also a contribution of ours.

Still in the context of the VOCaL project, we gave the first steps towards a specification language for OCaml. We have used this specification language to annotate several OCaml interface files. Such a specification is translated into the WhyML language via a Why3 plugin, whose development we also authored. Building on this translation tool and the Why3 extraction mechanism, we propose a toolchain/methodology to use Why3 as a platform to obtain verified OCaml programs. The translated specification links to a WhyML implementation via the Why3 refinement mechanism, which provides us with a proof that the verified implementation is a also a refinement of the specification written in the OCaml interface file. Using this toolchain, we were able to fed the VOCaL library with several *correct-by-construction* OCaml modules. These range from purely-applicative functorial data-structures such as pairing heaps, to strongly imperative ones such as a union-find library.

During our experiments with Why3 and the verification of OCaml programs, we came across the challenge of verifying arbitrarily pointer-based data structures. WhyML type system poses some constraints when it comes to reason about recursively-defined mutable data types. To circumvent some of these, we propose an approach based on an explicit memory model that we named *mini-heaps*. One distinctive aspect of this approach, contrarily to what is done in other tools that build memory models on top of Why3, is that we want to keep using WhyML types as much as possible, hence we only build a memory model for the parts of the program that cannot be encoded in the WhyML type system. Also, we do not introduce memory as a global mutable value of the whole program. We declare it as a private data type, and pass instances of the memory as a ghost

argument to each function, with the extraction mechanism erasing it from the final OCaml program. The WhyML type system guarantees that different fragments of the memory are separated.

Finally, we propose a formal specification that can be used to specify iteration processes, independently of the paradigm and underlying implementation. Such a specification is based on the use of two logical predicates to characterize the possible finite prefix sequences of elements enumerated during iteration. Our proposition is general enough to cope with the specification of non-terminating and/or non-deterministic processes. We have applied this approach to the paradigms of cursors and higher-order iterators, specifying and verifying implementations of both iteration providers and client code. We observe that our specification naturally imposes an abstract barrier between the client and the implementation of the iteration process. A key aspect of our methodology is that it is completely independent of the chosen verification tool. We used Why3 to conduct our experimental validation, but any other deductive verification tool could be used.

Plan. In Chap. 2, we use the example of a library of permutations as a gentle introduction to program verification within the Why3 framework. We explain how to write a WhyML version of the library, which includes both logical specification and implementation of functions manipulating mathematical permutations. We conclude this chapter by showing how to use the Why3 extraction mechanism to turn the verified library into a correct-by-construction, executable OCaml code.

In Chap. 3, we introduce a programming language called KidML. This language features recursive definition, local variables binding, conditional expressions, exceptions, mutable state, and, most importantly, ghost code. We adopt a step-by-step presentation of the KidML language, its type system, and operational semantics. The chapter terminates with a proof of type soundness.

In Chap. 4, we present and formalize an extraction procedure for the KidML language. For the most intrinsic cases of our extraction function, we provide examples of small KidML programs and the result of their extraction. We prove that the extraction function preserves well-typedness and semantics behavior of KidML expressions. We conclude with an overview of the actual Why3 extraction mechanism implementation, as well as some differences with respect to the procedure that is formalized in this chapter.

In Chap. 5, we propose a methodology to produce verified OCaml programs. Using the proof of an union-find library as a running example, we give a complete and detailed presentation of how our methodology works in practice. We dedicate the remaining of the chapter to presenting interesting challenges that arise during the proof of OCaml programs. For each one, we give an illustrative case study. We conclude with a statistical summary on the experimental use of our methodology.

In Chap. 6, we introduce a modular approach to reason about iteration. We apply this approach in the specification of two different iteration paradigms: cursors and higher-order iterators. For each one, we present the proof of several case studies (both iteration providers and client code).

In Chap. 7, we summary the main contributions of this thesis and draw some conclusions from our work. We conclude by enumerating some possible lines of future work.

The most important property of a program is whether it accomplishes the intentions of its user.

Tony Hoare

2

The Art of Program Verification, with Why3

We use a small library of functions manipulating permutations to illustrate how program verification is conducted in Why3. Our road-map is, in some sense, very simple. First, we write a WhyML version of such library, together with its logical specification. Then, we start what is the actual verification task. Within Why3, this mostly corresponds to call its *Verification Conditions Generator* and employ SMT solvers to discharge the generated verification conditions. Finally, we can use the Why3 *code extraction mechanism* to produce a *correct-by-construction* version of the library that can be compiled down to an executable.

In this chapter, we chose OCaml as the target language of our extraction. We stress that our approach to verification is oriented towards the *production* of an executable program from a proof, rather than taking an *existing* code written in a compilable programming language and prove it. We give, at the end of the chapter, the OCaml code of the permutations library, for illustration purposes.

2.1 A Library of Permutations

2.1.1 Notation

We use standard mathematical notation to better illustrate some of our examples. We use π to denote some permutation, and we define a permutation over the set $\{0, 1, \ldots, n-1\}$ as a bijection from this set to himself. A permutation can be represented using what is normally called the Cauchy's *two-line notation* [120]. A permutation π is written as a two-line matrix where the first line are the elements $x_0, x_1, \ldots, x_{n-1}$ of the set and the second one the images $\pi(x_i)$, for $0 \leq i < n$, as follows:

$$\begin{pmatrix} x_0 & x_1 & x_2 & \ldots & x_{n-1} \\ \pi(x_0) & \pi(x_1) & \pi(x_2) & \ldots & \pi(x_{n-1}) \end{pmatrix}$$

A permutation can also be represented using the *cycle notation*. For a permutation π, the *cycle* of x is the sequence $(x \;\; \pi(x) \;\; \pi^2(x) \ldots)$ of repeated applications of π to x until it returns back to x. This sequence of values forms the *orbit* of x. We continue this enumeration of cycles from π until all the elements of the permutation are written in some cycle. Let us consider, for instance, the

following permutation over elements from 0 to 5:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 4 & 2 \end{pmatrix}$$

One possible cycle representation for it is (0 1)(3 4 2). This representation is often referred to as the *decomposition into disjoint cycles*. Finally, we use $\|\pi\|$ to denote the number of elements in the permutation π.

2.1.2 Library Interface

We explain in this section the behavior of each function in the permutations library. The interface for this library is the following one:

OCaml
```
type t = int array

val id            : int -> t
val transposition : int -> int -> int -> t
val compose       : t -> t -> t
val inverse       : t -> t
val cycle_length  : t -> int -> int
```

We encode a permutation as an array of integer values, so the type `t` is just an alias for the type `int array`. Function `id` takes as an argument an integer `n` and returns the identity permutation from 0 to $n-1$. For instance, the array `[|0; 1; 2; 3; 4|]` is the result of applying `id` to 5.

The `transposition` function takes as arguments three integers `n`, `i`, and `j` and creates a permutation `t` of `n` elements where every element is mapped to itself, except for `i` and `j`, for which `t(i) = j` and `t(j) = i`. For instance, the call `transposition 5 0 4` returns the permutation

$$(0\,4)(1)(2)(3)$$

which is represented as the array `[|4; 1; 2; 3; 0|]`.

Function `compose` computes the composition of two given permutations, *i.e.*, given two permutations π_1 and π_2 such that $\|\pi_1\| = \|\pi_2\|$, this function returns the permutation π where $\pi(i) = \pi_2(\pi_1(i))$, for all $0 \leq i < \|\pi_1\|$. For instance, applying `compose` to π_1 =`[|1; 0; 3; 4; 2|]` and π_2 =`[|1; 4; 3; 0; 2|]`, we get the array `[|4; 1; 0; 2; 3|]`.

Function `inverse` says it all in the name: it takes a permutation `t` and computes its inverse. The inverse permutation of π, noted π^{-1}, is a permutation such that $\pi^{-1}(\pi(i)) = i$, for all $0 \leq i < \|\pi\|$. Let us consider the following permutation:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 4 & 3 & 0 & 2 \end{pmatrix}$$

Its inverse is the following permutation:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 3 & 0 & 4 & 2 & 1 \end{pmatrix}$$

Finally, function `cycle_length` is, likely, the most interesting one of the permutations library. It takes as arguments a permutation `t` and an integer `i`, and computes the length of the orbit of `i`, *i.e.*, the number `k` of elements we find, when traversing the orbit, before returning back to `i`. This is exactly how this function computes its result: it starts with `t(i)` and consecutively computes $t^2(i)$, $t^3(i)$, ..., until $t^k(i) = i$, for some $0 \leq k \leq \|t\|$.

2.2 Program Specification and Proof

In this section we describe the task of writing a WhyML version of the permutation library, together with its formal specification. The complete WhyML code for this development is given in Appendix A.

2.2.1 The Type of Permutations

Let us present in detail the type `t` of permutations. We declare it as the following record type:

```
type t = {                                                    WhyML
           a: array63;
  ghost  inv: array63;
  ghost size: int;
}
```

Field `a` represents the permutation itself and the two *ghost fields* `inv` and `size` represent, respectively, the inverse permutation of `a` and the number of elements in the permutation (we shall comment on the type `array63` later in this section). Field `size` is only introduced as a facility to write simpler code, as we can get the number of elements of an array in constant time. This is our first encounter with some form of *ghost code*, so let us take a moment to detail over the use of such an artifact. Ghost code represents a piece of arbitrary code (by arbitrary we mean that it can contain any feature from the WhyML language, such as assignments, loops, or function calls) that is introduced to ease the proof process. Having explicitly introduced field `inv` makes easier the reasoning about properties of permutations, increasing proof automation. This can be seen as a sort of *constructive reasoning*, since possessing the inverse permutation side-by-side with `a` avoids introducing some forms of existential quantification, as we will see in the forthcoming proofs. The interest of `size` is less obvious, since the length of an array is computed in constant time, but later in this section we will see how the value in this field actually helps SMT solvers to conclude some proofs, namely that of function `inverse`.

We refer to the dual of ghost code as *regular code*, *i.e.*, that part of the code that we want to compile and run on some hardware. This operational point of view quickly leads us to the concept of *code extraction*. By code extraction we mean a mechanical way of translating a program written in a proof language, for instance WhyML, into a programming language for which there exists a compiler, for instance OCaml. The main task of extraction is to erase any proof-related element, *including* any manifestation of ghost code. Taking type `t` declaration as an example, extracting such type declaration would result in

```
type t = { a: array63 }                                       WhyML
```

by erasing fields `inv` and `size`[1].

Type Invariant. Each field is very precisely related to the other two, in some kind of *logical connection*. We want to be able to verify that each value of type `t` we built actually respects such logical connection. In WhyML, we can wrap up a type definition with a *type invariant*, a logical formula that every inhabitant of that type must always satisfy. For the case of type `t`, this is as follows:

```
type t = {                                                    WhyML
           a: array63;
```

[1] It is a bit of a shame that after extraction we get a declaration of a record type with a single field. The Why3 extraction mechanism, described in Chap. 4, is able to optimize such singleton types.

```
  ghost  inv: array63;
  ghost size: int;
} invariant { length a = length inv = size }
  invariant { forall i. 0 <= i < size -> 0 <= a[i]   < size }
  invariant { forall i. 0 <= i < size -> 0 <= inv[i] < size }
  invariant { forall i. 0 <= i < size -> a[inv[i]] = i }
  invariant { forall i. 0 <= i < size -> inv[a[i]] = i }
  by { a = make 1 0; inv = make 1 0; size = 1 }
```

We start by asserting that both `a` and `inv` are arrays of the same length, equal to the value of `size`. The next two lines of the invariant state that the elements of `a` and `inv` are integers between `0` and `size`. The last two lines of the invariant closely follow the laws that define an inverse permutation. For a permutation in `a` and its inverse in `inv`, we have that `a[inv[i]] = i` and `inv[a[i]] = i`, for all $0 \leq$ `i` $<$ `size`. Actually, the last line of the invariant can be deduced from the previous ones. This requires, however, a non-trivial auxiliary lemma. By including it as part of the invariant, we rest assured that such property holds for every value we manipulate of type `t` since Why3 will generate a proof obligation for the preservation of the type invariant. Finally, the line starting with `by` is used to introduce a witness that there is at least an inhabitant of type `t` that respects this invariant. Such witness is an actual value of type `t`, in this case we chose the one where both `a` and `inv` are the array `[|0|]`. By exhibiting a witness for a type invariant we avoid declaring a new type with an inconsistent invariant, which prevents us from introducing an inconsistency in our proof context.

Mathematical and Computer Arithmetic. As users of WhyML, we are very quickly confronted with the decision on which types best match the purpose of some part of our program. A good example of such tension is to chose between unbounded mathematical integers or some form of machine arithmetic, both supported by WhyML. Mathematical integers are great to reason about, since they keep us closer to the logic of theorem provers, but prevent us from reasoning about integer overflows, which is crucial for any realistic software. On the other hand, using a fined-grain model of machine arithmetic makes it safe to translate integer values to native integers of some existing programming language, but makes it harder to reason about. It is, thus, very important to carefully chose the arithmetic type of each variable, as it can dramatically impact the outcome of our proof.

Fields of record type `t` are a perfect example of the duality between unbounded and machine integers. Let us start with field `size`: since it is introduced for specification purposes only, we are very pragmatic in our choice and declare it of type `int`, the WhyML predefined type of unbounded integers. As for fields `a` and `inv`, we chose the type `array63`, a WhyML model of arrays indexed by and containing 63-bits signed integers. The type `array63` is declared in the Why3 standard library as follows:

WhyML

```
type array63 = private {
  mutable ghost elts: seq int;
          ghost size: int;
} invariant { 0 <= size = length elts <= max_int }
  invariant { forall i. 0 <= i < length elts -> in_bounds elts[i] }
```

This type logically represents an array as a two two-fields records, where field `elts` is the finite mathematical sequence of integers that represents the elements of the array, and field `size` is the number of elements in the array. We note that field `elts` is declared as a mutable field, to account for possible modifications in the elements of the array. Whenever we want to change some element of the array, we need to create a new sequence and assign it to the `elts` field. This would

imply to re-prove, at each time, that the length of the array does not change. We overcome such potential hamper by stating in the invariant the condition `size = length elts` which, since `size` is declared as an immutable field, ensures that the length of a value of type `array63` is always the same.

The invariant of `array63` states that the length of sequence `elts` is, at most, equal to the constant `max_int` of type `int63`. The type `int63` is the WhyML type for 63-bits signed integers, declared as the following *range type*:

```
type int63 = < range -0x4000_0000_0000_0000 0x3fff_ffff_ffff_ffff >
```
WhyML

The purpose of a range type is self-explanatory: it is a Why3 artifact to introduce a numerical type whose values are within a certain limit. It takes the form `< range l u >`, where `l` and `u` represent, respectively, the minimum and maximum value allowed for this type. For `int63`, $\texttt{l} = -\texttt{0x4000\_0000\_0000\_0000} = -2^{62}$ and $\texttt{u} = \texttt{0x3fff\_ffff\_ffff\_ffff} = 2^{62} - 1$. This corresponds exactly to OCaml's built-in `int` type on 64-bit architectures, a 64-bit word where a bit is used for the sign and another bit is reserved for the *garbage collector* to distinguish between pointers and integer values. The condition `length elts <= max_int` ensures that an array of type `array63` can be safely indexed by a value of type `int63`.

The second line of type `array63` invariant describes the possible values an array of this type can contain. We state that, for each `i` within the bounds of sequence `elts`, the element `elts[i]` respects the predicate `in_bounds`. This predicate asserts that its argument is a value between `min_int` and `max_int`. In fact, this condition of `array63` invariant is what allows us to state it represents an array whose elements are 63-bit signed integers.

If we look carefully, it might seem that the invariant of `array63` is ill-typed. Let us examine in detail the first line of this invariant: here, `0`, `size`, and `length elts` are terms of type `int`, while `max_int` is of type `int63`. The operator `<=` takes two arguments of type `int` and returns a Boolean value. It seems that applying `<=` to `max_int` would result in a typing error. What happens in fact is that, after parsing this formula, Why3 is able to automatically apply a *coercion* from type `int63` to type `int`. Such coercion is declared in the `Int63` module, as follows:

```
function to_int (x : int63) : int = int63'int x
```
WhyML

```
meta coercion function to_int
```

Here, `int63'int` is an automatically generated logical function for the range type `int63`, which converts a value of type `int63` to the corresponding `int` value. When reading `length elts <= max_int`, we must actually think as if the coercion would be explicitly written, resulting in the following well-typed formula:

```
invariant { 0 <= size = length elts <= to_int max_int }
```
WhyML

Now, if we are even more watchful, we can spot other coercion applications in the invariant of type `t` of permutations. Indeed, the declaration of type `array63` comes along with the following coercion:

```
meta coercion function elts
```
WhyML

In Why3, record fields are treated as if they were regular functions, which is consistent with the concept of *projections*. There is no difference in declaring a symbol `f` as coercion, whether this symbol was introduced using the keyword `function` or as a record field. As we shall see in the following, the fact that Why3 treats record fields as regular functions allows for a certain degree of flexibility when accessing the values of nested records fields. Each application of the operator `[]` in the type invariant of `t` is, thus, an operation over sequences. Had we not used `elts` as a coercion, the invariant of type `t` would look like

```
invariant { length (elts a) = length (elts inv) = size }
invariant { forall i. 0 <= i < size -> 0 <= (elts a)[i]   < size }
invariant { forall i. 0 <= i < size -> 0 <= (elts inv)[i] < size }
invariant { forall i. 0 <= i < size -> (elts a)[(elts inv)[i]] = i }
invariant { forall i. 0 <= i < size -> (elts inv)[(elts a)[i]] = i }
```
WhyML

It would be quite painful to write, each time, the application of `elts` or `to_int`. We humans seem to make quite naturally these implicit conversions between types, thus a code without explicitly writing coercion functions is actually closer to what our brain expects.

2.2.2 Proving Library Functions

We focus now on the specification and proof of each function of the permutations library. For every function we present its functional specification, *i.e.*, its pre- and postconditions, as well as proof elements like loop invariants and termination measures.

Function `id`. Applying function `id` to a non-negative argument n creates a new permutation over integers between 0 and $n-1$, where each element is mapped to itself. For instance, `id` 5 returns the array `[|0; 1; 2; 3; 4|]`. This behavior is described using the following specification:

```
let id (n: int63) : t
  requires { 0 <= n }
  ensures  { result.size = n }
  ensures  { forall i. 0 <= i < n -> result.a[i] = i }
```
WhyML

Let us note that `n` is of type `int63`, which subsumes the application of the `to_int` coercion in the pre- and postconditions. The body of function `id` starts with the creation of a fresh array to store the permutation, as follows:

```
= let a = make n 0 in
  let ghost inv = make n 0 in
```
WhyML

Array `inv` is used to store the inverse permutation. We use here syntax **`let ghost inv...`** to explicitly mark `inv` as ghost variable, which forbids us from using it in regular code. This is yet another way to introduce pieces of ghost code in our program, other than declaring ghost record fields. The remaining of this function consists of a loop filling `a` and `inv` with values from 0 to $n-1$. This is as simple as the following **`for`** loop:

```
  for i = 0 to n - 1 do
    invariant { forall j. 0 <= j < i -> a[j] = inv[j] = j }
      a[i] <- i;
    inv[i] <- i
  done;
```
WhyML

The loop invariant is not a challenge neither for humans (to figure it out), neither for automated solvers (to prove its initialization and preservation), as it merely follows the code structure. Function `id` ends up naturally by returning the new value of type `t`:

```
  { size = to_int n; a = a; inv = inv }
```
WhyML

The verification condition generated by Why3 for function `id` is discharged in no time by SMT solvers. Although straightforwardly defined and easy to prove correct, function `id` is not without its subtleties. Let us consider the following alternative implementation for `id`:

```
let id (n: int63) : t                                                    WhyML
  requires { 0 <= n }
  ensures  { result.size = n }
  ensures  { forall i. 0 <= i < n -> result.a[i] = i }
= let a = make n 0 in
  for i = 0 to n - 1 do
    invariant { forall j. 0 <= j < i -> a[j] = j }
      a[i] <- i;
  done;
  { size = to_int n; a = a; inv = a }
```

Instead of manipulating the array `inv`, we assign directly the value of `a` to `inv`. From a programming point of view, this implementation is not satisfactory since it introduces a *memory aliasing* between two different fields of the same record. Each time we modify the value of `a` we would also modify the value of `inv`, which is surely not what we want. Nonetheless, such program is still accepted by the Why3's type system and the proof replays like a charm. A small test program, however, quickly reveals that this "does not mean what we think it means"[2]:

```
let bad_assignment () =                                                  WhyML
  let t = id 42 in
    t.a[0] <- 17;
  t.inv[0] <- 17
```

This program is refused by Why3 with the following, slightly cryptic, error message:

```
Error:
This expression makes a ghost modification in the non-ghost variable t
```

What is reported here as the problem, seems exactly what we want to do. We are aware that `inv` is a ghost field of `t`, and we want to modify it via a ghost modification. The reason why this piece of code is rejected lies deeply in the Why3 *type system with effects* [63]. Why3 features a type and effect system with singleton regions [141] that allows one to statically track all aliasing occurring in some program. So, for the case of function `id`, the system recognizes the alias between fields `a` and `inv` of the created record, and propagates this information to the (internal) type of `id`. Why3 is now mapping fields `a` and `inv` to point to the same singleton region, which means that whenever we modify the contents of `inv` the system knows that we modify the contents of `a`, as well. Since `a` is a regular field, modifying it through `inv` would result in ghost code modifying regular data, preventing the extraction mechanism to erase this piece of ghost code. We name such a phenomenon an *interference*, and we shall use this designation throughout this thesis. In Sec. 3.1.2, we introduce typing rules that take into account aliasing between regular and ghost fields, preventing us from typing function `bad_assignment`. Finally, the given version of function `id` where field `inv` is assigned the array `inv`, does not introduce the aliasing problem, and so function `bad_assignment` would be accepted by the Why3's type system.

Functions `transposition`, `compose`, and `inverse`. The next functions in the permutations library are, perhaps, even simpler to define and prove than `id`. The first one takes as arguments integer values `i`, `j`, and `n` and returns the transposition permutation of set $\{0, \ldots, n-1\}$ on `i` and `j`. It is specified and implemented as follows:

```
let transposition (i j n: int63) : t                                     WhyML
```

[2] *The Princess Bride*, 1987.

```
  requires { 0 <= i < n /\ 0 <= j < n }
  ensures  { result.size = n }
  ensures  { result.a[i] = j /\ result.a[j] = i }
  ensures  { forall k. 0 <= k < n -> k <> i -> k <> j -> result.a[k] = k }
= let t = id n in
  swap t.a i j;
  swap t.inv i j;
  t
```

The pre-condition requires that both `i` and `j` are elements of the set $\{0, \ldots, n-1\}$. The postcondition ensures that `i` and `j` are interchanged in array `t.a` and that for any other element `k` the permutation behaves as the identity function. Function `swap` is part of the `ArrayInt63` module and is straightforwardly defined. Functional correctness of function `transposition`, with respect to the given specification, is easily proved using SMT solvers.

The second one, `compose`, takes as arguments permutations `p` and `q` and computes the composition of these two permutations, *i.e.*, the permutation that maps each element `i` to `p(q(i))`. Its code and specification are as follows:

WhyML

```
let compose (p q: t) : t
  requires { p.size = q.size }
  ensures  { result.size = p.size }
  ensures  { forall i. 0 <= i < p.size -> result.a[i] = p.a[q.a[i]] }
= let n = p.a.length in
  let res = make n 0 in
  let ghost ires = make n 0 in
  for i = 0 to n - 1 do
    invariant { forall j. 0 <= j < i ->  res[j] = p.a  [q.a  [j]] }
    invariant { forall j. 0 <= j < i -> ires[j] = q.inv[p.inv[j]] }
     res[i] <- p.a  [q.a  [i]];
    ires[i] <- q.inv[p.inv[i]]
  done;
  { size = to_int n; a = res; inv = ires }
```

We state in the pre-condition that both `p` and `q` are permutations over the same set of elements. The postcondition establishes that the resulting permutation is also a permutation over the same set as the given permutations, where each element is mapped to the composition of `p` and `q`. The loop invariant straightforwardly follows the code structure. All the generated verification conditions generated for function `compose` are immediately proved by SMT solvers. Let us note that variable `ires` is declared as a ghost variable. This variable is only used inside ghost code, so it is of interest to erase it from the resulting extracted code. Had we not marked it with the keyword **`ghost`**, the extraction mechanism would still produce a correct code, but one containing variable `ires`. This would result in a useless computation, incurring some execution penalty for function `compose`. We can also observe a similar situation with variable `inv` in function `id`.

Finally, function `inverse` takes as argument a permutation `t` and computes its inverse. The specification of this function is easily deduced, as the inverse permutation is already stored in field `inv`. The complete code and specification is as follows:

WhyML

```
let inverse (t: t) : t
  ensures { result.size = t.size }
  ensures { forall i. 0 <= i < t.size -> result.a[i] = t.inv[i] }
= let n = t.a.length in
  let res = make n 0 in
```

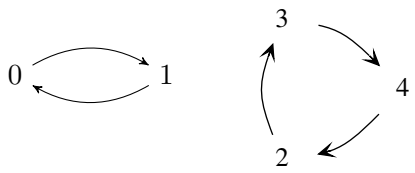
```
for i = 0 to n - 1 do
  invariant { forall j. 0 <= j < i -> res[t.a[j]] = j }
  res[t.a[i]] <- i
done;
{ size = to_int n; a = res; inv = copy t.a }
```

All the verification conditions generated for `inverse` are automatically proved. It is worth pointing out that, even if we already possess the inverse permutation, we cannot directly copy it to field `a`, *i.e.*, replace the last line of the `inverse` function by

```
{ size = to_int n; a = copy t.inv; inv = copy t.a }
```
WhyML

as the newly created record would be *contaminated* by ghost code. This means that the whole record would be considered as a ghost value, since the regular field `a` was assigned to a piece of ghost code. Function `inverse` would thus be considered a ghost function, getting erased at extraction time. On the other hand, copying the array `a` into field `inv` is not a problem, since assigning a regular value to a ghost field is not a contaminating operation. The typing rules we introduce in Sec. 3.1.2 account for both the contaminating and non-contaminating case.

Function `cycle_length`. Function `cycle_length` is the most challenging one in the permutations library. Let us consider the permutation $(0\,1)(3\,4\,2)$, schematically represented as follows:



The above diagrams show how to compute the length of a cycle in a permutation. Given an element `i` of the permutation, we follow the orbit of `i` until we get back to `i`. For the given example, the first orbit is of length 2 and the second one of length 3. A possible specification for `cycle_length` is the following one:

```
let cycle_length (t: t) (i: int63) : int63
  requires { 0 <= i < t.size }
  ensures  { exists s. 0 < length s = result /\
             forall j. 0 < j < result -> s[j] = t.a[s[j - 1]] /\
               s[0] = i /\ t.a[s[result - 1]] = i /\
               distinct s }
```
WhyML

The postcondition postulates the existence of a sequence `s` that represents, in fact, the orbit of `i`. This sequence has exactly the returned length of the cycle, denoted `result` in the postcondition, each element in the sequence is the image of its predecessor, its first element is `i`, the image of last element of `s` is also `i`, and finally the elements of `s` are pairwise distinct.

The given specification is logically correct and complete, and we could very well prove that the implementation of `cycle_length` respects it. However, the existential quantification can create a bottleneck in the proof, since instantiating an existential quantification remains a challenge for automated solvers. In order to make the task of provers more amenable, we can make function `cycle_length` returning the witness sequence, together with the integer value computed for the

length of the cycle. This is done in Why3 by declaring the function return type to be a tuple containing both regular and *ghost components*, as follows:

```
let cycle_length (t: t) (i: int63) : (n: int63, ghost s: seq int)
```
WhyML

The specification changes accordingly:

```
  requires { 0 <= i < t.size }
  ensures  { 0 < length s = n }
  ensures  { forall j. 0 < j < n -> s[j] = t.a[s[j - 1]] }
  ensures  { s[0] = i /\ t.a[s[n - 1]] = i }
  ensures  { distinct s }
```
WhyML

This introduces the notion of *partially ghost result*: the regular part is the actual computed result; ghost results are there to aid the proof task, for instance removing existential quantification in postconditions.

Function `cycle_length` body begins as follows:

```
= let n = ref 1 in
  let ghost s = ref (singleton (to_int i)) in
  let x = ref t.a[i] in
```
WhyML

WhyML uses the same vocabulary as other ML languages when it comes to references manipulation: expression `ref v` creates a fresh reference to `v`; expression `r := v` assigns `v` to reference `r`; finally, expression `!r` refers to the dereferencing operation, *i.e.*, it returns the value stored in the memory location denoted by `r`. Reference `n` stores the value that shall represent the length of the cycle of `i`, and that will be returned as the regular result of `cycle_length`. Reference `s` is introduced as a ghost reference and stores a value that is used only for specification purposes, *i.e.*, the sequence representing the orbit that is the ghost result of the function. Function `singleton` returns a fresh one-element sequence, which is here `to_int i`, the mathematical representation of the machine integer `i`. Finally, reference `x` stores the next element in the orbit, starting with the image of `i` in the permutation, `t.a[i]`.

The rest of function `cycle_length` consists of a loop that repeatedly computes the next element in the orbit, until it comes back to `i`. The code is as follows:

```
  while !x <> i do
    s := snoc !s (to_int !x);
    x := t.a[!x];
    incr n;
  done;
  !n, !s
```
WhyML

The `snoc` function appends to the end of the sequence, given as first argument, the element given as second argument. We note that the word `snoc` is the mirror of `cons`, a classical operation from the ML tradition that adds an element to the beginning of a sequence.

We give now, step-by-step, the *loop invariants* that allow us to deduce the postcondition of function `cycle_length`. The length of sequence `!s` is always positive and equal to the value of `!n`:

```
    invariant { 0 < length !s = !n }
```
WhyML

Reference `x` always contains a value belonging to the permutation, *i.e.*, between 0 and `t.size`, so does each index `j` of sequence `!s`:

```
    invariant { 0 <= !x < t.size }
    invariant { forall j. 0 <= j < !n -> 0 <= !s[j] < t.size }
```
WhyML

The first element of `!s` is always `i` while the last one is always `!x`:

```
invariant { !s[0] = i }                                          WhyML
invariant { !x = t.a[!s[!n - 1]] }
```

Sequence `!s` forms an iteration of the permutation:

```
invariant { forall j. 0 < j < !n -> !s[j] = t.a[!s[j - 1]] }     WhyML
```

Finally, we state that all elements of `s` are pairwise distinct:

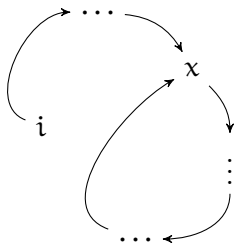
```
invariant { distinct !s }                                        WhyML
```

The verification conditions generated by Why3, regarding invariants preservation and initialization, are automatically discharged by a combination of SMT solvers. Also, this set of invariants allows us to deduce the given postcondition.

We focus now on verifying the safety of this code. The access `t.a[!x]` poses no problem since, by invariant, `0 <= !x < t.size` holds, and so this is a valid access. The other potential source of safety violation is the line `incr n`. Because we are using here machine integers, the call to `incr` generates a verification condition of the form

```
to_int n < max_int63                                             WhyML
```

where `to_int n` is the mathematical representation of the value `!n` before the call to `incr`. What this verification condition demands is a proof of *arithmetic overflow absence*, which in this case amounts to prove that the value of `!n` is strictly less than the constant `max_int63`. Intuitively, this is true because `!n` is the length of `!s`, which forms a sub-set of elements of the permutation `t.a`, so `!n` is bound by the length of `t.a`, another `int63` value. This is an easy reasoning for a human, but not so easy for automated provers, which explains why we are not able to discharge this verification condition using any SMT solver. Let us use *reductio ad absurdum* to verify that this property actually holds. The following diagram illustrates the situation where, while traversing the orbit of `i`, we would return to an already visited element that is not `i`:



Such situation contradicts the hypotheses that the permutation introduces an *injective function* on elements $\{0, \ldots, \texttt{t.size} - 1\}$, as well as the invariant `distinct !s`. We explore this contradiction by employing the *pigeonhole principle*: if the length of `!s` would be greater than `t.size`, then clearly some elements of `!s` would be repeated. We state the pigeonhole principle through the lemma given in Fig. 2.1. This is defined as a *lemma function*: a ghost, effect-free, terminating program whose contract is automatically translated to a lemma. Lemma functions come with the great flexibility that proving the property expressed in its contract is as simple as writing a program that verifies that property. This is not different from what we do for any program. In fact, this almost feels like the Curry-Howard correspondence *programs as proofs* [77]. For the `pigeonhole` function, we state that for any function `f` whose domain ranges from `0` to `n-1`, the co-domain ranges from `0` to `m-1`, and for which `m < n`, then there exists two different elements `i1` and `i2` for which `f i1 = f i2`. The proof proceeds as follows: we search for an `i`, $0 \leq \texttt{i} < \texttt{n}$, such that `f i = m - 1`. If we find such `i`, then we search for a `j`, $\texttt{i} + 1 \leq \texttt{j} < \texttt{n}$, such that `f j = m - 1`. If we find such `j`, then we are done. Otherwise, we remove `m - 1` from the co-domain of `f`, shrink

the domain to values between `0` and `n-2`, and re-start our search. Finally, if no `i` is found such that `f i = m - 1`, then we remove `m-1` from the co-domain of `f` and re-start our search.

The `pigeonhole` lemma function is turned automatically into the following lemma:

```
lemma pigeonhole: forall n m: int, f: int -> int.                    WhyML
  0 <= m < n -> forall i. 0 <= i < n -> 0 <= f i < m ->
  exists i1, i2. 0 <= i1 < i2 < n /\ f i1 = f i2
```

which is added to the proof context. However, even with such statement within our hypotheses, we are not able to prove the absence of overflow for the value `!n`. This lemma is actually difficult to instantiate by SMT solvers, mainly because they are unable to build the function to give as an argument to the lemma function. Since `pigeonhole` is treated by Why3 as any other program, we can instrument the code of `cycle_length` with an explicit call to `pigeonhole`. Passing to it the right arguments, we recover an instance of the postcondition that allows SMT solvers to discharge the remaining verification condition. We do this as follows:

```
while !x <> i do                                                     WhyML
  ...
  s := snoc !s (to_int !x);
  x := t.a[!x];
  if to_int !n + 1 > t.size then pigeonhole (length !s) t.size (get !s);
  incr n;
done;
```

The partial application `get !s` returns a function of type `int -> int` which we use as the argument of `pigeonhole`. This `if..then` expression introduces a logical contradiction: if it would be the case that `!n + 1 > t.size`, then the postcondition of `pigeonhole` gives us that `get !s i1 = get !s i2`, for two different indexes `i1` and `i2` of `!s`, which contradicts the hypotheses that all elements in `!s` are distinct. This contradiction is explored by SMT solvers to conclude the proof that `incr n` does not overflow. Let us note that, since `pigeonhole` is a ghost function, the whole `if..then` expression is considered ghost, and so is erased during extraction. In Chap. 3 we introduce typing rules that ensure such behavior.

The reasoning we just performed, using the pigeonhole principle, actually leads to a nice termination measure for function `cycle_length`. The proof that `!n` does not overflow amounts to the proof that the length of `!s` is never greater than `t.size`. We have just found our termination measure for the `while` loop: `t.size - length !s`. This value is a valid measure since it is non-negative, after the whole reasoning above, and it decreases at each iteration, as we always add a new element to `!s`. We add it as a *loop variant*, as follows:

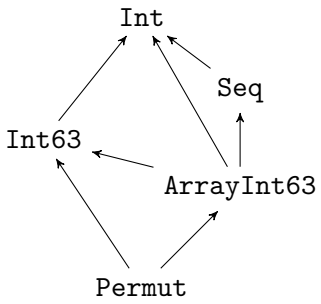
```
while !x <> i do                                                     WhyML
  variant { t.size - length !s }
```

The newly generated verification conditions regarding loop termination are proved almost immediately by SMT solvers, concluding the proof of function `cycle_length`.

To conclude this section, let us summarize how we use existing Why3 modules to build our library of permutations. Assuming the library is encapsulated in a WhyML module named `Permut`, its dependencies graph is the following one:

```
let rec lemma pigeonhole (n m: int) (f: int -> int)                    WhyML
  requires { 0 <= m < n }
  requires { forall i. 0 <= i < n -> 0 <= f i < m }
  ensures  { exists i1, i2. 0 <= i1 < i2 < n /\ f i1 = f i2 }
  variant  { m }
= for i = 0 to n - 1 do
    invariant { forall k. 0 <= k < i -> f k < m - 1 }
    if f i = m - 1 then begin
      for j = i + 1 to n - 1 do
        invariant { forall k. i < k < j -> f k < m - 1 }
        if f j = m - 1 then return
      done;
      let function g k = if k < i then f k else f (k + 1) in
      pigeonhole (n - 1) (m - 1) g;
      return end
  done;
  pigeonhole n (m - 1) f
```

Figure 2.1: Pigeonhole Principle.



Even if our permutations library is a rather modest size development, it features nonetheless an interesting composition of Why3 modules. The Why3's module system leads to a very comfortable modular development, abstracting away implementation details of used data structures, which lets us focus only on the specification and implementation of our library. Another interesting aspect of our development is the use of data structures that combine the use of mathematical and machine integers. Once again, the use of Why3 modules makes it very easy to compose definitions and types introduced for each numerical representation. Finally, the aforementioned coercion mechanism is a most valuable feature when it comes to write clearer and more concise code.

2.3 Code Extraction

The last mile in our quest to build a verified permutations library is to get an actual *executable implementation* of the verified functions. In this section, we describe how we use the Why3 extraction mechanism to generate a correct-by-construction OCaml implementation out of the Why3 proof. The current Why3 code extraction machinery is a contribution of this thesis. This is further detailed in Chap. 4.

Compared to the proof effort, the step from a verified WhyML implementation to an executable OCaml code is actually much simpler. To compile WhyML down to some existing programming

OCaml (extracted)

```
type t = (int array)

let id (n: int) : t =
  let a = Array.make n 0 in
  begin
    let o = n - 1 in let o1 = 0 in for i = o1 to o do a.(i) <- i done; a
  end

let swap (a: (int array)) (i1: int) (j: int) : unit =
  let v = a.(i1) in begin let o = a.(j) in a.(i1) <- o; a.(j) <- v end

let transposition (i1: int) (j: int) (n: int) : t =
  let t1 = id n in begin swap t1 i1 j; t1 end

let compose (p: t) (q: t) : t =
  let n = Array.length p in
  let res = Array.make n 0 in
  begin
    let o = n - 1 in
    let o1 = 0 in for i1 = o1 to o do res.(i1) <- (p.((q.(i1)))) done; res
  end

let inverse (t1: t) : t =
  let n = Array.length t1 in
  let res = Array.make n 0 in
  begin
    let o = n - 1 in
    let o1 = 0 in for i2 = o1 to o do res.((t1.(i2))) <- i2 done; res
  end

let cycle_length (t1: t) (i3: int) : int =
  let n = ref 1 in
  let x = ref (t1.(i3)) in
  begin
    while not ((!x) = i3) do
      begin let o = t1.((!x)) in x := o; incr n end done;
    (!n)
  end
```

Figure 2.2: Automatically Extracted OCaml Code.

language, OCaml in our case, we use the `extract` tool from the Why3 platform. The command line used to extract the permutations library is:

Terminal

```
> why3 extract -D ocaml64 --recursive -o permut.ml permut.mlw
```

We assume the permutations library to be contained in a Why3 file named `permut.mlw`. The output of the `extract` command is specified using the `-o` option. The `--recursive` option instructs the `extract` tool to perform a dependencies analysis on functions from file `permut.mlw`, recursively extracting any WhyML symbol (auxiliary function, type, or exception definition) on which the extracted code might depend on. For the case of the permutations library, `transposition` depends on the `swap` function from the Why3 standard library. The definition of `swap` is added to the file `permut.ml` only because we use the `--recursive` option.

The OCaml code obtained from running this command is given in Fig. 2.2. We argue that, even if this is an automatically extracted code, it remains readable and well-structured. A given WhyML code is translated into the Why3 internal AST using a variant of *A-normal form* [69], hence some `let o = ...` that remain in the OCaml code. Some extra parentheses are also added during extraction. This is explained by the fact that we are defensive regarding some substitutions introduced at extraction time, as we explain in the following.

One may wonder why other functions, for instance `Array.make` and `Array.copy`, are not recursively extracted. The reason lies in the argument of the `-D` option: `ocaml64` is the name of the *extraction driver*, a file to rule the whole extraction process. A driver is a text file that establishes a set of rules regarding the translation of identifiers during extraction. Let us take a look at a fragment from the `ocaml64` driver:

Driver

```
module mach.array.ArrayInt63
  syntax type array63 "(int array)"
  syntax val  length  "Array.length %1"
  syntax val  ([])    "%1.(%2)"
  syntax val  ([]<-)  "%1.(%2) <- %3"
  syntax val  make    "Array.make %1 %2"
  syntax val  copy    "Array.copy %1"
end
```

A driver consists of a set of declarations of the form `module M ... end`, where we specify a substitution rule for some of the identifiers defined in module `M`. Using this driver we assume a 64-bit architecture, hence the name `ocaml64`. Each line of the form

Syntax

```
syntax <decl_kind> <id> "<text_to_replace>"
```

settles that any occurrence of `<id>` is to be replaced by `<text_to_replace>`, where `<decl_kind>` defines the kind of symbol we want to translate. This ranges over program functions (`val` keyword), type symbols (`type` keyword), or exception declarations (`exception` keyword). The example above establishes a substitution for identifiers introduced in module `ArrayInt63`[3]. Using an extraction driver we can directly map some elements of our formal development to their OCaml counterpart, whenever these have a matching semantics. For instance, the type `array63` being a WhyML model for an array indexed by and containing 63-bits signed integers, we replace every occurrence of type `array63` by the OCaml type `int array`. The next line in the driver establishes a substitution rule for function `length`, defined in the Why3 standard library as follows:

WhyML

```
val length (a: array63) : int63
  ensures { to_int result = a.length }
```

[3]The `ArrayInt63` module is part of the Why3 standard library, contained in the file `array.mlw` of sub-directory `mach`.

This is a *non-defined* function whose behavior is completely characterized via its specification. Its semantics is exactly that of function `length` from the OCaml `Array` module, and so we add this rule in our driver. The syntax `"Array.length %1"` tells the code printer to take the first argument given to `length` in the WhyML code and apply it, as well, to `Array.length`. This illustrates an important use for extraction drivers: while WhyML allows to mix defined and undefined functions in the same context, the OCaml language does not[4]. It is, thus, required to replace any occurrence of a non-defined symbol by an OCaml counterpart. This is also the case for function `([])`, the direct access to the n-th element of an array[5], for which a substitution is defined in line

```
syntax val ([]) "%1.(%2)"                                    Driver
```

The fact that a rule for such function is given in the driver raises an interesting discussion: while function `length` is specified using only a postcondition, function `([])` is declared in the Why3 standard library with a pre-condition to prevent out of bounds accesses, as follows:

```
val ([]) (a: array63) (i: int63) : int63                     WhyML
  requires { 0 <= i < a.length }
  ensures  { to_int result = a[i] }
```

Nothing forbids us from taking a WhyML code calling `([])` where its pre-condition is not proved, and still extract it to OCaml. In such a case, we could obtain a code that fails at run-time with an index out of bounds exception. Let us make a statement to prevent such situation: we only extract fully-proved WhyML code. In particular, safety must have been proved, meaning no access out of bounds, for instance, is expected to occur while executing the extracted code. We stick to this hypotheses throughout this thesis. The remaining of the `ocaml64` driver has no significant difference with what we have already presented, so we do not detail it further.

We point out that the driver actually belongs to the trusted computing base of the extraction process. The substitutions we define in a driver are *textually* performed, *i.e.*, a symbol contained in the driver is directly replaced by the corresponding code written between quotes. A typo in the driver can lead to a faulty extracted code, even if the WhyML implementation has been completely verified.

[4] Our module system, mixing defined and undefined symbols, is actually close to that of Mixins [5]

[5] Why3 follows the OCaml convention, where the parentheses around the name of a function introduce an infix operation.

Well-typed programs do not go wrong.

Robin Milner

3 KidML

In this chapter, we present and formalize KidML, a programming language in the tradition of the ML family. KidML features recursive functions, conditional expressions, top-level declarations of new data types, and ghost code. This is a stateful language, as the programmer can mutate the value of a record field, declare and manipulate exceptions, and write divergent programs. We equip KidML with an operational semantics, in the form of a big-step evaluation judgment. To describe divergent programs, we introduce a co-evaluation relation by taking a co-inductive interpretation of the set of inductive rules that form the evaluation judgment. We also design a type system in order to rule out some KidML expressions, which would make evaluation to get stuck in some erroneous state. We present, in an incrementally way, the KidML syntactic constructions, as well as its semantics and type system in Sec. 3.1. The combination of ghost code and stateful traits makes it a non-trivial task to conceive a proper type system for KidML. The KidML programs accepted by our type system must guarantee the property of *non-interference* between ghost and regular code, *i.e.*, that we can safely erase ghost code from a program without jeopardizing its operational meaning. In Sec. 3.2, we provide a proof of type soundness with respect to the devised operational semantics.

The WhyML language is our main source of inspiration in the design of KidML. Many of the features we present here can already be found in the programming language of the Why3 verification framework. In fact, one can very well consider KidML as a sub-set of the WhyML language or, more precisely, a subset of the Why3 internal AST of a WhyML program. In Sec. 3.3, we report on the main differences between KidML and WhyML. Finally, let us mention that KidML is the input language of our code extraction algorithm presented in the next chapter.

3.1 The KidML Language, Step-by-Step

In this section we present the KidML language, step-by-step. We start with a simple core language containing few syntactic constructors and then gradually increment the language with new forms of expressions. Each time a new set of constructions is introduced, we give the corresponding semantics evaluation rules, as well as the type-checking rules. Representative examples of each new construction are also given.

3.1.1 Core Language

KidML core language is an effect-free deterministic language, with means to introduce *ghost code*. It is made up of the following syntactic constructions:

$$e \quad ::= \quad \bar{a} \mid \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \mid \texttt{ghost}\ e \mid \texttt{if}\ a\ \texttt{then}\ e\ \texttt{else}\ e$$

Let us elaborate on each syntactic construction. Here, $\bar{a}$ stands for a sequence of *atomic expressions*, where an atomic expression is either a variable or a value, as follows:

$$a ::= x \mid v$$

We use the bar-notation to introduce a (possibly empty) sequence of elements. When convenient, we use a comma to separate the different components of a sequence, as this is close to the ML tradition. We use, as well, some familiar terms to describe particular sequences, for instance *pair* and *triplet* to refer, respectively, to two- and three-element sequences. A *value* is a *constant*, denoted by c:

$$v ::= c$$

In KidML, constants range over integer numbers, *i.e.* 42, 73[1], or 1729; the Boolean values `true` and `false`; and the value `()`, the sole inhabitant of the `Unit` type.

Before further describing the syntax of our core language, let us introduce how we formalize its *operational semantics* and *type system*. We define the operational semantics of our language using a *big-step* style. Our evaluation judgment takes the form

$$e \Downarrow \bar{v}$$

and asserts that an expression e evaluates (after a finite number of steps) to a sequence of values $\bar{v}$. The following rule is the axiom of our judgment, and describes the evaluation of atomic expressions:

$$\frac{}{\bar{v} \Downarrow \bar{v}}\ (\textsc{EvalVBar})$$

We only define the semantics of closed expressions. No free variables can thus appear while evaluating a sequence of atomic expressions.

Our typing judgment is defined as a three-place predicate of the form

$$\Gamma \vdash e : \overline{\beta\tau}$$

asserting that, under a variable typing context Γ, an expression e is assigned the typing information $\overline{\beta\tau}$. We define Γ as a map from variable names to typing information of the form $\overline{\beta\tau}$. Here, $\beta\tau$ stands for a type τ annotated with a ghost status β, which we abbreviate to π in the following. The sequence $\overline{\beta}$ is called the *mask* of expression e. Type τ and ghost status β are defined as follows:

$$\begin{array}{lcl} \tau & ::= & \alpha \mid T\bar{\tau} \\ \beta & ::= & \texttt{reg} \mid \texttt{ghost} \\ \pi & ::= & \beta\tau \end{array}$$

A type τ is either a type variable α or the application of a type constructor T to the sequence $\bar{\tau}$. We assume that T ranges, at least, over the type `Int` for integer constants, type `Bool` for Boolean

[1] The numeral 73 is Sheldon Copper's, a fictional character from the *The Big Bang Theory* show, favorite number. It is introduced in the beginning of season 4, episode 10 (the 73th episode of the entire show), which we invite the reader to watch for a very entertaining moment around number theory.

$$\boxed{\overline{\beta_1} \sqsubseteq \overline{\beta_2}}$$

$$\frac{}{\overline{\texttt{reg} \sqsubseteq \beta}} \qquad \frac{}{\overline{\beta \sqsubseteq \texttt{ghost}}}$$

$$\frac{\|\overline{\beta_1}\| = \|\overline{\beta_2}\| \qquad \forall i.\ \beta_{1_i} \sqsubseteq \beta_{2_i}}{\overline{\beta_1} \sqsubseteq \overline{\beta_2}}$$

Figure 3.1: Masks order relation.

constants, and `Unit` for value (). The ghost status of a type is introduced using either the keyword `reg` marking a regular type, or the keyword `ghost` marking a ghost type.

The following describe the typing relation for atomic expressions and sequences of atomic expressions:

$$\frac{\texttt{Typeof}(c) = \tau}{\Gamma \vdash c : \texttt{reg}\,\tau}\ (\textsc{TConst}) \qquad \frac{\Gamma(x) = \beta\tau}{\Gamma \vdash x : \beta\tau}\ (\textsc{TVar}) \qquad \frac{\forall i.\ \Gamma \vdash a_i : \beta_i\tau_i}{\Gamma \vdash \bar{a} : \overline{\beta\tau}}\ (\textsc{TABar})$$

The rule **(TConst)** uses the `Typeof` oracle to retrieve the type of a constant c. Note that we consider a constant to be regular code, as stated in the conclusion of this rule. In rule **(TVar)**, we fetch the type of a variable x directly from environment Γ. The type of a variable is stored in Γ together with ghost status. Finally, we use the **(TABar)** rule to type a sequence of atomic expressions. Each premise of this rule is used to type each atomic expression a_i, using either **(TConst)** or **(TVar)**.

Let us return to the syntactic constructions of our language. The $\texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2$ expression is used to locally bind the result of evaluating e_1 in expression e_2. The syntax $\overline{\beta x}$ accounts for multiple variables binding. Each variable is bound together with its *ghost* status. We write $\texttt{reg}\,x$ to declare the variable x as *regular code*, while $\texttt{ghost}\,x$ indicates that x can only be used within *ghost code*. This is a light-weight mechanism to introduce ghost code in our language. To evaluate a `let .. in` expression we introduce the following big-step semantics rule:

$$\frac{e_1 \Downarrow \bar{v}' \qquad e_2[\bar{x} \mapsto \bar{v}'] \Downarrow \bar{v}}{\texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow \bar{v}}\ (\textsc{EvalLet})$$

The evaluation of a `let .. in` expression is rather intuitive: expression e_1 is first evaluated to a sequence of values $\bar{v}'$; second, $e_2[\bar{x} \mapsto \bar{v}']$ is evaluated to the sequence $\bar{v}$, which is the result of evaluating the whole `let .. in` expression. Let us recall that execution never takes into account the ghost status of variables. Indeed, from a semantics point of view, ghost and regular code are indistinguishable from each other. This will become even clearer once we give the evaluation rule for a $\texttt{ghost}\ e$ expression.

The following rule is used to assign a type to a `let .. in` expression:

$$\frac{\Gamma \vdash e_1 : \overline{\beta_1\tau_1} \qquad \overline{\beta_1} \sqsubseteq \overline{\beta} \qquad \Gamma + [\bar{x} : \overline{\beta\tau_1}] \vdash e_2 : \overline{\pi_2}}{\Gamma \vdash \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 : \overline{\pi_2}}\ (\textsc{TLet})$$

We briefly comment on the first and the last premises of this rule, as the second one deserves more attention. Expression e_1 is assigned the type sequence $\overline{\tau_1}$ together with mask $\overline{\beta_1}$. Expression e_2 is typed with $\overline{\pi_2}$ under the environment $\Gamma + [\bar{x} : \overline{\beta\tau_1}]$, the extension of Γ where each variable x_i is assigned type $\beta_i\tau_{1_i}$. Type $\overline{\pi_2}$ is the type assigned to the whole `let .. in` expression. Let us

now focus on the second premise of (**TLet**). It uses operator $\sqsubseteq$ to compare the ghost status $\overline{\beta}$ given for $\bar{x}$ with the mask $\overline{\beta_1}$ inferred for expression e_1. Comparing two masks simply amounts to comparing their components point-wise, as given in Fig. 3.1. The operator $\sqsubseteq$ is used in the (**TLet**) rule to forbid binding a ghost expression to a variable that we might have declared as a regular variable.

Let us resort to the code from Chapter 2 to illustrate how to use the typing rules presented so far. In page 20, the body of function `cycle_length` is as follows:

WhyML
```
let n = ref 1 in
let ghost s = ref (singleton (to_int i)) in
...
!n, !s
```

We can use rule (**TLet**) to derive a type for such expression, as follows:

$$\frac{\ldots \qquad \Gamma + [\mathtt{n} : \mathtt{reg\ ref\ int63};\ \mathtt{s} : \mathtt{ghost\ ref\ (seq\ int)}] \vdash \mathtt{!n},\ \mathtt{!s} : (\mathtt{reg\ int63},\ \mathtt{ghost\ seq\ int})}{\begin{array}{c}\Gamma \vdash \mathtt{let\ reg\ n = ref\ 1\ in\ let\ ghost\ s = ref\ (singleton\ (to\_int\ i))\ in} \\ \mathtt{!n},\ \mathtt{!s} : (\mathtt{reg\ int63},\ \mathtt{ghost\ seq\ int})\end{array}}$$

We explicitly omit the derivation trees for variables $\mathtt{n}$ and $\mathtt{s}$, since these are of no particular interest, as well as the premises comparing ghost status via operator $\sqsubseteq$. Even though the second component of the pair $\mathtt{!n}, \mathtt{!s}$ is ghost, the whole pair can only be seen as *partially* ghost, given the regular status of the first component. This is why, after extraction, this pair is converted simply into $\mathtt{!n}$ and the type of the function `cycle_length` becomes `reg int63`. How the extraction mechanism deals with partially ghost types and results, and why such an approach is sound, is the subject of the Chapter 4.

Besides locally binding a variable with a ghost status, another mean to introduce ghost code in KidML is by writing `ghost e`. In such a case, the whole expression is to be considered a ghost expression, independently of the status of its sub-expressions. The following typing rule precisely characterizes this behavior:

$$\frac{\Gamma \vdash e : \overline{\pi}}{\Gamma \vdash \mathtt{ghost}\ e : \text{👻}(\overline{\pi})}\ (\textsc{TGhost})$$

Function $\text{👻}(\cdot)$, which we designate as *type ghostification*, converts the type $\overline{\pi}$ into a *completely* ghost type, as follows:

$$\text{👻}(\overline{\beta\tau}) \quad \triangleq \quad \overline{\mathtt{ghost}\ \tau}$$

From a semantics perspective, the ghost status of an expression has no impact over its execution. The evaluation of `ghost e` simply *forgets* about the `ghost` keyword and continues evaluating e. This is formalized via the following rule:

$$\frac{e \Downarrow \bar{v}}{\mathtt{ghost}\ e \Downarrow \bar{v}}\ (\textsc{EvalGhost})$$

According to this rule, at execution time there is no need to distinguish between ghost and regular expressions.

We conclude the presentation of our core language with the conditional expression `if..then..else`. We note that the test of a conditional expression is necessarily an atomic expression. We use here a variant of *A-normal form* [69], which has the benefit of reducing the number of cases to consider when defining the semantics of KidML, without compromising its expressive power. Indeed, the evaluation of an `if..then..else` expression can be described using only the following two rules:

$$\frac{e_1 \Downarrow \bar{v}}{\mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2 \Downarrow \bar{v}}\ (\textsc{EvalIfTrue}) \qquad \frac{e_2 \Downarrow \bar{v}}{\mathtt{if\ false\ then}\ e_1\ \mathtt{else}\ e_2 \Downarrow \bar{v}}\ (\textsc{EvalIfFalse})$$

$\boxed{\overline{\beta}_1 \sqcup \overline{\beta}_2}$

$$
\begin{array}{rcl}
\texttt{ghost} \sqcup \beta & \triangleq & \texttt{ghost} \\
\texttt{reg} \sqcup \beta & \triangleq & \beta \\
\beta_1^1 \dots \beta_1^n \sqcup \beta_2^1 \dots \beta_2^n & \triangleq & \beta_1^1 \sqcup \beta_2^1 \dots \beta_1^n \sqcup \beta_2^n
\end{array}
$$

Figure 3.2: Mask Union.

We recall that we only define the evaluation relation for closed programs, so the only admissible atomic expressions for the conditional test are the `true` and `false` constants. Intuitively, it is easy to understand why the use of A-normal form does not limit the expressiveness of our language: every compound expression can be bound to a variable via a `let..in` binding. For instance, expression

```
if x > 42 then e1 else e2
```
KidML

is transformed into

```
let o = x > 42 in if o then e1 else e2
```
KidML

which fits in our syntax. We agree that it is sometimes painful to introduce all the necessary `let..in` to respect the use of A-normal formal. However, such program transformation can very easily be mechanized, and one could imagine it to be part of a pre-processor that converts expressions to the equivalent A-normal form ones.

In order to type an `if..then..else` expression, we distinguish two different cases, according to the ghost status of the test expression. We give thus two different typing rules, as follows:

$$
\frac{\Gamma \vdash a : \texttt{ghost Bool} \qquad \Gamma \vdash e_1 : \overline{\pi_1} \qquad \Gamma \vdash e_2 : \overline{\pi_2}}{\Gamma \vdash \texttt{if } a \texttt{ then } e_1 \texttt{ else } e_2 : \text{👻}(\overline{\pi_1} \cup \overline{\pi_2})} \ (\textsc{TIfGhost}) \qquad \frac{\Gamma \vdash a : \texttt{reg Bool} \qquad \Gamma \vdash e_1 : \overline{\pi_1} \qquad \Gamma \vdash e_2 : \overline{\pi_2}}{\Gamma \vdash \texttt{if } a \texttt{ then } e_1 \texttt{ else } e_2 : \overline{\pi_1} \cup \overline{\pi_2}} \ (\textsc{TIf})
$$

If the test is a ghost expression, as described in the (**TIfGhost**) rule, then the whole expression is contaminated, and we ghostify its type. Before describing the (**TIf**) rule, let us introduce the *type union* operator $\cup$, whose definition is given Fig. 3.3. This operation takes two typing information of the form $\overline{\pi_1}$ and $\overline{\pi_2}$ and performs what we refer to as *mask union*. The mask union operation is defined Fig. 3.2, and corresponds to the operation of merging two masks of the same length. During mask union, when doing $\beta_1 \sqcup \beta_2$, if either β_1 or β_2 is equal to `ghost`, the resulting mask is also `ghost`. This is of particular importance in the (**TIf**) rule: since the test is a regular expression, the final ghost status of the `if..then..else` expression is computed in the type union operation. Consequently, if one of the branches is a completely ghost expression, then it contaminates the whole `if..then..else` expression. This is exactly the case in function `cycle_length` of page 20, for the following expression:

```
if to_int !n + 1 > to_int t.size then pigeonhole (to_int t.size) !s
```
WhyML

Here, calling the ghost function `pigeonhole` contaminates the whole branch (contamination via function call is be discussed in Sec. 3.1.3), which in turns contaminates the whole `if..then..else`. The `else` can be omitted as it simply corresponds to the constant `()`, the `unit` value in the ML tradition. Given its ghost status, this expression is erased at extraction time, as shown in Fig. 2.2.

$\boxed{\overline{\pi_1} \cup \overline{\pi_2}}$

$$\overline{\beta_1 \tau} \cup \overline{\beta_2 \tau} \triangleq \overline{(\beta_1 \sqcup \beta_2)\tau}$$

Figure 3.3: Type Union.

3.1.2 Imperative Features

We extend our core language with common imperative constructions, namely *records* manipulation. Our definition of a record is standard: a record represents a collection of values stored together as one, where each component is identified by a different field name. Records are a useful mechanism to introduce new data types, which we refer to as *composite* data types, since they aggregate values of different types. The concept of record is fairly common in computer programming, and we can find it outside the realm of functional languages. For instance, in the C language we use *structures* to group items of different types under a single type name; in object-oriented languages, *objects* can be used for the same purpose.

We equip our language with the ability to define new data types via records, using the following syntax:

$$\texttt{type}\ \mathsf{T}\overline{\alpha} \quad = \quad \{\, \overline{\mathsf{f} : \pi} \,\}$$

The keyword **type** is followed by the name of the newly defined type together with a, possibly empty, vector of type variables to account for the definition of polymorphic data types. The definition of the composite type is enclosed within curly braces and consists of a sequence of pairs of the form $\mathsf{f}_i : \pi_i$, a field name followed by a type with a ghost status. We constraint type definitions to non-recursive record definitions only. The type `permutation` of Sec. 2.2.1 is defined in KidML as follows:

```
type t = { a: reg array63; inv: ghost array63; size: ghost int }
```
KidML

For readability purposes, we use a semicolon to separate each component of the record.

We extend KidML grammar of expressions to account for record manipulations, as follows:

$$e \quad ::= \quad \ldots \mid \{\, \overline{\mathsf{f} = \mathsf{a}} \,\} \mid \mathsf{a.f} \mid \mathsf{a.f} \leftarrow \mathsf{a}$$

As in the previous section, we detail over each new construction individually, giving for each one the corresponding typing and evaluation rules. Most of the material in this section is inspired by Chapter 13 of Benjamin Pierce's textbook *Types and Programming Languages* [125]. The first new construction is used to create a new value of a composite type. Within curly braces, we give a sequence of pairs $\mathsf{f}_i = \mathsf{a}_i$, meaning the value of atomic expression a_i is assigned to field f_i. Note that every field of the record must be given during construction. From an operational point of view, creating a record corresponds to the allocation of a memory block in the store. Thus, an expression of the form $\{\overline{\mathsf{f} = \mathsf{a}}\}$ evaluates down to the *memory location* that points to the beginning of the allocated block. We extend the class of KidML values accordingly:

$$v \quad ::= \quad \ldots \mid \mathsf{l}$$

Using locations, we build a very simply model of stores as a partial function from locations to the memory representation of a record, as follows:

$$\mu \quad \triangleq \quad \mathsf{l} \mapsto \{\, \overline{\mathsf{f} = v} \,\}$$

We use meta-variable μ to range over stores. We extend our evaluation relation, as well, to account for the presence of stores. It becomes a four-place predicate of the form

$$\mu \cdot e \Downarrow \mu' \cdot \bar{v}$$

where μ and μ' represent the *state* of the store before and after evaluating expression e, respectively. Returning μ' allows us to propagate the modified store to future evaluations. For instance, evaluating an expression of the form $\{\, \overline{f = a} \,\}$ adds to the store a fresh location pointing to a new record. This is formally described via the following rule:

$$\frac{l \notin \mathtt{dom}(\mu) \qquad \mu' = \mu[l \mapsto \{\, \overline{f = v} \,\}]}{\mu \cdot \{\, \overline{f = v} \,\} \Downarrow \mu' \cdot l} \text{ (EvalRecord)}$$

The first premise ensures that l is a fresh location, where function $\mathtt{dom}(\mu)$ returns the set of allocated location in μ. Notation $\mu[l \mapsto r]$ stands for a new store obtained by replacing the binding of l in μ to the record r, leaving other bindings unchanged. It creates a new biding if l is not in the domain of μ. We update operational semantics rules given in Sec. 3.1.1 to cope with the changes made on the evaluation relation:

$$\frac{}{\mu \cdot \bar{v} \Downarrow \mu \cdot \bar{v}} \text{ (EvalVBar)} \qquad \frac{\mu \cdot e_1 \Downarrow \mu' \cdot \bar{v}' \qquad \mu' \cdot e_2[\bar{x} \mapsto \bar{v}'] \Downarrow \mu'' \cdot \bar{v}}{\mu \cdot \mathtt{let}\ \overline{\beta x} = e_1\ \mathtt{in}\ e_2 \Downarrow \mu'' \cdot \bar{v}} \text{ (EvalLet)}$$

$$\frac{\mu \cdot e \Downarrow \mu' \cdot \bar{v}}{\mu \cdot \mathtt{ghost}\ e \Downarrow \mu' \cdot \bar{v}} \text{ (EvalGhost)} \qquad \frac{\mu \cdot e_1 \Downarrow \mu' \cdot \bar{v}}{\mu \cdot \mathtt{if}\ \mathtt{true}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow \mu' \cdot \bar{v}} \text{ (EvalIfTrue)}$$

$$\frac{\mu \cdot e_2 \Downarrow \mu' \cdot \bar{v}}{\mu \cdot \mathtt{if}\ \mathtt{false}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow \mu' \cdot \bar{v}} \text{ (EvalIfFalse)}$$

In order to type locations, we extend our typing judgment with a *store typing context* Σ, a function from locations to types, as follows:

$$\Gamma \cdot \Sigma \vdash e : \pi$$

To assign a type to a location l, we introduce the following typing rule:

$$\frac{\Sigma(l) = T\bar{\tau}}{\Gamma \cdot \Sigma \vdash l : \mathtt{reg}\ T\bar{\tau}} \text{ (TLoc)}$$

This is analogous to the type assignment of variables via rule **(TVar)**. The sequence $\bar{\tau}$ instantiates the sequence $\bar{\alpha}$, the arguments of type T. There are no remaining type variables in $\bar{\alpha}$. We note that, like we do for constants, we type every location as a regular value. Assigning every KidML value a regular status conforms to the fact that our semantic relation does not distinguish ghost and regular data. On the other hand, an expression $\{\, \overline{f = a} \,\}$ can be assigned either a regular or ghost status. We distinguish the two possibilities by adding the following rules to our type system:

$$\frac{\mathtt{type}\ T\bar{\alpha} = \{\, \overline{f : \beta_f \tau_f} \,\} \qquad \forall i.\ \Gamma \cdot \Sigma \vdash a_i : \beta_i \tau_{f_i}[\bar{\alpha} \mapsto \bar{\tau}] \qquad \forall i.\ \beta_i \sqsubseteq \beta_{f_i}}{\Gamma \cdot \Sigma \vdash \{\, \overline{f = a} \,\} : \mathtt{reg}\ T\bar{\tau}} \text{ (TRecord)}$$

$$\frac{\mathtt{type}\ T\bar{\alpha} = \{\, \overline{f : \beta_f \tau_f} \,\} \qquad \forall i.\ \Gamma \cdot \Sigma \vdash a_i : \beta_i \tau_{f_i}[\bar{\alpha} \mapsto \bar{\tau}] \qquad \exists i.\ \beta_i \not\sqsubseteq \beta_{f_i}}{\Gamma \cdot \Sigma \vdash \{\, \overline{f = a} \,\} : \mathtt{ghost}\ T\bar{\tau}} \text{ (TRecordGhost)}$$

Rules **(TRecord)** and **(TRecordGhost)** differ only in the third premise. For the latter rule, there exists at least one ghost expression a_i assigned to a regular field f_i. This contaminates the

whole record creation expression. The former establishes that all a_i have a smaller or equal ghost status than the one of field f_i. Let us note the use of substitution $[\overline{\alpha} \mapsto \overline{\tau}]$ in both rules. We use this type substitution to create an instance of the polymorphic type $T\overline{\alpha}$. As an illustrative example, we use rule **(TRecord)** to type the last line of function `id` (page 16):

$$\frac{\begin{array}{c}\texttt{type t} = \{\,\texttt{a} : \texttt{reg array63};\ \texttt{inv} : \texttt{ghost array63};\ \texttt{size} : \texttt{ghost int}\,\} \\ \Gamma(\texttt{n}) = \texttt{reg int} \qquad \Gamma(\texttt{a}) = \texttt{reg array63} \qquad \Gamma(\texttt{inv}) = \texttt{ghost array63} \qquad \ldots\end{array}}{\Gamma \cdot \emptyset \vdash \{\,\texttt{size} = \texttt{to\_int n};\ \texttt{a} = \texttt{a};\ \texttt{inv} = \texttt{inv}\} : \texttt{reg t}}$$

We build such typing derivation using an empty store typing, as there are no locations in scope. Environment Γ binds, at least, variables `n`, `a`, and `inv`. We do not give the premises concerning mask relation, as these are easily checked to be true. Nonetheless, we point out that `n` is a regular variable assigned to the ghost field `size` (function `to_int` returns a value with the same ghost status as its argument). Assigning a regular value to a ghost field respects the $\sqsubseteq$ relation, hence the `reg` status assigned to the whole expression.

The next construction we describe is the access to the contents of a record field, written $a.f$. The operational semantics of this construction is defined by the following rule:

$$\frac{\mu(l) = \{\,\ldots f_i = v_i \ldots\}}{\mu \cdot l.f_i \Downarrow \mu \cdot v_i}\ (\textsc{EvalGet})$$

We take a location l, retrieve its binding on μ, and return the value v_i associated with field f_i. We recall that we only define the semantics of closed programs, thus expression a in $a.f$ must be a location. Assigning a type to an expression of the form $a.f$ is rather intuitive: having defined a type $T\overline{\alpha}$ containing a field named f, we check if expression a is of type $T\overline{\tau}$. If so, the type of $a.f$ is the type defined for f. The following two rules implement this intuition:

$$\frac{\texttt{type}\ T\overline{\alpha} = \{\ldots,\ f : \beta_f \tau_f,\ \ldots\} \qquad \Gamma \cdot \Sigma \vdash a : \texttt{reg}\ T\overline{\tau}}{\Gamma \cdot \Sigma \vdash a.f : \beta_f\ \tau_f[\overline{\alpha} \mapsto \overline{\tau}]}\ (\textsc{TGet}) \qquad \frac{\texttt{type}\ T\overline{\alpha} = \{\ldots,\ f : \beta_f \tau_f,\ \ldots\} \qquad \Gamma \cdot \Sigma \vdash a : \texttt{ghost}\ T\overline{\tau}}{\Gamma \cdot \Sigma \vdash a.f : \texttt{ghost}\ \tau_f[\overline{\alpha} \mapsto \overline{\tau}]}\ (\textsc{TGetGhost})$$

We note that substitution $[\overline{\alpha} \mapsto \overline{\tau}]$ appears in conclusion and the type $T\overline{\tau}$ in premise, contrarily to rules **(TRecord)** and **(TRecordGhost)**. The ghost status assigned in the conclusion is computed as follows: if a is a regular expression, expression $a.f$ is given the status β_f of field f; if a is a ghost expression, we do not even need to inspect the status of f, the whole expression $a.f$ is a ghost expression. For instance, in the code of function `transposition` (page 17), both `t.a` and `t.inv` are typed using rule **(TGet)**, where the first one is assigned a regular status (`a` is a regular field), while the second one is assigned a ghost status (`inv` is a ghost field).

We finally detail on the expression $a.f \leftarrow a$. The purpose of such an expression is to modify the value associated to a field of an allocated record. This means a direct modification of memory, marking our first encounter with *mutability* in KidML. Other than being a useful way to define new data types, records also offer mechanisms to build modifiable values. In languages like OCaml or WhyML itself, a value of a record type is *immutable* by default, *i.e.*, it is not possible to directly modify the value of individual fields. Nonetheless, we can declare fields as *mutable*, which allows us to modify the value associated to that field via a side-effect. In fact, both in OCaml and in WhyML, mutable fields are the only means to introduce in-place modifiable values[2]. In the design of KidML, we take on a different road: every record field is a mutable field. This choice simplifies our type system and semantics evaluation as there is no need to check for the mutability of fields.

[2]The OCaml array type is a special case of mutable data structure. The `Array` module of the OCaml standard library is actually linked with external C code defining the type of arrays, an allocated block in the heap, and functions over arrays (`length`, `get`, or `set`, for instance).

In order to devise an evaluation rule for expression $a.f \leftarrow a$, we combine elements from both rules **(EvalRecord)** and **(EvalGet)**. Let us consider an expression of the form $l.f \leftarrow a$. We start by fetching the binding of l in the store. Then, we build up a new store where l points to a new record, where the value of field f is replaced by the value of expression a. The following rule materializes this description:

$$\frac{\mu(l) = \{\ldots\ f_i = v_i\ \ldots\} \qquad \mu' = \mu[l \mapsto \{\ldots\ f_i = v\ \ldots\}]}{\mu \cdot l.f_i \leftarrow v \Downarrow \mu' \cdot ()} \text{ (EvalAssign)}$$

The resulting store μ' contains exactly the same bindings as μ, except for l. The new store μ' represents a *functional update* of μ. Modifying the value of a record field is done via a side-effect, thus no significant value is to be returned during the evaluation of $l.f_i \leftarrow a$. We use, thus, constant $()$, the single inhabitant of type `Unit`, in the conclusion of **(EvalAssign)**.

The presence of mutability and side-effects has a significant impact in our typing relation. We refine the type assigned to an expression to include information about the *effects* triggered during the evaluation of such an expression. We update our typing judgment to the following relation:

$$\Gamma \cdot \Sigma \vdash e : (\overline{\pi},\ \epsilon_{\texttt{reg}},\ \epsilon_{\texttt{ghost}})$$

Keeping track of effects is what will allow us to use our system to show rich properties about well-typed programs, for instance, the absence of ghost code interference for well-typed programs. Our type system actually becomes a *type and effects system* [141]. We keep the type and ghost information $\overline{\pi}$ and add $\epsilon_{\texttt{reg}}$ and $\epsilon_{\texttt{ghost}}$ which stand for, respectively, the regular and ghost effects of e. As we will see throughout this chapter, separating the effects depending on whether they are produced via regular or ghost code, is quite useful when designing and reasoning about our type system. In the current setup, by effects we mean the modifications made on the store, which we refer to as the *writing effects* of an expression. Later, we shall extend the notion of effect with primitives affecting the execution flow of a program. We keep track of the changes in the store by gathering the name of locations and the corresponding fields that may have been modified through the execution of a program. For instance, if we consider the expression $x.i \leftarrow 42$, where x is a bound regular variable of a type containing a regular integer field i, we want to assign this expression the type $(\texttt{Unit},\ \emptyset + x.i,\ \emptyset)$. To do so, we add the following rule to our type system:

$$\frac{\begin{array}{c}\texttt{type}\ T\overline{\alpha} = \{\ldots, f : \texttt{reg}\ \tau_f, \ldots\} \\ \Gamma \cdot \Sigma \vdash a_l : (\texttt{reg}\ T\overline{\tau},\ \emptyset,\ \emptyset) \qquad \Gamma \cdot \Sigma \vdash a_r : (\texttt{reg}\ \tau_f[\overline{\alpha} \mapsto \overline{\tau}],\ \emptyset,\ \emptyset)\end{array}}{\Gamma \cdot \Sigma \vdash a_l.f \leftarrow a_r : (\texttt{Unit}, \emptyset + a_l.f,\ \emptyset)} \text{ (TAssign)}$$

The first premise is used to ensure that field f is actually part of the definition of type $T\overline{\alpha}$. We represent the empty effect as $\emptyset$. Atomic expressions are *pure expressions*, *i.e.*, they do not produce any kind of effect. The notation $\epsilon + a_l.f$ stands for the extension of effect ϵ with the writing effect $a_l.f$. For the given rule, $\emptyset + a_l.f$ corresponds to the singleton effect containing $a_l.f$. To illustrate how an expression can produce ghost effects, let us consider the very same expression $x.i \leftarrow 42$ but, this time, field f is declared of type ghost integer. Such an expression is typeable using the following rule:

$$\frac{\begin{array}{c}\texttt{type}\ T\overline{\alpha} = \{\ldots, f : \texttt{ghost}\ \tau_f, \ldots\} \\ \Gamma \cdot \Sigma \vdash a_l : (\beta_l T\overline{\tau},\ \emptyset,\ \emptyset) \qquad \Gamma \cdot \Sigma \vdash a_r : (\beta_r \tau_f[\overline{\alpha} \mapsto \overline{\tau}],\ \emptyset,\ \emptyset)\end{array}}{\Gamma \cdot \Sigma \vdash a_l.f \leftarrow a_r : (\texttt{Unit},\ \emptyset,\ \emptyset + a_l.f)} \text{ (TAssignGhostField)}$$

In this rule, we do not need to consider the ghost status of neither a_l or a_r. The ghost status of f immediately indicates that this expression produces a ghost effect. On the other hand, if f is

declared as a regular field, we need to inspect the status of a_l to type the expression $a_l.f \leftarrow a_r$. If a_r is a regular expression, we can resort to rule (**TAssign**). Otherwise, we need the following new rule:

$$\frac{\texttt{type}\ \mathsf{T}\overline{\alpha} = \{\ldots, f : \texttt{reg}\ \tau_f, \ldots\} \qquad \Gamma \cdot \Sigma \vdash a_l : (\texttt{ghost}\ \mathsf{T}\overline{\tau},\ \emptyset,\ \emptyset) \qquad \Gamma \cdot \Sigma \vdash a_r : (\beta_r \tau_f[\overline{\alpha} \mapsto \overline{\tau}],\ \emptyset,\ \emptyset)}{\Gamma \cdot \Sigma \vdash a_l.f \leftarrow a_r : (\texttt{Unit},\ \emptyset,\ \emptyset + a_l.f)}\ (\textsc{TAssignGhost})$$

In KidML, we use the type `Unit` to represent an empty list of results. No particular ghost status is thus attached to this type.

Let us consider the eight different ways to combine the ghost status of a_l, f, and a_r in the expression $a_l.f \leftarrow a_r$. As soon as f is a ghost field, we use rule (**TAssignGhostField**) to conclude the typing derivation. Such rule is used independently of the ghost status of either a_l and a_r, covering four out of the eight possible combinations. If we consider in turn a_l as a ghost expression and f a regular field, rule (**TAssignGhost**) is the one to be used. It applies both when a_r is a regular or a ghost expression, which adds two other possible expressions covered by our type system. Finally, the sole possibility to use rule (**TAssign**) happens when every element of expression $a_l.f \leftarrow a_r$ has a regular status. This adds up to seven possible typeable combinations under our typing rules. The only expression that we reject is the one where a_r is a ghost expression, while both a_l and f possess a regular status. An alternative type system, where, for instance, such an expression would be accepted as contaminated by the ghost status of a_r, would not be sound. We want the extraction mechanism to erase this expression, while it modifies regular data through a regular effect that we must preserve at the extracted program. This is a manifestation of ghost code interference. The type system of Why3 also rejects such kind of expressions, aborting execution with the following error message:

```
Error:
This expression makes a ghost modification in the non-ghost variable a_l
```

This is very similar to what we described for function `bad_assignment` in Sec. 2.2.2 (page 17). The error message is, in fact, the same on both cases. However, the reason why we must reject `bad_assignment` is more subtle. Let us take a step back and recall the definition of `array63`, the type assigned to fields `a` and `inv` of record type `t` (page 13). Type `array63` is defined as the following record type:

WhyML
```
type array63 = private {
  mutable ghost elts: seq int;
          ghost size: int;
}
```

Given such a definition, we can conclude that both fields `a` and `inv` store values that are memory locations. The former field is associated with a regular location, while the latter points to a ghost location. Indeed, without further restriction, the line `t.inv[0] <- 17` in function `bad_assign` is typeable under rule (**TAssignGhostField**). As we described in Sec. 2.2.2, the source of the problem lies in the erroneous version of function `id`. In line `{ size = to_int n; a = a; inv = a }`, we create an alias between fields `a` and `inv`, *i.e.*, an alias between a regular and a ghost field. This alias must be rejected by our type system, as it would violate the aforementioned condition of non-interference in well-typed programs. To this end, we must restrict the effects that can be assigned to an expression. We introduce the class of *admissible* effects, a constraint on type-checking rules to only assign effects that cannot jeopardize the static guarantees yielded by our type system. The admissible effects of an expression are defined as follows:

$$\texttt{adm}(\overline{\beta\tau},\ \epsilon_{\texttt{reg}},\ \epsilon_{\texttt{ghost}}) \triangleq \neg\texttt{reg}\ \texttt{writes}(\epsilon_{\texttt{ghost}})$$

Predicate `adm` holds if there are no regular writing effects in the set $\epsilon_{\texttt{ghost}}$, *i.e.*, no regular location is mutated via ghost code. This condition is added as a global side-condition of our type system, *i.e.*, we decorate each type-checking rule with the implicit premise that the type assigned in the conclusion must respect predicate `adm`. After this extension, function `bad_assignment` is no longer accepted by our type system.

To conclude this section, we show how to update the type-checking rules given so far to accommodate our type with effects setting. Let us establish some syntactic conventions that we shall use throughout the remaining of this thesis. For readability purposes, we denote the regular effects of an expression using meta-variable ϵ and the ghost effects with γ. We also use σ to abbreviate the type assigned to an expression, *i.e.*,

$$\sigma \quad ::= \quad (\overline{\beta\tau},\ \epsilon,\ \gamma)$$

The first modified rule we present is **(TLet)**. It changes as follows:

$$\frac{\Gamma \cdot \Sigma \vdash e_1 : (\overline{\beta_1\tau_1},\ \epsilon_1,\ \gamma_1) \qquad \overline{\beta_1} \sqsubseteq \overline{\beta} \qquad \Gamma + [\overline{x} : \overline{\beta\tau_1}] \cdot \Sigma \vdash e_2 : (\overline{\pi_2},\ \epsilon_2,\ \gamma_2)}{\Gamma \cdot \Sigma \vdash \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 : (\overline{\pi_2},\ \epsilon_1 \cup \epsilon_2,\ \gamma_1 \cup \gamma_2)}\ (\textsc{TLet})$$

Each individual expression e_1 and e_2 is now assigned an effect, respectively $(\epsilon_1,\ \gamma_1)$ and $(\epsilon_2,\ \gamma_2)$, which we combine to compute the effect of the whole `let..in` expression. Operation $\epsilon_1 \cup \epsilon_2$ (resp. $\gamma_1 \cup \gamma_2$) simply amounts to a set union operation. Next, **(TGhost)** is straightforwardly updated as follows:

$$\frac{\Gamma \cdot \Sigma \vdash e : \sigma}{\Gamma \cdot \Sigma \vdash \texttt{ghost}\ e : \text{👻}(\sigma)}\ (\textsc{TGhost})$$

Operation $\text{👻}(\cdot)$ is changed accordingly:

$$\text{👻}(\overline{\beta\tau},\ \emptyset,\ \gamma) \quad \triangleq \quad (\overline{\texttt{ghost}\ \tau},\ \emptyset,\ \gamma)$$

It is worth pointing out that $\text{👻}(\cdot)$ can only be applied to a type σ containing no regular effect. Ghostifying the type of an expression containing regular effects would, once again, result in ghost code interference. We implicitly add this pre-condition of $\text{👻}(\cdot)$ to every type-checking rule using the ghostification operation in its conclusion. Finally, modified rules for `if..then..else` expressions feature both ghostification and type union:

$$\frac{\Sigma \cdot \Gamma \vdash a : (\texttt{ghost}\ \texttt{Bool},\ \emptyset,\ \emptyset) \qquad \Sigma \cdot \Gamma \vdash e_1 : \sigma_1 \qquad \Sigma \cdot \Gamma \vdash e_2 : \sigma_2}{\Sigma \cdot \Gamma \vdash \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \text{👻}(\sigma_1 \cup \sigma_2)}\ (\textsc{TIfGhost}) \qquad \frac{\Sigma \cdot \Gamma \vdash a : (\texttt{reg}\ \texttt{Bool},\ \emptyset,\ \emptyset) \qquad \Sigma \cdot \Gamma \vdash e_1 : \sigma_1 \qquad \Sigma \cdot \Gamma \vdash e_2 : \sigma_2}{\Sigma \cdot \Gamma \vdash \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \sigma_1 \cup \sigma_2}\ (\textsc{TIf})$$

The type union operation is straightforwardly refined as follows:

$$(\overline{\beta_1\tau},\ \epsilon_1,\ \gamma_1) \cup (\overline{\beta_2\tau},\ \epsilon_2,\ \gamma_2) \quad \triangleq \quad ((\overline{\beta_1} \sqcup \overline{\beta_2})\overline{\tau},\ \epsilon_1 \cup \epsilon_2,\ \gamma_1 \cup \gamma_2)$$

As we can notice, rules **(TIf)** and **(TIfGhost)** combine the effects of both branches and assign it as the resulting effect of the whole `if..then..else` expression. From a static point of view, we are not able to decide which of the two branches is to be executed, and thus we introduce an *over-approximation* of the entire expression effects. This means the effects our type system assigns to an expression are, in fact, an *upper bound* of the effects triggered during the evaluation of such expression. Rules **(TConst)**, **(TVar)**, **(TABar)**, **(TRecord)**, **(TRecordGhost)**, **(TGet)**, **(TGetGhost)** are not presented here as they are trivially updated. We systematically add the typing store Σ and two empty sets for the regular and ghost effects, since all of these rules type check pure expressions.

$\boxed{\sigma_1 \sqsubseteq \sigma_2}$

$$\frac{\epsilon_1 = \epsilon_2 \qquad \gamma_1 = \gamma_2 \qquad \overline{\beta_1} \sqsubseteq \overline{\beta_2}}{(\overline{\beta_1}\tau,\ \epsilon_1,\ \gamma_1) \sqsubseteq (\overline{\beta_2}\tau,\ \epsilon_2,\ \gamma_2)}$$

Figure 3.4: Types Order Relation.

3.1.3 Function Definition and Function Call

In our quest to turn KidML into a more realist programming language, we turn our attention towards *functions*. In this section, we enrich our grammar of expressions with syntactic mechanisms to locally define and call functions, as follows:

$$e \quad ::= \quad \ldots \mid \texttt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e\ \texttt{in}\ e \mid f\langle\overline{\tau}\rangle(\bar{a}) \mid \texttt{rec}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e\ \texttt{in}\ e$$

Functions definition. Let us detail on the first construction. We introduce a binding for a function using the keyword **`fun`**, followed by the function signature: function's name, a (possibly empty) sequence $\overline{\alpha}$ of type variables, the function arguments enclosed within parentheses, and finally the return type σ. The body of a function is written before the **`in`** keyword, and it is bound in the expression after **`in`**. We use variable f to range over the names of functions, clearly disjoint from the class of variables introduced via a `let..in` expression.

Contrarily to OCaml and other functional languages, an expression in KidML cannot evaluate down to a function. The way KidML treat functions is, thus, much closer to the C language than to the functional paradigm. Regarding our evaluation relation, we introduce a map δ from a function name to its formal arguments and body, *i.e.*,

$$\delta \quad \triangleq \quad f \mapsto (\overline{x : \pi}, e)$$

which we add to our semantics predicate as follows:

$$\delta \cdot \mu \cdot e \Downarrow \mu' \cdot \bar{\nu}$$

We shall refer to δ in the following as a *procedure environment*. Let us note that we do not return a new map δ as a result of evaluation. Such a map stores, at each point of evaluation, the functional symbols that are in the scope of expression e. After completely evaluate e, we can remove from scope locally bound functions. The following rule establishes the semantics for a `fun..in` expression:

$$\frac{\delta[f \mapsto (\overline{x : \pi}, e_1)] \cdot \mu \cdot e_2 \Downarrow \mu' \cdot \bar{\nu}}{\delta \cdot \mu \cdot \texttt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 \Downarrow \mu' \cdot \bar{\nu}}\ (\textsc{EvalFun})$$

In the above rule, notation $\delta[f \mapsto (\overline{x : \pi}, e_1)]$ stands for the functional update of δ with the new binding of f to $(\overline{x : \pi}, e_1)$. The semantics of a `fun..in` expression closely resembles that given in rule **(EvalLet)**, except that we do not evaluate expression e_1, the body of function f.

We extend our typing relation with a *functions typing context* Δ to account for the introduction of the procedure environment in the semantics side. Our typing judgment takes the following form:

$$\Delta \cdot \Gamma \cdot \Sigma \vdash e : \sigma$$

This is akin to what we did when we added the store to the evaluation relation. A new environment in semantics means, normally, a new typing context in the type-checking rules. We use the extended judgment to type-check a `fun..in` expression as follows:

$$\frac{\begin{array}{c}\Delta \cdot \Gamma + [\overline{x : \pi}] \cdot \Sigma \vdash e_1 : \sigma' \qquad \sigma'_{\restriction \Gamma + [\overline{x:\pi}]} \sqsubseteq \sigma \\ \Delta + [f : \forall \overline{\alpha}.(\overline{x : \pi}) \to \sigma] \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma'' \qquad \overline{\alpha} \# \Gamma \qquad \overline{\alpha} \# \Delta\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 : \sigma''} \ (\textsc{TFun})$$

A `fun..in` expression represents a *type generalization* point in KidML. Our function types are *polymorphic* with respect to the sequence $\overline{\alpha}$. Contrarily to what is often the practice in the design of type-checking systems for ML-like languages, we do not extend our typing environments Γ to contain explicit declarations of type variables. Therefore, we add the two side conditions $\overline{\alpha} \# \Gamma$ and $\overline{\alpha} \# \Delta$, which constrains $\overline{\alpha}$ to a sequence of type variables that do not appear in the co-domain of Γ and Δ. Let us note the use of operator $\sqsubseteq$ to compare types σ' and σ. This comparison lifts $\sqsubseteq$ to operate over types, as defined in Fig. 3.4. Following such definition, for some types σ_1 and σ_2, the relation $\sigma_1 \sqsubseteq \sigma_2$ holds only if both types exhibit the same set of effects. However, this may seem very restrictive, since we can produce some *local effects* in the body of a function, which are not visible from outside the function definition. If we consider, for instance, function `transposition` from Sec. 2.2.2 (page 17), from a caller point of view, this acts as an effect-free function and so we can only assign it the type

$$(\texttt{reg}\ t,\ \emptyset,\ \emptyset)$$

However, the body of this function is an effectful computation of the following type:

$$(\texttt{reg}\ t,\ \emptyset + \texttt{t.a},\ \emptyset + \texttt{t.inv})$$

It is clear that the effects in the return type of a function can refer, only, to the function arguments or to variables in scope before the `fun..in` expression. In other words, and using the same symbols as in rule (**TFun**), the effects of σ are those we get by restricting σ' to effects over variables declared in $\Gamma + [\overline{x : \pi}]$. We take exactly such extended typing environment to compute $\sigma'_{\restriction \Gamma + [\overline{x:\pi}]}$, *i.e.*, the type obtained by intersecting the variables referred in the effect of σ' with the variables in the domain of $\Gamma + [\overline{x : \pi}]$.

In rule (**TFun**), we can assign different masks to σ and to σ'. The relation between these two masks is established via the first premise of operator $\sqsubseteq$. The mask of σ' can be *less ghost* than the mask of σ. That fact that we explicitly write the return type of a function, allows us to declare a mask with more ghost components than the mask that is inferred for expression e_1. This is convenient, for instance, when we wish to share some value in ghost and regular code. Only because we declare a component of the result of a function to be ghost, it does not forbid us to use that component in regular code, within the body of the function. In other words, the mask of the return type is not used to contaminate the body of the function. Let us illustrate using function `cycle_length` presented in Sec. 2.2.2 (page 20). We declare function `cycle_length` to return a value of type (`reg int63, ghost seq int`), whereas its body is an expression of the form

WhyML
```
let n = ref 1 in
let ghost s = ref (singleton (to_int i)) in
...
!n, !s
```

In this case, expression `!n, !s` is of type ((`reg int63, ghost seq int`), $\emptyset$, $\emptyset$), the mask matching exactly the one given in the return type. On the other hand, had we written the body of `cycle_length` as

```
let n = ref 1 in                                                    WhyML
let s = ref (singleton (to_int i)) in
...
!n, !s
```

this would still be accepted by rule (**TFun**), with the mask (`reg int63, ghost seq int`). In such a case, we could use variable `s` inside regular code, and the `let s = ...` expression would stay in the extracted code. Nonetheless, the extraction mechanism still erases the second component of the pair `!n, !s`, in order to agree with the type of `cycle_length` after extraction, which becomes `reg int63`. In Chap. 4 we show how our extraction function takes into account the mask of the return type to produce a type-checking program.

Function application. When it comes to apply a function symbol to a sequence of arguments, we write in KidML an expression of the form $f\langle\overline{\tau}\rangle(\bar{a})$. The type sequence $\overline{\tau}$ is used to instantiate the polymorphic type of f, as we shall see in the forthcoming type-checking rules, while the sequence $\bar{a}$ represents the effective arguments of f. Let us note that this expression is in A-normal form, as the effective arguments are limited to atomic expressions. The semantics of function application is as follows:

$$\frac{\delta(f) = (\overline{x : \pi}, e) \qquad \|\overline{x : \pi}\| = \|\bar{v}\| \qquad \delta \cdot \mu \cdot e[\bar{x} \mapsto \bar{v}] \Downarrow \mu' \cdot \overline{v'}}{\delta \cdot \mu \cdot f\langle\overline{\tau}\rangle(\bar{v}) \Downarrow \mu' \cdot \overline{v'}} \text{ (EvalApp)}$$

First, we fetch the arguments and body of f from δ; second, we ensure the sequence of formal arguments and effective ones are of the same length (KidML does not support partial application); finally, we evaluate expression $e[\bar{x} \mapsto \bar{v}]$ down to $\overline{v'}$, which becomes the result of the whole application expression.

Assigning a type to an expression of the form $f\langle\overline{\tau}\rangle(\bar{a})$ raises a number of interesting questions. First, assuming we have $\Delta(f) = \forall\overline{\alpha}.(\overline{x : \beta\tau'}) \to \sigma$, we need to instantiate the polymorphic type of f. When writing an application expression, we explicitly give the sequence $\overline{\tau}$, which we use to build the type substitution $[\overline{\alpha} \mapsto \overline{\tau}]$. Let us use θ to denote such a substitution. We apply θ to the return type of f to compute the type of the whole $f\langle\overline{\tau}\rangle(\bar{a})$ expression, *i.e.* the typing derivation of an application must end up with a conclusion of the form

$$\Delta \cdot \Gamma \cdot \Sigma \vdash f\langle\overline{\tau}\rangle(\bar{a}) : \sigma\theta$$

The operation $\sigma\theta$ lifts type substitution to the level of types with effects as follows:

$$(\overline{\beta\tau},\ \epsilon,\ \gamma)\theta \quad \triangleq \quad (\overline{\beta\tau\theta},\ \epsilon,\ \gamma)$$

We must also propagate substitution θ to the type derivation of the effective arguments, *i.e.*, the typing derivation of an application must feature a premise of the form

$$\forall i.\ \Delta \cdot \Gamma \cdot \Sigma \vdash a_i : (\beta_i'\tau_i'\theta,\ \emptyset,\ \emptyset)$$

A second important point, when typing an application expression, is how to treat effects. The return type of a function might feature some effects, which we must instantiate with the effective arguments. We talk here about the operation of *effects instantiation*. This is not different from type instantiation: we introduce a substitution of the form $[\bar{x} \mapsto \bar{a}]$ (formal arguments to effective arguments), that we must apply to the inferred type of an application. We use ρ to denote such a substitution and we employ it as follows:

$$\Delta \cdot \Gamma \cdot \Sigma \vdash f\langle\overline{\tau}\rangle(\bar{a}) : \sigma\theta\rho$$

This updates the conclusion of an application typing derivation, where $\sigma\theta\rho$ is defined as follows:

$$(\overline{\beta\tau},\ \epsilon,\ \gamma)\theta\rho \quad \triangleq \quad (\overline{\beta\tau\theta},\ \epsilon\rho,\ \gamma\rho)$$

We are now in position to define the type-checking rules for a function application. We introduce the following two rules, in order to type-check $f\langle\overline{\tau}\rangle(\bar{a})$:

$$\frac{\begin{array}{c}\theta = [\overline{\alpha} \mapsto \overline{\tau}] \qquad \rho = [\bar{x} \mapsto \bar{a}] \qquad \|\overline{\alpha}\| = \|\overline{\tau}\| \qquad \|\bar{x}\| = \|\bar{a}\| \\ \Delta(f) = \forall\overline{\alpha}.(\overline{x : \beta\tau'}) \to \sigma \qquad \forall i.\ \Delta\cdot\Gamma\cdot\Sigma \vdash a_i : (\beta_i'\tau_i'\theta,\ \emptyset,\ \emptyset) \qquad \forall i.\ \beta_i' \sqsubseteq \beta_i\end{array}}{\Delta\cdot\Gamma\cdot\Sigma \vdash f\langle\overline{\tau}\rangle\ (\bar{a}) : \sigma\theta\rho}\ (\textsc{TApp})$$

$$\frac{\begin{array}{c}\theta = [\overline{\alpha} \mapsto \overline{\tau}] \qquad \rho = [\bar{x} \mapsto \bar{a}] \qquad \|\overline{\alpha}\| = \|\overline{\tau}\| \qquad \|\bar{x}\| = \|\bar{a}\| \\ \Delta(f) = \forall\overline{\alpha}.(\overline{x : \beta\tau'}) \to \sigma \qquad \forall i.\Delta\cdot\Gamma\cdot\Sigma \vdash a_i : (\beta_i'\tau_i'\theta,\emptyset,\emptyset) \qquad \exists i.\ \beta_i' \not\sqsubseteq \beta_i\end{array}}{\Delta\cdot\Gamma\cdot\Sigma \vdash f\langle\overline{\tau}\rangle\ (\bar{a}) : \text{👻}(\sigma\theta\rho)}\ (\textsc{TAppGhostArg})$$

Note that we impose the elements in substitutions θ and ρ to have the same length. In particular, this ensures that function f is applied to the correct number of effective arguments. It is worth pointing out that the effects in $\sigma\theta\rho$ must respect the implicit conditions that we only assign admissible effects in our typing derivations. In other words, substitution ρ cannot introduce a set of effects violating predicate `adm`. These two rules differ only on the rightmost premise. Similar to what we did in rules **(TRecord)** and **(TRecordGhost)**, we distinguish between the case where we must ghostify or preserve the ghost status of the type of a $f\langle\overline{\tau}\rangle\ (\bar{a})$ expression. In rule **(TApp)**, every effective argument is less or equally ghost as the corresponding formal argument. In rule **(TAppGhostArg)**, at least one a_i is a ghost expression, whereas the i-th argument of f is expected to be a regular value. We ghostify the type in the conclusion, and in such cases we say the expression is *contaminated by this arguments*.

A digression on the Why3 type system. An important aspect we must note is that the type inferred via the **(TApp)** rule can, very well, be an entirely ghost type. For instance, let us consider the following, very simple, KidML program:

```
fun ff⟨⟩ (x: reg int) = ghost x in ff⟨⟩ (42)
```
KidML

We can build a typing derivation for this program using the **(TFun)** and **(TApp)** rules as follows:

$$\frac{\dfrac{\dfrac{\vdots}{\emptyset\cdot[\mathtt{x} : \mathtt{reg\ int}]\cdot\emptyset \vdash \mathtt{ghost\ x} : \text{👻}(\mathtt{reg\ int},\ \emptyset,\ \emptyset)}\ (\textsc{TGhost}) \qquad \dfrac{\begin{array}{c}\Delta(\mathtt{ff}) = (\mathtt{x} : \mathtt{reg\ int}) \to (\mathtt{ghost\ int},\ \emptyset,\ \emptyset) \\ \Delta\cdot\emptyset\cdot\emptyset \vdash 42 : (\mathtt{reg\ int},\ \emptyset,\ \emptyset) \qquad \ldots\end{array}}{\Delta\cdot\emptyset\cdot\emptyset \vdash \mathtt{ff}\langle\rangle\ (42) : (\mathtt{ghost\ int},\ \emptyset,\ \emptyset)}\ (\textsc{TApp})}{\emptyset\cdot\emptyset\cdot\emptyset \vdash \mathtt{fun\ ff}\langle\rangle\ (\mathtt{x} : \mathtt{reg\ int}) = \mathtt{ghost\ x\ in\ ff}\langle\rangle\ (42) : (\mathtt{ghost\ int},\ \emptyset,\ \emptyset)}\ (\textsc{TFun})$$

In the above derivation, Δ stands for the environment $[\mathtt{ff} : (\mathtt{x} : \mathtt{reg\ int}) \to (\mathtt{ghost\ int},\ \emptyset,\ \emptyset)]$. We omit the derivation from rule **(TGhost)**, as well as the premise comparing masks of the formal and effective argument of `ff` in **(TApp)**. Whenever an application becomes a ghost expression because of the ghost status of the function return type, we say the expression is *contaminated by application*. If we take a look to the Why3 type system, we find a very similar contamination mechanism for these situations. In fact, in the definition of `cycle_length` (Sec. 2.2.2, page 20) we can already find an example of contamination by application. In the body of this function, we make call to the `pigeonhole` lemma function in the following expression:

```
if to_int !n + 1 > t.size then pigeonhole (Seq.length !s) t.size (get !s)
```
WhyML

In WhyML, we can directly declare a function as a ghost symbol, independently of the ghost status of its body, either by writing

```
let ghost foo (x: int) = ...
```
WhyML

or by introducing a lemma function of the form

```
let lemma foo (x: int) = ...
```
WhyML

The `pigeonhole` function falls in the latter category. The application

```
pigeonhole (Seq.length !s) t.size (get !s)
```
WhyML

is, thus, contaminated by the ghost status of `pigeonhole`, and so is the whole `if..then` expression. Getting back to the KidML type system, if we use our type-checking rules to assign a type to the `pigeonhole` application, we end up with the derivation

$$\frac{\begin{array}{c}\vdots\end{array}}{\Delta \cdot \Gamma \cdot \emptyset \vdash \texttt{pigeonhole}\ (\texttt{Seq.length}\ \texttt{!s})\ \texttt{t.size}\ (\texttt{get}\ \texttt{!s}) : (\texttt{Unit},\ \emptyset,\ \emptyset)}\ (\textsc{TApp})$$

where Δ binds, at least, a type to functions `pigeonhole`, `Seq.length`, `get`, and `(!)`, and the typing context Γ stores a binding, for, at least, variable s. We consider the `Unit` type to refer to an empty sequence of results, with no particular ghost attached. The call to `pigeonhole`, and consequently the whole `if..then..else` expression, does not return any result neither performs side-effects. Such expression can, thus, be erased at extraction time.

Recursive definitions. We extend the KidML language with local definition of recursive functions. The syntax $\texttt{rec}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e\ \texttt{in}\ e$ is very similar to the `fun..in` construction, except for the use of the `rec` keyword to indicate the recursive nature of the definition. The semantics of a recursive function is given by the following rule:

$$\frac{\delta[f \mapsto (\overline{x : \pi}, e_1)] \cdot \mu \cdot e_2 \Downarrow \mu' \cdot \bar{\nu}}{\delta \cdot \mu \cdot \texttt{rec}\ f(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 \Downarrow \mu' \cdot \bar{\nu}}\ (\textsc{EvalRec})$$

This is the exact same semantics as that of a `fun..in` expression.

Our evaluation relation asserts that $\delta[f \mapsto (\overline{x : \pi}, e_1)] \cdot \mu \cdot e_2 \Downarrow \mu' \cdot \bar{\nu}$ holds if e_2 evaluates down to $\bar{\nu}$ in a *finite* number of steps. Such an hypotheses makes it meaningless to use our semantics to deal with *divergent computations*. To account for possible divergent evaluations, we extend the possible outcome of an evaluation: either it terminates on a sequence of values $\bar{\nu}$ or it diverges. This leads us to the notion of *semantic results*, which we define as follows:

$$r \ ::= \ \bar{\nu} \mid \texttt{div}$$

Divergence is explicitly represented using the newly introduced constant `div`. We update our evaluation predicate accordingly:

$$\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r$$

A divergent KidML expression can be seen as an expression inducing an *infinite evaluation* derivation. In order to reason about such *infinite* derivation trees, we introduce a *coevaluation* judgment of the form

$$\delta \cdot \mu \cdot e \Downarrow^{\text{co}} \mu' \cdot r$$

which defines a coinductive interpretation of the evaluation rules, where $\delta \cdot \mu \cdot e \Downarrow^{\text{co}} \mu' \cdot r$ is the conclusion of a finite or infinite derivation tree. We write $\delta \cdot \mu \cdot e \Downarrow^{\text{co}} \texttt{div}$ to state that e diverges

during execution. For each evaluation rule, we introduce a coevaluation rule of the same form. For instance, the following is the updated (**EvalRec**) rule together with its coinductive counterpart:

$$\frac{\delta[f \mapsto (\overline{x : \pi}, e_1)] \cdot \mu \cdot e_2 \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{rec}\ f(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 \Downarrow \mu' \cdot r}\ (\textsc{EvalRec})$$

$$\frac{\delta[f \mapsto (\overline{x : \pi}, e_1)] \cdot \mu \cdot e_2 \Downarrow^{co} \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{rec}\ f(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 \Downarrow^{co} \mu' \cdot r}\ (\textsc{CoEvalRec})$$

We represent coevaluation rules using a double bar.

Before presenting the type-checking rule for recursive functions, let us take a step back and look at the evaluation of a `let..in` expression. First, we (straightforwardly) update the (**EvalLet**) rule to cope with the changes on the evaluation predicate:

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot \bar{v} \qquad \delta \cdot \mu' \cdot e_2[\bar{x} \mapsto \bar{v}] \Downarrow \mu'' \cdot r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow \mu'' \cdot r}\ (\textsc{EvalLet})$$

Following the above rule, we observe that the semantic result r in the conclusion is taken from the evaluation of $e_2[\bar{x} \mapsto \bar{v}]$. This only makes sense when the first premise holds, *i.e.*, when there is a finite evaluation for expression e_1 down to $\bar{v}$. If the execution of e_1 diverges, we must *propagate* divergence as the result of evaluating the whole `let..in` expression. The following premise asserts the divergence of e_1:

$$\delta \cdot \mu \cdot e_1 \Downarrow^{co} \mu' \cdot \texttt{div}$$

Our treatment of divergence as a constant and the use of a coinductive judgment builds on the works of Arthur Charguéraud [30], and Xavier Leroy and Hervé Grall [98, 100]. Anticipating the extensions from next section, let us give a generalized presentation for this premise. We introduce a predicate `abort` to describe semantic results that break the normal execution flow of a program. Since constant `div` is, for now, the only mean to interfere in the execution of an expression, predicate `abort` is simply defined as follows:

$$\frac{}{\texttt{abort div}}\ (\textsc{AbortDiv})$$

Using this definition, the inductive and co-inductive semantic rules for a divergent `let..in` expression are as follows:

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot \bar{v} \qquad \delta \cdot \mu' \cdot e_2[\bar{x} \mapsto \bar{v}] \Downarrow^{co} \mu'' \cdot r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow^{co} \mu' \cdot r}\ (\textsc{CoEvalLet})$$

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow^{co} \mu' \cdot r \qquad \texttt{abort}\ r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow^{co} \mu' \cdot r}\ (\textsc{CoEvalLetAbort})$$

It is interesting to note that our definitions allow us to write judgments of the form $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot \texttt{div}$ or $\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot \bar{v}$, even if we are only interested in $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot \bar{v}$ and $\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot \texttt{div}$. In next section, we prove a number of properties about the behavior of our inductive and co-inductive judgments, which rule out some unnatural behaviors.

We finally describe the type assignment rule for recursive definitions. First, we must clarify the role of divergence in our type and effects context. In KidML, we follow the same direction as in WhyML and treat divergence as an effect. We extend our definition of effects accordingly:

$$\epsilon, \gamma \triangleq (\varphi, \texttt{div})$$

The meta-variable φ stands for the set of writing effects and `div` is used as a Boolean mark to indicate whether the expression may diverge. Using the extended definition of effects, we devise the following type-checking rule for recursive functions:

$$\frac{\sigma = (\overline{\beta\tau},\ \epsilon + \texttt{div},\ \gamma) \qquad \begin{array}{c}\Delta + [\mathsf{f} : \forall\overline{\alpha}.(\overline{x : \pi}) \to \sigma] \cdot \Gamma + [\overline{x : \pi}] \cdot \Sigma \vdash e_1 : \sigma' \qquad \sigma'_{\restriction \Gamma + [\overline{x:\pi}]} \sqsubseteq \sigma \\ \Delta + [\mathsf{f} : \forall\overline{\alpha}.(\overline{x : \pi}) \to \sigma] \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma'' \qquad \overline{\alpha} \# \Gamma \qquad \overline{\alpha} \# \Delta\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{rec}\ \mathsf{f}\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 : \sigma''}\ (\textsc{TRecDiv})$$

Let us note that, by default, we assign every recursive function a divergent status (leftmost premise). In Sec. 3.1.5, we extend our type-checking system with proof-related elements, namely oracle functions that state the termination or divergence of evaluation. In the case of provable terminating definitions, we shall be able to remove divergence from the function effects. Besides divergence, the other difference between rule **(TRecDiv)** and **(TFun)** is the extension of Δ with the type of function f when type-checking e_1.

To conclude this section, we detail on how to treat divergence as an admissible effect. This is, actually, very straightforward: ghost expressions cannot diverge and so we must forbid divergence to appear in the ghost effects assigned to an expression. We update our definition of predicate `adm` as follows:

$$\begin{array}{rcl}\texttt{adm}(\overline{\beta\tau},\ \epsilon,\ \gamma) & \triangleq & \neg\texttt{obs}\ \texttt{writes}(\gamma) \wedge \\ & & \neg\texttt{div}(\gamma)\end{array}$$

It is worth pointing out that, for now, we can only type-check non-ghost recursive functions. We automatically add the divergence effect in the **(TRecDiv)** rule, making it impossible to build a typing derivation for recursive ghost functions, as this would violate the global condition that every assigned effect is a valid effect. As aforementioned, in Sec. 3.1.5 we shall introduce means to decide upon the termination of a recursive definition, which we will use to relax the constraints of rule **(TRecDiv)**.

3.1.4 Exceptions

In this section, we continue extending KidML with new language features, namely *exception* declaration, raising and handling. We add to our language a top-level construction to declare exceptions, as follows:

$$\texttt{exception}\ \mathsf{E} : \overline{\pi}$$

The keyword `exception` is followed by E, the name of a new exception. The $\overline{\pi}$ sequence represents the arguments of the exception, *i.e.*, a sequence of types together with a ghost status. For instance, the declaration

$$\texttt{exception Return (reg Int, ghost Bool)}$$

introduces the exception `Return` whose first argument is a regular integer and the second one is a ghost Boolean value.

We change our grammar of expressions as follows:

$$e \ ::= \ \ldots \mid \texttt{raise}\ \mathsf{E}\bar{a} \mid \texttt{try}\ e\ \texttt{with}\ \overline{\mathsf{E}\bar{x} \Rightarrow e}$$

An expression of the form $\texttt{raise}\ \mathsf{E}\bar{a}$ represents the classical exception raising constructor. From a semantic point of view, a `raise` expression cannot be further evaluated, which we represent by adding the following rule to our evaluation judgment definition:

$$\frac{}{\delta \cdot \mu \cdot \texttt{raise}\ \mathsf{E}\bar{v} \Downarrow \mu \cdot \texttt{raise}\ \mathsf{E}\bar{v}}\ (\textsc{EvalRaise})$$

We change our definition of semantic results accordingly:

$$r \ ::= \ \ldots \mid \texttt{raise}\ E\bar{v}$$

Exceptions are control-flow changing constructions, which make them amenable to be included in our predicate `abort`:

$$\frac{}{\texttt{abort div}}\ (\textsc{AbortDiv}) \qquad \frac{}{\texttt{abort raise}\ E}\ (\textsc{AbortExn})$$

We treat exceptions in KidML as an effect, following the same approach as in the WhyML language. In the context of our type and effects system, we update the effects assigned to an expression to a triple of the form

$$\epsilon,\ \gamma \ \triangleq \ (\varphi,\ \texttt{div},\ \mathcal{X})$$

where $\mathcal{X}$ is the set of raised exceptions. Using this extended definition of effects, we give the following type-checking rules for a `raise` expression:

$$\frac{\texttt{exception}\ E : \beta_1\tau_1 \ldots \beta_n\tau_n \qquad \forall i.\ \Delta\cdot\Gamma\cdot\Sigma \vdash a_i : (\beta_i'\tau_i, \emptyset, \emptyset) \qquad \forall i.\ \beta_i' \sqsubseteq \beta_i}{\Delta\cdot\Gamma\cdot\Sigma \vdash \texttt{raise}\ E\bar{a} : (\overline{\texttt{reg}\,\tau'},\ \emptyset + \texttt{raises}\ E,\ \emptyset)}\ (\textsc{TRaiseReg})$$

$$\frac{\texttt{exception}\ E : \beta_1\tau_1 \ldots \beta_n\tau_n \qquad \forall i.\ \Delta\cdot\Gamma\cdot\Sigma \vdash a_i : (\beta_i'\tau_i, \emptyset, \emptyset) \qquad \exists i.\ \beta_i' \not\sqsubseteq \beta_i}{\Delta\cdot\Gamma\cdot\Sigma \vdash \texttt{raise}\ E\bar{a} : (\overline{\texttt{ghost}\,\tau'},\ \emptyset,\ \emptyset + \texttt{raises}\ E)}\ (\textsc{TRaiseGhost})$$

We distinguish between regular and ghost exceptions using the ghost status of arguments in exception declaration and the status of each component in sequence $\bar{a}$. This is a similar approach to what we did in rules for record creation and function application. The leftmost premise of **(TRaiseReg)** and **(TRaiseGhost)** rules stipulates that an exception named E was declared with arguments $\beta_1\tau_1 \ldots \beta_n\tau_n$.

The second syntactic construction we add to KidML is the exception handler. The semantics of a $\texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e}$ expression is the following. First, we evaluate expression e_0. If it evaluates down to a semantic result r that is not a `raise` expression, either a sequence of values or divergence, the whole `try..with` expression evaluates down to r. On the other hand, if e_0 evaluates down to $\texttt{raise}\ E'\bar{v}$, two different outcomes are possible: E' is exactly exception E_i from sequence $\overline{E\bar{x} \Rightarrow e}$, in which case the result of evaluating expression e_i is the result of the whole `try..with` expression; or, if E' does not match any exception in $\overline{E\bar{x}}$, we *propagate* such an exception as the result of the whole evaluation. The following big-step rules systematize this informal explanation:

$$\frac{\delta\cdot\mu\cdot e_0 \Downarrow \mu'\cdot\bar{v}}{\delta\cdot\mu\cdot\texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow \mu'\cdot\bar{v}}\ (\textsc{EvalTry})$$

$$\frac{\delta\cdot\mu\cdot e \Downarrow \mu'\cdot r \qquad \texttt{abort}\ r \qquad \forall i.\ E_i \neq r}{\delta\cdot\mu\cdot\texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow \mu'\cdot r}\ (\textsc{EvalTryAbort})$$

$$\frac{\begin{array}{c}\delta\cdot\mu\cdot e_0 \Downarrow \mu''\cdot\texttt{raise}\ E'\bar{v} \\ E' = E_i \qquad \forall j.\ j < i \rightarrow E' \neq E_j \qquad \delta\cdot\mu''\cdot e_i[\overline{x_i} \mapsto \bar{v}] \Downarrow \mu'\cdot r\end{array}}{\delta\cdot\mu\cdot\texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow \mu'\cdot r}\ (\textsc{EvalTryExn})$$

Rule **(EvalTryAbort)** describes, at the same time, the case where we propagate divergence out of the evaluation of e_0, or an exception that is not listed by the handler.

We now focus on type-checking `try..with` expressions. The following are the three premises used to assign types to the sub-expressions of an exception handler:

$$\Delta \cdot \Gamma \cdot \Sigma \vdash e_0 : (\overline{\pi},\ \epsilon,\ \gamma) \qquad \forall i.\ \texttt{exception}\ E_i : \overline{\pi_i} \qquad \Delta \cdot \Gamma + [\bar{x} : \overline{\pi_i}] \cdot \Sigma \vdash e_i : \sigma_i$$

We detail on each of these three conditions. The first is rather straightforward and is used to assign the body e_0 with some type $(\overline{\pi},\ \epsilon,\ \gamma)$. Next, we verify that every exception E_i listed in the handler is defined with arguments $\overline{\pi_i}$. Finally, we type every expression e_i with the type σ_i, under a typing context extended with the bindings $\bar{x} : \overline{\pi}$. Using these three premises, we devise the following type-checking rule:

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_0 : (\overline{\pi},\ \epsilon,\ \gamma) \qquad \forall i.\ \texttt{exception}\ E_i : \overline{\pi_i} \qquad \Delta \cdot \Gamma + [\bar{x} : \overline{\pi_i}] \cdot \Sigma \vdash e_i : \sigma_i}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} : (\overline{\pi},\ \epsilon - \overline{\texttt{raises}\ E},\ \gamma - \overline{\texttt{raises}\ E}) \cup \bigcup_i \sigma_i}\ (\textsc{TTry})$$

An exception raised in e_0 and caught by the handler do not interfere with the execution of the *whole* `try..with` expression, hence we remove it from the effect assigned in the conclusion. Notation $\epsilon - \overline{\texttt{raises}\ E_i}$ (resp. γ) stands for the regular (resp. ghost) effect ϵ (resp. γ) from which we remove the sequence $\overline{E}$ of exceptions listed in the handler. Nonetheless, the final effect of a `try..with` expression *does* depend on the effects of each individual expression e_i. We add the effect of every type σ_i via the union operation on the right-hand side of the conclusion. Such an operation also changes the expression mask: we take the typing information $\overline{\pi}$ from the body of the handler e_0 and combine its mask with the mask of every type σ_i. This means, that, even if e_0 is a regular expression, the whole `try..with` expression gets contaminated if there is a ghost expression e_i.

Before giving an example of a typing derivation using rules **(TRaise)** and **(TTry)**, we extend our notion of admissible effects to accommodate exceptions. Contrarily to divergence, both regular and ghost expressions can raise exceptions. Nonetheless, a ghost exception propagates a ghost effect up until the first handler that catches it. In other words, raising an exception within ghost code is a source of ghost contamination, and so it can only be done if the enclosing expression is free of regular effects. We add such a condition to our predicate `adm`, as follows:

$$\begin{aligned}\texttt{adm}(\overline{\beta\tau},\ \epsilon,\ \gamma) \triangleq\ & \neg\texttt{reg}\,\texttt{writes}(\gamma) \wedge \\ & \neg\texttt{div}(\gamma) \wedge \\ & \texttt{raises}(\gamma) \Rightarrow \epsilon = \emptyset \wedge \overline{\beta} = \overline{\texttt{ghost}}\end{aligned}$$

The fact that a ghost exception implies a completely ghost mask $\overline{\beta}$ is also deeply connected to how the ghostification operation $\text{👻}(\cdot)$ treats exceptions. We relax the condition of a completely empty set of regular effects (page 37) and give the following updated definition for $\text{👻}(\cdot)$:

$$\text{👻}(\overline{\beta\tau},\ (\emptyset, \bot, \mathcal{X}),\ \gamma) \triangleq (\overline{\texttt{ghost}\,\tau},\ \emptyset,\ \gamma \cup \mathcal{X})$$

We allow the ghostification of terminating expressions (we use $\bot$ to represent the absence of divergence), containing no regular writing effects, but possibly raising regular exceptions. Such regular exceptions then become ghost exceptions, as noted by $\gamma \cup \mathcal{X}$.

In order to illustrate the type-checking of raising and handling exceptions, let us consider the following KidML program:

KidML

```
exception A
exception B

fun foo⟨⟩ (x: reg int) : reg int =
  try if x > 42 then raise A else 89
  with A -> ghost raise B in
foo (73)
```

We begin by declaring the two zero-argument exceptions `A` and `B`. Next, the main part of this program is function `foo`. It takes a regular integer argument and returns a regular integer. The body of this function is a `try..with` expression, catching only exception `A`. Let us focus, first, on the body of the handler. This is an `if..then..else` expression, where the `then` branch simply raises exception `A` and the `else` branch returns 89 (as a matter of convenience, we do not respect here the A-normal form convention, writing the test of the `if..then..else` as a compound expression). We can build the following typing derivation for the body of the handler:

$$\dfrac{\dfrac{\ldots}{\ldots}\,(\textsc{TRaise})\quad\dfrac{\vdots}{\Delta\cdot\Gamma\cdot\Sigma\vdash \texttt{raise A} : (\texttt{reg int}, \emptyset + \texttt{raises A}, \emptyset)}\qquad\dfrac{\vdots}{\Delta\cdot\Gamma\cdot\Sigma\vdash \texttt{89} : (\texttt{reg int}, \emptyset, \emptyset)}\,(\textsc{TConst})}{\Delta\cdot\Gamma\cdot\Sigma\vdash \texttt{if x} > \texttt{42 then raise A else 89} : (\texttt{reg int}, \emptyset + \texttt{raises A}, \emptyset)\cup(\texttt{reg int}, \emptyset, \emptyset)}\,(\textsc{TIf})$$

We omit the typing trees for the test expression, as well as for the premises in rules (**TRaise**) and (**TConst**) as these are straightforward. The (**TRaise**) rule allows to type the `raise A` expression with type `reg int`, which allows then to match the same type as for the `else` branch and complete the derivation with rule (**TIf**). The type assigned to this `if..then..else` expression, after type union, is

$$(\texttt{reg int}, \emptyset + \texttt{raises A}, \emptyset)$$

We focus now on the typing derivation for the expression after the `with A` clause. We use the (**TGhost**) rule as follows:

$$\dfrac{\Delta\cdot\Gamma\cdot\Sigma\vdash \texttt{raise B} : (\texttt{reg int}, \emptyset + \texttt{raises B}, \emptyset)}{\Delta\cdot\Gamma\cdot\Sigma\vdash \texttt{ghost raise B} : \texttt{👻}(\texttt{reg int}, \emptyset + \texttt{raises B}, \emptyset)}\,(\textsc{TGhost})$$

The only regular effect is the rise of exception `B`, hence it is legal to apply the ghostification operation, which results in the type

$$(\texttt{ghost int}, \emptyset, \emptyset + \texttt{raises B})$$

This type respects the conditions of the `adm` predicate, making the above derivation fit our global condition of admissible effects. Finally, we combine the two given typing derivations to type-check the whole `try..with` expression, as follows:

$$\dfrac{\begin{array}{c}\Delta\cdot\Gamma\cdot\Sigma\vdash \texttt{if..then..else} : (\texttt{reg int}, \emptyset + \texttt{raises A}, \emptyset)\\ \texttt{exception A}\qquad \Delta\cdot\Gamma\cdot\Sigma\vdash \texttt{ghost raise B} : (\texttt{ghost int}, \emptyset, \emptyset + \texttt{raises B})\end{array}}{\Delta\cdot\Gamma\cdot\Sigma\vdash \texttt{try..with..} : (\texttt{reg int}, \emptyset + \texttt{raises A} - \texttt{raises A}, \emptyset)\cup(\texttt{ghost int}, \emptyset, \emptyset + \texttt{raises B})}\,(\textsc{TTry})$$

Since we remove exception `A` from the regular effects of the type at the left-hand side of the union operation, the final type computed is

$$(\texttt{ghost int}, \emptyset, \emptyset + \texttt{raises B})$$

which fits in our definition of admissible effects. Using this typing derivation, we observe that, indeed, the ghost status and ghost effect of the `ghost raise B` expression is propagated to the whole `try..with` expression, which turns the `foo` function into a ghost function. The application `foo (73)` is, consequently, a ghost expression which we could easily verify using the (**TFun**) rule.

Loop structure. We conclude this section by adding to KidML a very simple loop construction, as follows:

$$e \;::=\; \ldots \mid \texttt{loop}\; e$$

An expression of the form `loop` e stands for the infinite evaluation of expression e, as formalized by the following evaluation rule:

$$\frac{\delta \cdot \mu \cdot \texttt{let reg \_} = e \;\texttt{in loop}\; e \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{loop}\; e \Downarrow \mu' \cdot r}\ (\textsc{EvalLoop})$$

We introduce the `loop` construction after having introduced exceptions, simply because exceptions offer a mean to escape the body of an infinite loop. In fact, from our evaluation relation point of view, we are only interested in judgments of the form $\delta \cdot \mu \cdot \texttt{loop}\; e \Downarrow \mu' \cdot \texttt{raise}\; E\bar{v}$, for some exception E. The type-checking rule for a `loop` e expression is straightforwardly defined as follows:

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e : (\texttt{Unit},\ \epsilon,\ \gamma)}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{loop}\; e : (\overline{\texttt{reg}\;\tau},\ \epsilon + \texttt{div},\ \gamma)}\ (\textsc{TLoopDiv})$$

The typing information assigned in the conclusion is actually a design choice for the KidML language. Even if the body e must be typed as a `Unit` expression, the whole `loop` is assigned a generic type $\overline{\texttt{reg}\;\tau}$. As for the rule **(TRaise)**, this means we can assign any type to a `loop` e expression, depending on the context in which we type-check such an expression. Perhaps, the most obvious way would be to type-check `loop` e also as a `Unit` expression. Let us use an example written in KidML to show how the second solution would create a practical pitfall. The following KidML program computes the integer square root of a non-negative integer, using the Newton-Raphson method:

KidML

```
exception Sqrt : reg int

fun isqrt⟨⟩ (x: reg int) : (reg int, ∅, ∅) =
  if x = 0 then 0 else
  if x <= 3 then 1 else
  let reg y = ref x in
  let reg z = ref ((1 + x) / 2) in
  try
    loop if !z >= !y then raise Sqrt !y;
      y := !z;
      z := (x / !z + !z) / 2
  with Sqrt y -> y in
isqrt 17
```

Function `isqrt` computes a non-negative result such that $\texttt{isqrt}(x) = \lfloor\sqrt{x}\rfloor$. We use the `loop` structure to iterate until the stopping condition `!z >= !y` is reached. In that case, we raise the `Sqrt` exception applied to the value of reference `y`, which stores the integer square root at the end of the computation. For the example above, the expression `isqrt 17` evaluates down to 4, as expected. Using the **(TLoopDiv)** rule, we can type-check the body of the `try..with` expression with type

$$(\texttt{reg int},\ \emptyset + \texttt{raises Sqrt},\ \emptyset)$$

and so the whole `try..with` expression is assigned the type

$$(\texttt{reg int},\ \emptyset,\ \emptyset)$$

which exactly matches the type of the `isqrt` function. Had we chosen to assign the `Unit` type in the conclusion of the (**TLoopDiv**) rule and the given implementation of the `isqrt` function would be rejected. In that case, we would have to write a dummy integer value after the `loop` expression, for instance

KidML
```
fun isqrt⟨⟩ (x: reg int) : (reg int, ∅, ∅) =
  ...
  try loop ...; 1729
  with Sqrt y -> y in
...
```

which is rather unpleasant. Moreover, we know that `1729` is an *unreachable point* of execution, which makes it even more regrettable to include in the code. We could replace the constant `1729` by some primitive marking an unreachable point by execution. In OCaml we would write `assert false` after the loop structure:

OCaml
```
let isqrt (x: int) : int =
  ...
  while true do ... done; assert false
  with Sqrt y -> y in
...
```

In WhyML, we would add to the body of the `isqrt` function an `absurd` expression, as follows:

WhyML
```
let isqrt (x: int) : int =
  ...
  while true do ... done; absurd
  with Sqrt y -> y end in
...
```

We extend the KidML language with such an `absurd` primitive in the next section. Nonetheless, we keep our design choice for the (**TLoopDiv**) rule, since our language *only* features infinitely executing loops. This is not the case with the more general `while..do` loops from OCaml and WhyML, even if a simple static analysis could easily detect that a `while true do` expression stands for an infinite loop.

3.1.5 Proof-related Elements

We finally come to an end in the development of the KidML language, its operational semantics, and type system. In this section, we extend KidML with some proof-related elements, namely a syntactic primitive to mark unreachable points in the code and means to prove termination.

Unreachable points. In order to mark unreachable points in the code, we follow the same approach as WhyML and extend KidML with the `absurd` keyword, as follows:

$$e \;::=\; \ldots \mid \texttt{absurd}$$

From a semantics point of view, an `absurd` represents a completely evaluated expression, and so we add the following axiom to our (co-)evaluation judgment:

$$\frac{}{\delta \cdot \mu \cdot \texttt{absurd} \Downarrow \mu \cdot \texttt{absurd}}\ (\textsc{EvalAbsurd})$$

We also extend the set of possible semantic results to include `absurd`, as follows:

$$r \;::=\; \ldots \mid \texttt{absurd}$$

We complete, as well, our definition of predicate `abort`:

$$\frac{}{\texttt{abort absurd}}$$

The `absurd` result is not to be confused with an execution error. An expression that evaluates down to `absurd` is not a stuck expression. Indeed, this is a well-typed expression, which we type-check via the following rule:

$$\frac{}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{absurd} : (\overline{\texttt{reg}\,\tau},\ \emptyset,\ \emptyset)}\ (\textsc{TAbsurd})$$

We note that, similarly to what happens with `raise` expressions, the **(TAbsurd)** rule assigns the contextual type $\overline{\texttt{reg}\,\tau}$ in the conclusion. However, we choose to treat an `absurd` expression differently from an exception, hence the new semantic result.

The purpose of the `absurd` primitive is to be used together with a proof system, in order to formally demonstrate that some point in the program is never reached. We implicitly assume that every KidML program are formally verified, in particular all of the `absurd` are proved to be placed in unreachable execution points. In other words, a KidML program shall never evaluate down to an `absurd` result. This might be a little puzzling as it seems that, after all, there is no good reason to include the `absurd` keyword in the KidML syntax. In fact, the `absurd` expression triggers some particular type-checking situations when used in an `if..then..else` expression. These are described using the following rules:

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash a : (\beta\texttt{Bool},\ \emptyset,\ \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e : \sigma}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{if}\ a\ \texttt{then absurd else}\ e : \sigma}\ (\textsc{TIfAbsurd1})$$

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash a : (\beta\texttt{Bool},\ \emptyset,\ \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e : \sigma}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{if}\ a\ \texttt{then}\ e\ \texttt{else absurd} : \sigma}\ (\textsc{TIfAbsurd2})$$

For an expression of the form $\texttt{if}\,a\,\texttt{then}\,e_1\,\texttt{else}\,e_2$, where either e_1 or e_2 is an `absurd`, we can forget about the ghost status of the test expression a when type-checking the whole `if..then..else` expression. For instance, the expression

```
if ghost true then absurd else 42
```
KidML

is assigned the regular type $(\texttt{reg int},\ \emptyset,\ \emptyset)$, despite the ghost status of the test expression.

The particular role of `absurd` when type-checking `if..then..else` expressions is better understood by taking an operational perspective, *i.e.*, if we think of such expressions in terms of code extraction. As we assume that every single `absurd` is proved to be unreachable, for expressions of the form

```
if a then absurd else e
```
KidML

or

```
if a then e else absurd
```
KidML

we can *bypass* the conditional structure and drop the branch corresponding to the absurd case. The type of the whole `if..then..else` must follow the type assigned to the e expression. In next chapter, we present our extraction function, for which we devise a particular extraction scheme to deal with the combination of `if..then..else` and `absurd` expressions.

Termination. When it comes to recursion and iteration, we have only considered the possibility of divergent evaluations. To overcome such a limitation, we extend the KidML type system with

a proof oracle to state the termination of a given expression. We use the `CheckTermination`$(\cdot)$ function to implement such an oracle, where `CheckTermination`(e) states the evaluation of expression e always terminates. There are a few different approaches to implement the `CheckTermination` function: in WhyML, we annotate functions with termination measures, commonly called *variants*, from which the Why3 system generates sufficient verification conditions to prove the termination of the annotated expression; rich type theories encode directly in their types algebra conditions that check the termination of recursive definitions [12]. In the KidML type-with-effects system we abstract away from a particular materialization of the `CheckTermination` oracle. We take here exactly the same approach as in the work of Jean-Christophe Filliâtre, Léon Gondelman and Andrei Paskevich [64]. We only care to know that some recursive or iterative definitions are provably terminating. We can benefit from such information to avoid assigning systematically a divergence effect in the conclusion of our type-checking rules. For instance, we can separate the type-checking of recursive functions in two different rules, depending on the result of the `CheckTermination` function, as follows:

$$\frac{\begin{array}{c}\Delta + [\mathsf{f} : \forall\overline{\alpha}.(\overline{x : \pi}) \to \sigma] \cdot \Gamma + [\overline{x : \pi}] \cdot \Sigma \vdash e_1 : \sigma' \qquad \sigma' \sqsubseteq \sigma \\ \neg\texttt{CheckTermination}(\texttt{rec}\ \mathsf{f}\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1) \\ \sigma = (\overline{\beta\tau}, \epsilon + \texttt{div}, \gamma) \qquad \Delta + [\mathsf{f} : \forall\overline{\alpha}.(\overline{x : \pi}) \to \sigma] \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma'' \qquad \overline{\alpha} \# \Gamma \qquad \overline{\alpha} \# \Delta\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{rec}\ \mathsf{f}\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 : \sigma''}\ (\textsc{TRecDiv})$$

$$\frac{\begin{array}{c}\Delta + [\mathsf{f} : \forall\overline{\alpha}.(\overline{x : \pi}) \to \sigma] \cdot \Gamma + [\overline{x : \pi}] \cdot \Sigma \vdash e_1 : \sigma' \qquad \sigma' \sqsubseteq \sigma \\ \texttt{CheckTermination}(\texttt{rec}\ \mathsf{f}\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1) \\ \sigma = (\overline{\beta\tau}, \epsilon, \gamma) \qquad \Delta + [\mathsf{f} : \forall\overline{\alpha}.(\overline{x : \pi}) \to \sigma] \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma'' \qquad \overline{\alpha} \# \Gamma \qquad \overline{\alpha} \# \Delta\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{rec}\ \mathsf{f}\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 : \sigma''}\ (\textsc{TRec})$$

The **(TRecDiv)** rule is a simple update from the previously shown **(TRecDiv)** rule, where we add the

$$\neg\texttt{CheckTermination}(\texttt{rec}\ \mathsf{f}\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1)$$

premise, *i.e.*, we cannot prove the termination of every call to the recursive function `f`. On the other hand, rule **(TRec)** features the premise

$$\texttt{CheckTermination}(\texttt{rec}\ \mathsf{f}\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1)$$

to indicate that every call to function `f` always terminates, which avoids us from assigning the `div` effect in the return type σ, leftmost premise. From a programmatic point of view, if we can type-check a recursive function without the divergence effect, then we can use such function in a ghost context. For instance, the following KidML program

```
rec fact⟨⟩ (n: reg int) : (reg int, ∅, ∅) =                    KidML
  if n <= 0 then 1
  else n * fact (n - 1) in
ghost fact 42
```

is now accepted by our type-system: one can prove that function `fact` always terminates (*e.g.*, using the value of argument `n` as a termination measure), and so we use the **(TRec)** to type-check the definition of `fact` with an empty effect; rule **(TGhost)** assigns expression `ghost fact (42)` the type

$$👻(\texttt{reg int}, \emptyset, \emptyset)$$

which respects our predicate `adm`. Without the `CheckTermination` oracle, we would have to include the divergence effect in the return type of `fact`, ending up with the type

$$👻(\texttt{reg int}, \emptyset + \texttt{div}, \emptyset)$$

for the `ghost fact (42)` expression, which clearly violates the definition of admissible effects.

Oracle `CheckTermination` can also be used to type-check expressions featuring the `loop` construction. This is no different from what we did with recursive definitions, as shown by the following rules:

$$\frac{\texttt{CheckTermination}(\texttt{loop}\ e) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e : (\texttt{Unit},\ \epsilon,\ \gamma)}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{loop}\ e : (\overline{\texttt{reg}\ \tau},\ \epsilon,\ \gamma)}\ (\textsc{TLoop}) \qquad \frac{\neg\texttt{CheckTermination}(\texttt{loop}\ e) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e : (\texttt{Unit},\ \epsilon,\ \gamma)}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{loop}\ e : (\overline{\texttt{reg}\ \tau},\ \epsilon + \texttt{div},\ \gamma)}\ (\textsc{TLoopDiv})$$

Following the result of `CheckTermination`, we either add or not the `div` effect in the conclusion of the rule. The latter corresponds to the case of rule **(TLoop)**, the former to **(TLoopDiv)**.

3.2 KidML Formalization

e	::=	Expressions
	$\mid \bar{a}$	
	$\mid \texttt{let}\ \overline{\beta x} = e\ \texttt{in}\ e$	
	$\mid \texttt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e\ \texttt{in}\ e$	
	$\mid \texttt{rec}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e\ \texttt{in}\ e$	
	$\mid f\langle\overline{\tau}\rangle(\bar{a})$	
	$\mid \texttt{if}\ a\ \texttt{then}\ e\ \texttt{else}\ e$	
	$\mid \{\overline{f = a}\}$	
	$\mid a.f$	
	$\mid a.f \leftarrow a$	
	$\mid \texttt{loop}\ e$	
	$\mid \texttt{raise}\ E\bar{a}$	
	$\mid \texttt{try}\ e\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e}$	
	$\mid \texttt{ghost}\ e$	
	$\mid \texttt{absurd}$	
a	::=	Atomic Expressions
	$\mid x$	
	$\mid v$	
v	::=	Values
	$\mid c$	
	$\mid l$	

Figure 3.5: KidML Syntax.

The purpose of this section is prove that one can safely write programs in the KidML language. We want to prove that the static guarantees issued from our type system are sound, with respect to the introduced operational semantics relation. This means to prove that if a KidML expression is considered a well-typed expression, then it is safe to evaluate this expression or, in other words, no run-time errors will ever occur[3]. This formalization effort follows exactly the famous slogan by Robin Milner, "well-typed programs cannot go wrong" [110].

[3]Anyone who has ever programmed on a Unix system knows very well the frustration of a *segmentation fault* error.

3.2.1 Semantics

Throughout the previous section, we introduced the operational semantics rules for the KidML language. The syntax of the language is resumed in Fig. 3.5, while the complete set of rules of the evaluation judgment are given in Fig. 3.6. The **(EvalErr)** rule is explained later in this section. For completeness purposes, we also present the rules of our coinductive judgment, Fig. 3.7. The relation $\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot r$ holds if and only if this is the conclusion of a finite or infinite derivation built from these rules. In fact, to express the special case $\delta \cdot \mu \cdot e \Downarrow^{co} \texttt{div}$ we do here an abuse of notation, since it does not make sense to state divergence together with a resulting store μ'.

Our complete definition of semantics results is as follows:

$$r ::= \bar{v} \mid \texttt{raise}\ E\bar{v} \mid \texttt{absurd} \mid \texttt{div}$$

Judgment properties. Drawing inspiration from the works of Arthur Charguéraud [30, Sec.2.4] and Xavier Leroy [98, lemmas 5–8], let us prove some properties about our two judgments. First, we want to limit the use of our judgments to derivations of the form $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r$ and $\delta \cdot \mu \cdot e \Downarrow^{co} \texttt{div}$. We can only derive $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot \texttt{div}$ if e contains a sub-expression that diverges, as stated via the following lemma:

Lemma 3.2.1. *If* $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r$ *then* $r \neq \texttt{div}$.

Proof. Straightforward induction on $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r$. □

In the following, we only describe the evaluation of divergent expressions via the coinductive judgment. We state, as well, that the coinductive judgment contains the inductive one:

Lemma 3.2.2. *If* $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r$ *then* $\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot r$.

Proof. Straightforward induction on $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r$. □

Next, we show that if $\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot r$ holds then either the evaluation of e diverges, or terminates into some result r. This corresponds to the following lemma:

Lemma 3.2.3. *If* $\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot r$ *then either* $\delta \cdot \mu \cdot e \Downarrow^{co} \texttt{div}$ *or* $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r$.

Proof. By coinduction and case analysis on the rule used at the bottom of the derivation $\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot r$. We detail only the cases of co-evaluation of `let..in` expressions.

case (CoEvalLetAbort). Co-evaluation ends up with

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow^{co} \mu' \cdot r \qquad \texttt{abort}\ r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow^{co} \mu' \cdot r}$$

By co-induction hypotheses either

$$\delta \cdot \mu \cdot e_1 \Downarrow^{co} \texttt{div}$$

or

$$\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot r \qquad r \neq \texttt{div}$$

If it is the first case, we complete the proof with the following derivation:

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow^{co} \texttt{div} \qquad \texttt{abort div}}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow^{co} \texttt{div}}\ (\textsc{CoEvalLetAbort})$$

If it is the second case, we complete the proof with the following derivation:

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot r \qquad \texttt{abort}\ r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow \mu' \cdot r}\ (\textsc{EvalLetAbort})$$

$$\boxed{\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r}$$

$$\frac{}{\delta \cdot \mu \cdot \bar{v} \Downarrow \mu \cdot \bar{v}}\ (\textsc{EvalVBar}) \qquad \frac{}{\delta \cdot \mu \cdot \texttt{raise}\ E\bar{v} \Downarrow \mu \cdot \texttt{raise}\ E\bar{v}}\ (\textsc{EvalRaise})$$

$$\frac{\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{ghost}\ e \Downarrow \mu' \cdot r}\ (\textsc{EvalGhost}) \qquad \frac{\delta \cdot \mu \cdot \texttt{let reg\_} = e\ \texttt{in loop}\ e \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{loop}\ e \Downarrow \mu' \cdot r}\ (\textsc{EvalLoop})$$

$$\frac{}{\delta \cdot \mu \cdot \texttt{absurd} \Downarrow \mu \cdot \texttt{absurd}}\ (\textsc{EvalAbsurd}) \qquad \frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot r \qquad \texttt{abort}\ r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow \mu' \cdot r}\ (\textsc{EvalLetAbort})$$

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot \bar{v} \qquad \delta \cdot \mu' \cdot e_2[\bar{x} \mapsto \bar{v}] \Downarrow \mu'' \cdot r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow \mu'' \cdot r}\ (\textsc{EvalLet})$$

$$\frac{\delta(f) = (\overline{x : \pi}, e) \qquad \|\overline{x : \pi}\| = \|\bar{v}\| \qquad \delta \cdot \mu \cdot e[\bar{x} \mapsto \bar{v}] \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot f\langle\bar{\tau}\rangle(\bar{v}) \Downarrow \mu' \cdot r}\ (\textsc{EvalApp})$$

$$\frac{\delta[f \mapsto (\overline{x : \pi}, e_1)] \cdot \mu \cdot e_2 \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 \Downarrow \mu' \cdot r}\ (\textsc{EvalFun})$$

$$\frac{\delta[f \mapsto (\overline{x : \pi}, e_1)] \cdot \mu \cdot e_2 \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{rec}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 \Downarrow \mu' \cdot r}\ (\textsc{EvalRec})$$

$$\frac{\delta \cdot \mu \cdot e_2 \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{if false then}\ e_1\ \texttt{else}\ e_2 \Downarrow \mu' \cdot r}\ (\textsc{EvalIfFalse})$$

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{if true then}\ e_1\ \texttt{else}\ e_2 \Downarrow \mu' \cdot r}\ (\textsc{EvalIfTrue})$$

$$\frac{\delta \cdot \mu \cdot e_0 \Downarrow \mu' \cdot \bar{v}}{\delta \cdot \mu \cdot \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow \mu' \cdot \bar{v}}\ (\textsc{EvalTry})$$

$$\frac{\delta \cdot \mu \cdot e_0 \Downarrow \mu' \cdot r \qquad \texttt{abort}\ r \qquad \forall i.\ E_i \neq r}{\delta \cdot \mu \cdot \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow \mu' \cdot r}\ (\textsc{EvalTryAbort})$$

$$(\textsc{EvalTryExn})\quad \frac{\delta \cdot \mu \cdot e_0 \Downarrow \mu'' \cdot \texttt{raise}\ E'\bar{v} \qquad E' = E_i \qquad \forall j.\ j < i \rightarrow E' \neq E_j \qquad \delta \cdot \mu'' \cdot e_i[\bar{x} \mapsto \bar{v}] \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow \mu' \cdot r}$$

$$\frac{l \notin \texttt{dom}(\mu) \qquad \mu' = \mu[l \mapsto \{\overline{f = v}\}]}{\delta \cdot \mu \cdot \{\overline{f = v}\} \Downarrow \mu' \cdot l}\ (\textsc{EvalRecord}) \qquad \frac{\mu(l) = \{\ldots f_i = v_i \ldots\}}{\delta \cdot \mu \cdot l.f_i \Downarrow \mu' \cdot v_i}\ (\textsc{EvalGet})$$

$$\frac{\mu(l) = \{\ldots\ f_i = v_i\ \ldots\} \qquad \mu' = \mu[l \mapsto \{\ldots\ f_i = v\ \ldots\}]}{\delta \cdot \mu \cdot l.f_i \leftarrow v \Downarrow \mu' \cdot ()}\ (\textsc{EvalAssign}) \qquad \frac{\neg(e \downarrow)}{\delta \cdot \mu \cdot e \Downarrow \texttt{err}}\ (\textsc{EvalErr})$$

Figure 3.6: Inductive Evaluation Rules.

$$\boxed{\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot r}$$

$$\frac{}{\delta \cdot \mu \cdot \bar{v} \Downarrow^{co} \mu \cdot \bar{v}} \text{(CoEvalVBar)} \qquad \frac{}{\delta \cdot \mu \cdot \texttt{raise}\ E\bar{v} \Downarrow^{co} \mu \cdot \texttt{raise}\ E\bar{v}} \text{(CoEvalRaise)}$$

$$\frac{\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{ghost}\ e \Downarrow^{co} \mu' \cdot r} \text{(CoEvalGhost)} \qquad \frac{}{\delta \cdot \mu \cdot \texttt{absurd} \Downarrow \mu \cdot \texttt{absurd}} \text{(CoEvalAbsurd)}$$

$$\frac{\delta \cdot \mu \cdot \texttt{let reg\_} = e\ \texttt{in loop}\ e \Downarrow^{co} \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{loop}\ e \Downarrow^{co} \mu' \cdot r} \text{(CoEvalLoop)}$$

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow^{co} \mu' \cdot r \qquad \texttt{abort}\ r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow^{co} \mu' \cdot r} \text{(CoEvalLetAbort)}$$

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot \bar{v} \qquad \delta \cdot \mu' \cdot e_2[\bar{x} \mapsto \bar{v}] \Downarrow^{co} \mu'' \cdot r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow^{co} \mu'' \cdot r} \text{(CoEvalLet)}$$

$$\frac{\delta(f) = (\overline{x : \pi}, e) \qquad \|\overline{x : \pi}\| = \|\bar{v}\| \qquad \delta \cdot \mu \cdot e[\bar{x} \mapsto \bar{v}] \Downarrow^{co} \mu' \cdot r}{\delta \cdot \mu \cdot f\langle\overline{\tau}\rangle(\bar{v}) \Downarrow^{co} \mu' \cdot r} \text{(CoEvalApp)}$$

$$\frac{\delta[f \mapsto (\overline{x : \pi}, e_1)] \cdot \mu \cdot e_2 \Downarrow^{co} \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 \Downarrow^{co} \mu' \cdot r} \text{(CoEvalFun)}$$

$$\frac{\delta[f \mapsto (\overline{x : \pi}, e_1)] \cdot \mu \cdot e_2 \Downarrow^{co} \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{rec}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 \Downarrow^{co} \mu' \cdot r} \text{(CoEvalRec)}$$

$$\frac{\delta \cdot \mu \cdot e_2 \Downarrow^{co} \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{if false then}\ e_1\ \texttt{else}\ e_2 \Downarrow^{co} \mu' \cdot r} \text{(CoEvalIfFalse)}$$

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow^{co} \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{if true then}\ e_1\ \texttt{else}\ e_2 \Downarrow^{co} \mu' \cdot r} \text{(CoEvalIfTrue)}$$

$$\frac{\delta \cdot \mu \cdot e_0 \Downarrow^{co} \mu' \cdot \bar{v}}{\delta \cdot \mu \cdot \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow^{co} \mu' \cdot \bar{v}} \text{(CoEvalTry)}$$

$$\frac{\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot r \qquad \texttt{abort}\ r \qquad \forall i.\, E_i \neq r}{\delta \cdot \mu \cdot \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow^{co} \mu' \cdot r} \text{(CoEvalTryAbort)}$$

Figure 3.7: Co-inductive Evaluation Rules (1/2).

$$\boxed{\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot r}$$

(CoEvalTryExn)
$$\frac{\delta \cdot \mu \cdot e_0 \Downarrow \mu'' \cdot \texttt{raise}\ E'\bar{v} \qquad E' = E_i \qquad \forall j.\, j < i \rightarrow E' \neq E_j \qquad \delta \cdot \mu'' \cdot e_i[\bar{x} \mapsto \bar{v}] \Downarrow^{co} \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow^{co} \mu' \cdot r}$$

$$\frac{l \notin \texttt{dom}(\mu) \qquad \mu' = \mu[l \mapsto \{\,\overline{f = v}\,\}]}{\mu \cdot \{\,\overline{f = v}\,\} \Downarrow \mu' \cdot l}\ (\text{CoEvalRecord}) \qquad \frac{\mu(l) = \{\ldots f_i = v_i \ldots\}}{\mu \cdot l.f_i \Downarrow \mu \cdot v_i}\ (\text{CoEvalGet})$$

(CoEvalAssign)
$$\frac{\mu(l) = \{\ldots f_i = v_i \ldots\} \qquad \mu' = \mu[l \mapsto \{\ldots f_i = v \ldots\}]}{\mu \cdot l.f_i \leftarrow v \Downarrow \mu' \cdot ()} \qquad \frac{\neg(e \downarrow)}{\delta \cdot \mu \cdot e \Downarrow^{co} \texttt{err}}\ (\text{CoEvalErr})$$

Figure 3.8: Co-inductive Evaluation Rules (2/2).

case (CoEvalLet). Co-evaluation ends up with

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot \bar{v} \qquad \delta \cdot \mu' \cdot e_2[\bar{x} \mapsto \bar{v}] \Downarrow^{co} \mu'' \cdot r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow^{co} \mu'' \cdot r}$$

By co-induction hypotheses

$$\delta \cdot \mu' \cdot e_2[\bar{x} \mapsto \bar{v}] \Downarrow^{co} \texttt{div}$$

or

$$\delta \cdot \mu' \cdot e_2[\bar{x} \mapsto \bar{v}] \Downarrow \mu'' \cdot r \qquad r \neq \texttt{div}$$

If it is the first case, we complete the proof with the following derivation:

(CoEvalLet)
$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot \bar{v} \qquad \delta \cdot \mu' \cdot e_2[\bar{x} \mapsto \bar{v}] \Downarrow^{co} \texttt{div}}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow^{co} \mu'' \cdot r}$$

If it is the second case, we complete the proof with the following derivation:

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot \bar{v} \qquad \delta \cdot \mu' \cdot e_2[\bar{x} \mapsto \bar{v}] \Downarrow \mu'' \cdot r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow \mu'' \cdot r}\ (\text{EvalLet})$$

□

Finally, the coinductive relation is deterministic for terminating evaluations:

Lemma 3.2.4. *If* $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r$ *and* $\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot r'$ *then* $r = r'$.

Proof. By induction on $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r$ and case analysis on $\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot r'$. □

A corollary of this result is that if $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r$ holds, then $r \neq \texttt{div}$ and $\neg(\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r')$, for any other result r' such that $r \neq r'$.

$\boxed{e\downarrow}$

$$\frac{}{\bar{v}\downarrow}\ (\downarrow\textsc{VBar}) \qquad \frac{}{\texttt{raise}\ E\bar{v}\downarrow}\ (\downarrow\textsc{Raise}) \qquad \frac{}{\texttt{absurd}\downarrow}\ (\downarrow\textsc{Absurd})$$

$$\frac{}{\texttt{ghost}\ e\downarrow}\ (\downarrow\textsc{Ghost}) \qquad \frac{}{\texttt{loop}\ e\downarrow}\ (\downarrow\textsc{Loop}) \qquad \frac{}{\texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2\downarrow}\ (\downarrow\textsc{Let}) \qquad \frac{}{f\langle\bar{\tau}\rangle(\bar{v})\downarrow}\ (\downarrow\textsc{App})$$

$$\frac{}{\texttt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x:\pi}):\sigma = e_1\ \texttt{in}\ e_2\downarrow}\ (\downarrow\textsc{Fun}) \qquad \frac{}{\texttt{rec}\ f\langle\overline{\alpha}\rangle(\overline{x:\pi}):\sigma = e_1\ \texttt{in}\ e_2\downarrow}\ (\downarrow\textsc{Rec})$$

$$\frac{}{\texttt{if true then}\ e_1\ \texttt{else}\ e_2\downarrow}\ (\downarrow\textsc{IfTrue}) \qquad \frac{}{\texttt{if false then}\ e_1\ \texttt{else}\ e_2\downarrow}\ (\downarrow\textsc{IfFalse})$$

$$\frac{}{\texttt{try}\ e_0\ \texttt{with}\ \overline{Ex}\Rightarrow\overline{e}\downarrow}\ (\downarrow\textsc{Try}) \qquad \frac{}{\{\overline{f=a}\}\downarrow}\ (\downarrow\textsc{Record}) \qquad \frac{}{a.f\downarrow}\ (\downarrow\textsc{Get}) \qquad \frac{}{a_r.f\leftarrow a\downarrow}\ (\downarrow\textsc{Assign})$$

Figure 3.9: Progress Judgment.

Generic error rule. To describe expressions whose evaluation "goes wrong", we add to our definition of semantics results the `err` constant, as follows:

$$r ::= \ldots \mid \texttt{err}$$

When the evaluation of an expression gets stuck, we want to completely *abort* the current evaluation. This last sentence says it all: to propagate errors to top level, we must state that `err` satisfies the `abort` predicate. This completes our definition of `abort`, as follows:

$$\frac{}{\texttt{abort raise}\ E\bar{v}} \qquad \frac{}{\texttt{abort absurd}} \qquad \frac{}{\texttt{abort div}} \qquad \frac{}{\texttt{abort err}}$$

The introduction of the `err` constant in our language follows a classical approach when it comes to prove type soundness using a big-step operational semantics. A major drawback of such an approach is that we must augment our evaluation judgment with extra rules to describe terms that get stuck, which increases the size of the semantics and, worse, it can compromise the the type soundness result in case we miss some error rule. In order to circumvent this pitfall, we follow the approach proposed by Arthur Charguéraud [30] and equip our semantics judgment with a *generic error rule*. This captures very smoothly the idea that an expression gets stuck if no evaluation rule can be applied. We define the generic error rule in terms of the *progress judgment*, given Fig. 3.9. Relation $e\downarrow$ holds if there is at least one evaluation rule whose conclusion matches expression e. We introduce the following generic error rule:

$$\frac{\neg(e\downarrow)}{\delta\cdot\mu\cdot e\Downarrow\mu'\cdot\texttt{err}}$$

which asserts that if an evaluation cannot progress, then e must evaluate down to `err` for any procedures environment δ and store μ.

3.2.2 Type Soundness

In this section, we prove the soundness of our type system with respect to the introduced semantics evaluation. We follow a syntactic approach, as proposed by Andrew Wright and Matthias Felleisen

[145]. Using such an approach, type soundness is derived from two lemmas: the first, the *progress* lemma, states that closed well-typed expression can be evaluated down to a semantics result; the second lemma, the *preservation* property, states that a well-typed expression evaluates down to a well-typed result, not necessarily with the same ghost status. Our typing rules are summarized in Fig. 3.10 and 3.11. In order to state the progress lemma, we need the following two definitions:

Definition 3.2.1 (Well-typed store)**.** A store μ is said to be *well typed* with respect to a functions typing context Δ, a variables typing context Γ, and a store typing context Σ, written $\Sigma \vDash \mu$, if $\mathtt{dom}(\mu) = \mathtt{dom}(\Sigma)$ and $\Delta \cdot \Gamma \cdot \Sigma \vdash l : (\mathtt{reg}\,\Sigma(l), \emptyset, \emptyset)$, for every $l \in \mathtt{dom}(\mu)$.

Definition 3.2.2 (Well-typed procedure environment)**.** A procedure environment δ is said to be *well typed* with respect to a functions typing context Δ, a variables typing context Γ, and a store typing context Σ, written $\Delta \vDash \delta$, if $\mathtt{dom}(\delta) = \mathtt{dom}(\Delta)$ and $\Delta \cdot \Gamma + [\overline{x : \pi}] \cdot \Sigma \vdash e : (\mathtt{reg}\,\Sigma(l), \emptyset, \emptyset)$, for every $f \in \mathtt{dom}(\delta)$ such that $\delta(f) = (\overline{x : \pi}, e)$.

We can now state the progress lemma. This is as follows:

Lemma 3.2.5 (Progress)**.** *If* $\Delta \cdot \emptyset \cdot \Sigma \vdash e : \sigma$ *then* $e \downarrow$.

Proof. By case analysis on $\Delta \cdot \emptyset \cdot \Sigma \vdash e : \sigma$.

The proof of this lemma is almost trivial, thanks to the use of judgment $e \downarrow$. At each case, we show that there is at least one semantics evaluation rule that can be applied.

case (TVar). Impossible.

cases (TABar), (TIfAbsurd1), (TIfAbsurd2), (TIfGhost), (TIf), (TRaiseReg), (TApp), (TRaiseGhost), (TAppGhostArg), (TRecord), (TRecordGhost), (TGet), (TGetGhost), (TAssignGhostField), (TAssignGhost), and (TAssign). For each of these cases, every occurrence of an atomic sub-expression must be fully evaluated, since variables are not typeable in the empty environment. We complete the proof of each case by applying the corresponding ($\downarrow$) rule.

cases (TConst) and (TLoc) Particular cases of (TABar).

cases (TGhost), (TLoop), (TAbsurd), (TFun), (TRec), (TLet), and (TTry). For each case, we apply the corresponding ($\downarrow$) rule.

□

We now move to the preservation lemma. Before stating and proving such a result, we introduce a *substitution* lemma, a classic result in the formalization of type systems. This auxiliary result states that performing variable substitution over a well-typed expression results in a well-typed expression. This is later used in the preservation proof to show that, whenever a semantics evaluation involves substitution, the semantic result obtained is well-typed.

Lemma 3.2.6 (Substitution)**.** *If* $\Delta \cdot \Gamma + [\bar{x} : \overline{\beta\tau}] \cdot \Sigma \vdash e : (\overline{\beta_e \tau_e}, \epsilon, \gamma)$ *and* $\Delta \cdot \Gamma \cdot \Sigma \vdash \bar{a} : (\overline{\beta' \tau}, \emptyset, \emptyset)$*, where* $\overline{\beta'} \sqsubseteq \overline{\beta}$ *if* e *is a regular effectful expression, and* $(\overline{\beta_e \tau_e}, \epsilon, \gamma)[\bar{x} \mapsto \bar{a}]$ *does not compromise the global assumption of admissible effects, then* $\Delta \cdot \Gamma \cdot \Sigma \vdash e[\bar{x} \mapsto \bar{a}] : (\overline{\beta'_e \tau_e}, \epsilon, \gamma)[\bar{x} \mapsto \bar{a}]$ *for some* $\overline{\beta'_e}$*. Moreover, if* $\overline{\beta'} \sqsubseteq \overline{\beta}$ *then* $\overline{\beta'_e} \sqsubseteq \overline{\beta_e}$.

$\boxed{\Delta \cdot \Gamma \cdot \Sigma \vdash e : \sigma}$

$$\frac{\texttt{Typeof}(c) = \tau}{\Delta \cdot \Gamma \cdot \Sigma \vdash c : (\texttt{reg}\,\tau,\ \emptyset,\ \emptyset)}\ (\textsc{TConst}) \qquad \frac{\Sigma(l) = \mathsf{T}\overline{\tau}}{\Delta \cdot \Gamma \cdot \Sigma \vdash l : (\texttt{reg}\,\mathsf{T}\overline{\tau},\ \emptyset,\ \emptyset)}\ (\textsc{TLoc})$$

$$\frac{\Gamma(x) = \pi}{\Delta \cdot \Gamma \cdot \Sigma \vdash x : (\pi,\ \emptyset,\ \emptyset)}\ (\textsc{TVar}) \qquad \frac{\forall i.\ \Delta \cdot \Gamma \cdot \Sigma \vdash a_i : (\pi_i, \emptyset)}{\Delta \cdot \Gamma \cdot \Sigma \vdash \bar{a} : (\overline{\pi},\ \emptyset,\ \emptyset)}\ (\textsc{TABar})$$

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e : \sigma}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{ghost}\ e : \text{👻}(\sigma)}\ (\textsc{TGhost}) \qquad \frac{}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{absurd} : (\overline{\texttt{reg}\,\tau},\ \emptyset,\ \emptyset)}\ (\textsc{TAbsurd})$$

$$\frac{\begin{array}{c}\texttt{CheckTermination}(\texttt{loop}\ e) \\ \Delta \cdot \Gamma \cdot \Sigma \vdash e : (\texttt{Unit},\ \epsilon,\ \gamma)\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{loop}\ e : (\overline{\texttt{reg}\,\tau},\ \epsilon,\ \gamma)}\ (\textsc{TLoop}) \qquad \frac{\begin{array}{c}\neg\texttt{CheckTermination}(\texttt{loop}\ e) \\ \Delta \cdot \Gamma \cdot \Sigma \vdash e : (\texttt{Unit},\ \epsilon,\ \gamma)\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{loop}\ e : (\overline{\texttt{reg}\,\tau},\ \epsilon + \texttt{div},\ \gamma)}\ (\textsc{TLoopDiv})$$

$$\frac{\begin{array}{c}\Delta \cdot \Gamma + [\overline{x : \pi}] \cdot \Sigma \vdash e_1 : \sigma' \qquad \sigma'_{\restriction \Gamma + [\overline{x:\pi}]} \sqsubseteq \sigma \\ \Delta + [f : \forall \overline{\alpha}.(\overline{x : \pi}) \to \sigma] \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma'' \qquad \overline{\alpha} \# \Gamma \qquad \overline{\alpha} \notin \Delta\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 : \sigma''}\ (\textsc{TFun})$$

$$\frac{\begin{array}{c}\Delta + [f : \forall \overline{\alpha}.(\overline{x : \pi}) \to \sigma] \cdot \Gamma + [\overline{x : \pi}] \cdot \Sigma \vdash e_1 : \sigma' \qquad \sigma'_{\restriction \Gamma + [\overline{x:\pi}]} \sqsubseteq \sigma \\ \neg\texttt{CheckTermination}(\texttt{rec}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1) \\ \sigma = (\overline{\beta\tau}, \epsilon + \texttt{div}, \gamma) \qquad \Delta + [f : \forall \overline{\alpha}.(\overline{x : \pi}) \to \sigma] \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma'' \qquad \overline{\alpha} \notin \Gamma \qquad \overline{\alpha} \notin \Delta\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{rec}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 : \sigma''}\ (\textsc{TRecDiv})$$

$$\frac{\begin{array}{c}\Delta + [f : \forall \overline{\alpha}.(\overline{x : \pi}) \to \sigma] \cdot \Gamma + [\overline{x : \pi}] \cdot \Sigma \vdash e_1 : \sigma' \qquad \sigma'_{\restriction \Gamma + [\overline{x:\pi}]} \sqsubseteq \sigma \\ \texttt{CheckTermination}(\texttt{rec}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1) \\ \sigma = (\overline{\beta\tau}, \epsilon, \gamma) \qquad \Delta + [f : \forall \overline{\alpha}.(\overline{x : \pi}) \to \sigma] \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma'' \qquad \overline{\alpha} \notin \Gamma \qquad \overline{\alpha} \notin \Delta\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{rec}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 : \sigma''}\ (\textsc{TRec})$$

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash a : (\beta\texttt{Bool},\ \emptyset,\ \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{if}\ a\ \texttt{then}\ \texttt{absurd}\ \texttt{else}\ e_2 : \sigma}\ (\textsc{TIfAbsurd1})$$

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash a : (\beta\texttt{Bool},\ \emptyset,\ \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_1 : \sigma}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ \texttt{absurd} : \sigma}\ (\textsc{TIfAbsurd2})$$

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash a : (\texttt{ghost}\,\texttt{Bool},\ \emptyset,\ \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_1 : \sigma_1 \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma_2}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \text{👻}(\sigma_1 \cup \sigma_2)}\ (\textsc{TIfGhost})$$

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash a : (\texttt{reg}\,\texttt{Bool},\ \emptyset,\ \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_1 : \sigma_1 \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma_2}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \sigma_1 \cup \sigma_2}\ (\textsc{TIf})$$

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_1 : (\overline{\beta_1\tau_1},\ \epsilon_1,\ \gamma_1) \qquad \overline{\beta_1} \sqsubseteq \overline{\beta} \qquad \Delta \cdot \Gamma + [\overline{x : \beta\tau_1}] \cdot \Sigma \vdash e_2 : (\overline{\pi_2},\ \epsilon_2,\ \gamma_2)}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 : (\overline{\pi_2},\ \epsilon_1 \cup \epsilon_2,\ \gamma_1 \cup \gamma_2)}\ (\textsc{TLet})$$

Figure 3.10: Typing rules for expressions (1/2).

$\boxed{\Delta \cdot \Gamma \cdot \Sigma \vdash e : \sigma}$

$$\frac{\theta = [\overline{\alpha} \mapsto \overline{\tau}] \qquad \rho = [\bar{x} \mapsto \bar{a}] \qquad \|\overline{\alpha}\| = \|\overline{\tau}\| \qquad \|\bar{x}\| = \|\bar{a}\| \\ \Delta(f) = \forall \overline{\alpha}.(\overline{x : \beta\tau'}) \to \sigma \qquad \forall i.\, \Delta \cdot \Gamma \cdot \Sigma \vdash a_i : (\beta_i'\tau_i'\theta,\, \emptyset,\, \emptyset) \qquad \forall i.\, \beta_i' \sqsubseteq \beta_i}{\Delta \cdot \Gamma \cdot \Sigma \vdash f\langle\overline{\tau}\rangle\,(\bar{a}) : \sigma\theta\rho} \text{ (TApp)}$$

$$\frac{\theta = [\overline{\alpha} \mapsto \overline{\tau}] \qquad \rho = [\bar{x} \mapsto \bar{a}] \qquad \|\overline{\alpha}\| = \|\overline{\tau}\| \qquad \|\bar{x}\| = \|\bar{a}\| \\ \Delta(f) = (\overline{x : \beta\tau'}) \to \sigma \qquad \forall i.\Delta \cdot \Gamma \cdot \Sigma \vdash a_i : (\beta_i'\tau_i'\theta, \emptyset, \emptyset) \qquad \exists i.\, \beta_i \sqsubset \beta_i'}{\Delta \cdot \Gamma \cdot \Sigma \vdash f\langle\overline{\tau}\rangle\,(\bar{a}) : \text{Ω}(\sigma\theta\rho)} \text{ (TAppGhostArg)}$$

$$\frac{\texttt{exception}\ E : (\beta_1\tau_1 \ldots \beta_n\tau_n) \qquad \forall i.\, \Delta \cdot \Gamma \cdot \Sigma \vdash a_i : (\beta_i'\tau_i, \emptyset, \emptyset) \qquad \forall i.\, \beta_i' \sqsubseteq \beta_i}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{raise}\ E\bar{a} : (\overline{\texttt{reg}\,\tau},\, \texttt{raises}\ E,\, \emptyset)} \text{ (TRaiseReg)}$$

$$\frac{\texttt{exception}\ E : (\beta_1\tau_1 \ldots \beta_n\tau_n) \qquad \forall i.\, \Delta \cdot \Gamma \cdot \Sigma \vdash a_i : (\beta_i'\tau_i, \emptyset, \emptyset) \qquad \exists i.\, \beta_i \sqsubset \beta_i'}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{raise}\ E\bar{a} : (\overline{\texttt{ghost}\,\tau},\, \emptyset,\, \texttt{raises}\ E)} \text{ (TRaiseGhost)}$$

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_0 : (\overline{\pi},\, \epsilon,\, \gamma) \qquad \forall i.\, E_i : \overline{\pi_i} \qquad \forall i.\, \Delta \cdot \Gamma + [\bar{x} : \overline{\pi_i}] \cdot \Sigma \vdash e_i : \sigma_i}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} : (\overline{\pi},\, \epsilon - \overline{\texttt{raises}\ E_i},\, \gamma - \overline{\texttt{raises}\ E_i}) \cup \bigcup_i \sigma_i} \text{ (TTry)}$$

$$\frac{\texttt{type}\ T\overline{\alpha} = \{\overline{f : \beta_f\tau_f}\} \qquad \forall i.\, \Delta \cdot \Gamma \cdot \Sigma \vdash a_i : (\beta_i\tau_{f_i}[\overline{\alpha} \mapsto \overline{\tau}],\, \emptyset,\, \emptyset) \qquad \forall i.\, \beta_i \sqsubseteq \beta_{f_i}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \{\overline{f = a}\} : (\texttt{reg}\ T\overline{\tau},\, \emptyset,\, \emptyset)} \text{ (TRecord)}$$

$$\frac{\texttt{type}\ T\overline{\alpha} = \{\overline{f : \beta_f\tau_f}\} \qquad \forall i.\, \Delta \cdot \Gamma \cdot \Sigma \vdash a_i : (\beta_i\tau_{f_i}[\overline{\alpha} \mapsto \overline{\tau}],\, \emptyset,\, \emptyset) \qquad \exists i.\, \beta_{f_i} \sqsubset \beta_i}{\Delta \cdot \Gamma \cdot \Sigma \vdash \{\overline{f = a}\} : (\texttt{ghost}\ T\overline{\tau},\, \emptyset,\, \emptyset)} \text{ (TRecordGhost)}$$

$$\frac{\texttt{type}\ T\overline{\alpha} = \{\ldots,\, f : \beta_f\tau_f,\, \ldots\} \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash a : (\texttt{reg}\ T\overline{\tau},\, \emptyset,\, \emptyset)}{\Delta \cdot \Gamma \cdot \Sigma \vdash a.f : (\beta_f\ \tau_f[\overline{\alpha} \mapsto \overline{\tau}],\, \emptyset,\, \emptyset)} \text{ (TGet)}$$

$$\frac{\texttt{type}\ T\overline{\alpha} = \{\ldots,\, f : \beta_f\tau_f,\, \ldots\} \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash a : (\texttt{ghost}\ T\overline{\tau},\, \emptyset,\, \emptyset)}{\Delta \cdot \Gamma \cdot \Sigma \vdash a.f : (\texttt{ghost}\ \tau_f[\overline{\alpha} \mapsto \overline{\tau}],\, \emptyset,\, \emptyset)} \text{ (TGetGhost)}$$

$$\frac{\texttt{type}\ T\overline{\alpha} = \{\ldots, f : \texttt{ghost}\ \tau_f, \ldots\} \\ \Delta \cdot \Gamma \cdot \Sigma \vdash a_l : (\beta_l T\overline{\tau},\, \emptyset,\, \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash a_r : (\beta_r\tau_f[\overline{\alpha} \mapsto \overline{\tau}],\, \emptyset,\, \emptyset)}{\Delta \cdot \Gamma \cdot \Sigma \vdash a_l.f \leftarrow a_r : (\texttt{Unit},\, \emptyset,\, \emptyset + \texttt{writes}\ a_l.f)} \text{ (TAssignGhostField)}$$

$$\frac{\texttt{type}\ T\overline{\alpha} = \{\ldots, f : \texttt{reg}\ \tau_f, \ldots\} \\ \Delta \cdot \Gamma \cdot \Sigma \vdash a_l : (\texttt{ghost}\ T\overline{\tau},\, \emptyset,\, \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash a_r : (\beta_r\tau_f[\overline{\alpha} \mapsto \overline{\tau}],\, \emptyset,\, \emptyset)}{\Delta \cdot \Gamma \cdot \Sigma \vdash a_l.f \leftarrow a_r : (\texttt{Unit},\, \emptyset, \emptyset + \texttt{writes}\ a_l.f)} \text{ (TAssignGhost)}$$

$$\frac{\texttt{type}\ T\overline{\alpha} = \{\ldots, f : \texttt{reg}\ \tau_f, \ldots\} \\ \Delta \cdot \Gamma \cdot \Sigma \vdash a_l : (\texttt{reg}T\overline{\tau},\, \emptyset,\, \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash a_r : (\texttt{reg}\tau_f[\overline{\alpha} \mapsto \overline{\tau}],\, \emptyset,\, \emptyset)}{\Delta \cdot \Gamma \cdot \Sigma \vdash a_l.f \leftarrow a_r : (\texttt{Unit}, \emptyset + \texttt{writes}\ a_l.f,\, \emptyset)} \text{ (TAssign)}$$

Figure 3.11: Typing rules for expressions (2/2).

Proof. By induction on the derivation $\Delta \cdot \Gamma + [\bar{x} : \overline{\beta\tau}] \cdot \Sigma \vdash e : (\overline{\beta_e \tau_e}, \epsilon, \gamma)$.

The interesting cases happen when expression e is a variable or an assignment. The other cases are either immediate or can be deduced from the induction hypotheses. In the cases of `fun..in`, `rec..in`, `let..in`, and `try..with` expressions we additionally need to use standard weakening and permutations lemmas [125], which pose no particular difficulty, and so we do not detail them here.

case (TVar). The typing derivation ends up with

$$\frac{\Gamma(y) = \pi}{\Delta \cdot \Gamma + [\bar{x} : \overline{\beta\tau}] \cdot \Sigma \vdash y : (\pi,\ \emptyset,\ \emptyset)}$$

If variable y does not occur in $\bar{x}$, the result follows immediately. On the other hand, if y is equal to some x_i, then we have

$$y[\bar{x} \mapsto \bar{a}] \equiv a_i$$

The following holds by the hypotheses of the lemma

$$\Delta \cdot \Gamma \cdot \Sigma \vdash a_i : \beta' \tau_i, \emptyset, \emptyset$$

which concludes this case.

case (TAssign). The typing derivation ends up with

$$\frac{\texttt{type}\ T\overline{\alpha} = \{\ldots, f : \texttt{reg}\ \tau_f, \ldots\} \qquad \Delta \cdot \Gamma + [\bar{x} : \overline{\beta\tau}] \cdot \Sigma \vdash a_l : (\texttt{reg} T\overline{\tau},\ \emptyset,\ \emptyset) \qquad \Delta \cdot \Gamma + [\bar{x} : \overline{\beta\tau}] \cdot \Sigma \vdash a_r : (\texttt{reg}\ \tau_f[\overline{\alpha} \mapsto \overline{\tau}],\ \emptyset,\ \emptyset)}{\Delta \cdot \Gamma + [\bar{x} : \overline{\beta\tau}] \cdot \Sigma \vdash a_l.f \leftarrow a_r : (\texttt{Unit}, \emptyset + \texttt{writes}\ a_l.f,\ \emptyset)}$$

Since e is an effectful expression, we have $\overline{\beta'} \sqsubseteq \overline{\beta}$, *i.e.*, the atoms a_l and a_r can only be replaced by regular expressions. This prevents us from substituting regular locations by ghost ones, which would introduce a new ghost effect and eliminate an existing regular effect. By IH

$$\Delta \cdot \Gamma \cdot \Sigma \vdash a_l[\bar{x} \mapsto \bar{a}] : (\texttt{reg} T\overline{\tau}, \emptyset, \emptyset)$$

$$\Delta \cdot \Gamma \cdot \Sigma \vdash a_r[\bar{x} \mapsto \bar{a}] : (\texttt{reg}\ \tau_f[\overline{\alpha} \mapsto \overline{\tau}], \emptyset, \emptyset)$$

We can build the following derivation:

$$\frac{\texttt{type}\ T\overline{\alpha} = \{\ldots, f : \texttt{reg}\ \tau_f, \ldots\} \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash a_l[\bar{x} \mapsto \bar{a}] : (\texttt{reg} T\overline{\tau},\ \emptyset,\ \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash a_r[\bar{x} \mapsto \bar{a}] : (\texttt{reg}\ \tau_f[\overline{\alpha} \mapsto \overline{\tau}],\ \emptyset,\ \emptyset)}{\Delta \cdot \Gamma \cdot \Sigma \vdash a_l[\bar{x} \mapsto \bar{a}].f \leftarrow a_r[\bar{x} \mapsto \bar{a}] : (\texttt{Unit}, \emptyset + \texttt{writes}\ a_l.f,\ \emptyset)[\bar{x} \mapsto \bar{a}]}$$

which completes this case.

case (TAssignGhost). The typing derivation ends up with

$$\frac{\texttt{type}\ T\overline{\alpha} = \{\ldots, f : \texttt{reg}\ \tau_f, \ldots\} \qquad \Delta \cdot \Gamma + [\bar{x} : \overline{\beta\tau}] \cdot \Sigma \vdash a_l : (\texttt{ghost}\ T\overline{\tau},\ \emptyset,\ \emptyset) \qquad \Delta \cdot \Gamma + [\bar{x} : \overline{\beta\tau}] \cdot \Sigma \vdash a_r : (\beta_r \tau_f[\overline{\alpha} \mapsto \overline{\tau}],\ \emptyset,\ \emptyset)}{\Delta \cdot \Gamma + [\bar{x} : \overline{\beta\tau}] \cdot \Sigma \vdash a_l.f \leftarrow a_r : (\texttt{Unit},\ \emptyset, \emptyset + \texttt{writes}\ a_l.f)}$$

We can only replace a_l by another atomic ghost expression a_i. Otherwise, we would be introducing a regular effect via a ghost assignment (aliasing), which conflicts with our global condition of admissible effects. By IH

$$\Delta \cdot \Gamma \cdot \Sigma \vdash a_l[\bar{x} \mapsto \bar{a}] : (\texttt{ghost} T\overline{\tau}, \emptyset, \emptyset)$$

$$\Delta \cdot \Gamma \cdot \Sigma \vdash a_r[\bar{x} \mapsto \bar{a}] : (\beta'_r\ \tau_f[\overline{\alpha} \mapsto \overline{\tau}], \emptyset, \emptyset)$$

We can build the following derivation:

$$\frac{\begin{array}{c}\texttt{type}\ T\overline{\alpha} = \{\ldots, f : \texttt{reg}\ \tau_f, \ldots\} \\ \Delta \cdot \Gamma \cdot \Sigma \vdash a_l[\overline{x} \mapsto \overline{a}] : (\texttt{ghost}\ T\overline{\tau},\ \emptyset,\ \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash a_r[\overline{x} \mapsto \overline{a}] : (\beta_r\tau_f[\overline{\alpha} \mapsto \overline{\tau}],\ \emptyset,\ \emptyset)\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash a_l.f[\overline{x} \mapsto \overline{a}] \leftarrow a_r[\overline{x} \mapsto \overline{a}] : (\texttt{Unit},\ \emptyset, \emptyset + \texttt{writes}\ a_l.f)[\overline{x} \mapsto \overline{a}]}$$

which completes this case.

case (TAssignGhostField). The typing derivation ends up with

$$\frac{\begin{array}{c}\texttt{type}\ T\overline{\alpha} = \{\ldots, f : \texttt{ghost}\ \tau_f, \ldots\} \\ \Delta \cdot \Gamma + [\overline{x} \mapsto \overline{a}] \cdot \Sigma \vdash a_l : (\beta_l T\overline{\tau},\ \emptyset,\ \emptyset) \qquad \Delta \cdot \Gamma + [\overline{x} \mapsto \overline{a}] \cdot \Sigma \vdash a_r : (\beta_r\tau_f[\overline{\alpha} \mapsto \overline{\tau}],\ \emptyset,\ \emptyset)\end{array}}{\Delta \cdot \Gamma + [\overline{x} \mapsto \overline{a}] \cdot \Sigma \vdash a_l.f \leftarrow a_r : (\texttt{Unit},\ \emptyset,\ \emptyset + \texttt{writes}\ a_l.f)}$$

By IH

$$\Delta \cdot \Gamma \cdot \Sigma \vdash a_l[\overline{x} \mapsto \overline{a}] : (\beta_l' T\overline{\tau},\ \emptyset,\ \emptyset)$$
$$\Delta \cdot \Gamma \cdot \Sigma \vdash a_r[\overline{x} \mapsto \overline{a}] : (\beta_r' T\overline{\tau},\ \emptyset,\ \emptyset)$$

We can build the following derivation:

$$\frac{\begin{array}{c}\texttt{type}\ T\overline{\alpha} = \{\ldots, f : \texttt{ghost}\ \tau_f, \ldots\} \\ \Delta \cdot \Gamma \cdot \Sigma \vdash a_l[\overline{x} \mapsto \overline{a}] : (\beta_l' T\overline{\tau},\ \emptyset,\ \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash a_r[\overline{x} \mapsto \overline{a}] : (\beta_r'\tau_f[\overline{\alpha} \mapsto \overline{\tau}],\ \emptyset,\ \emptyset)\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash a_l[\overline{x} \mapsto \overline{a}].f \leftarrow a_r[\overline{x} \mapsto \overline{a}] : (\texttt{Unit},\ \emptyset,\ \emptyset + \texttt{writes}\ a_l.f)[\overline{x} \mapsto \overline{a}]}$$

which completes this case.

□

A last step before stating the preservation property is to show that if an expression e is typeable under a certain store typing Σ, then we can extend Σ with fresh bindings and e is still typeable.

Lemma 3.2.7 (Extension of Store Typing Context)**.** *If* $\Delta \cdot \Gamma \cdot \Sigma \vdash e : \sigma$ *then* $\Delta \cdot \Gamma \cdot \Sigma' \vdash e : \sigma$*, where* $\Sigma \subseteq \Sigma'$*,* i.e.*,* $\texttt{dom}(\Sigma) \subseteq \texttt{dom}(\Sigma')$ *and* $\forall\, l \in \texttt{dom}(\Sigma).\ \Sigma'(l) = \Sigma(l)$*.*

Proof. Straightforward induction on $\Delta \cdot \Gamma \cdot \Sigma \vdash e : \sigma$. □

We can finally state the preservation lemma, as follows:

Lemma 3.2.8 (Preservation)**.** *If* $\Delta \cdot \emptyset \cdot \Sigma \vdash e : (\overline{\beta\tau}, \epsilon, \gamma)$ *and* $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r$*, where* $\Sigma \vDash \mu$ *and* $\Delta \vDash \delta$*, then* $\Delta \cdot \emptyset \cdot \Sigma' \vdash r : (\overline{\beta'\tau}, \epsilon', \gamma')$ *where* $\overline{\beta'} \sqsubseteq \overline{\beta}$*,* $\epsilon' \subseteq \epsilon$*,* $\gamma' \subseteq \gamma$*,* $\Sigma \subseteq \Sigma'$*, and* $\Sigma' \vDash \mu'$*.*

Proof. By induction on the derivation $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r$ and by case analysis on the typing rule used at the bottom of the derivation.

cases (EvalVBar), (EvalRaise), and (EvalAbsurd). Trivial.

case (EvalRecord). Using lemma 3.2.7.

case (EvalGhost). Semantics evaluation ends up with

$$\frac{\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{ghost}\ e \Downarrow \mu' \cdot r}$$

The typing derivation ends up with

$$\frac{\Delta \cdot \emptyset \cdot \Sigma \vdash e : (\overline{\beta\tau}, \epsilon, \gamma)}{\Delta \cdot \emptyset \cdot \Sigma \vdash \texttt{ghost}\ e : \text{👻}(\overline{\beta\tau}, \epsilon, \gamma)}$$

By IH

$$\Delta \cdot \emptyset \cdot \Gamma' \vdash r : (\overline{\beta'\tau}, \epsilon', \gamma')$$

where $\overline{\beta'} \sqsubseteq \overline{\beta}$, $\epsilon' \subseteq \epsilon$, $\gamma' \subseteq \gamma$, $\Sigma \subseteq \Sigma'$, and $\Sigma' \vDash \mu'$.

case (EvalLoop). Semantics evaluation ends up with

$$\frac{\delta \cdot \mu \cdot \texttt{let reg \_} = e\ \texttt{in loop}\ e \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{loop}\ e \Downarrow \mu' \cdot r}$$

We distinguish two cases depending on which rule was used at the conclusion of the type derivation.

- case (**TLoop**): typing derivation ends up with

$$\frac{\texttt{CheckTermination(loop}\ e\texttt{)} \qquad \Delta \cdot \emptyset \cdot \Sigma \vdash e : (\texttt{Unit}, \epsilon, \gamma)}{\Delta \cdot \emptyset \cdot \Sigma \vdash \texttt{loop}\ e : (\overline{\texttt{reg}\ \tau}, \epsilon, \gamma)}$$

  The following judgment is valid:

$$\frac{\Delta \cdot \emptyset \cdot \Sigma \vdash e : (\texttt{Unit}, \epsilon, \gamma) \qquad \Delta \cdot [\_ : \texttt{reg Unit}] \cdot \Sigma \vdash \texttt{loop}\ e : (\overline{\texttt{reg}\ \tau}, \epsilon, \gamma)}{\Delta \cdot \emptyset \cdot \Sigma \vdash \texttt{let reg \_} = e\ \texttt{in loop}\ e : (\texttt{reg}\ \tau, \epsilon, \gamma)}$$

  By IH

$$\Delta \cdot \emptyset \cdot \Sigma' \vdash r : (\texttt{reg}\ \tau, \epsilon', \gamma')$$

  where $\epsilon' \subseteq \epsilon$, $\gamma' \subseteq \gamma$, $\Sigma \subseteq \Sigma'$, and $\Sigma' \vDash \mu'$.

- case (**TLoopDiv**): similar to the previous case.

case (EvalLetAbort). Semantics evaluation ends up with

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot r \qquad \texttt{abort}\ r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow \mu' \cdot r}$$

We can type an abort response with any type so we trivially have that

$$\Delta \cdot \emptyset \cdot \Sigma' \vdash r : (\overline{\texttt{reg}\ \tau_2}, \emptyset, \emptyset)$$

case (EvalLet). Semantics evaluation ends up with

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot \bar{v} \qquad \delta \cdot \mu' \cdot e_2[\bar{x} \mapsto \bar{v}] \Downarrow \mu'' \cdot r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 \Downarrow \mu'' \cdot r}$$

Typing derivation ends up with

$$\frac{\Delta \cdot \emptyset \cdot \Sigma \vdash e_1 : (\overline{\beta_1 \tau_1},\ \epsilon_1, \gamma_1) \qquad \overline{\beta_1} \sqsubseteq \overline{\beta} \qquad \Delta \cdot [\bar{x} : \overline{\beta \tau_1}] \cdot \Sigma \vdash e_2 : (\overline{\beta_2 \tau_2}, \epsilon_2, \gamma_2)}{\Delta \cdot \emptyset \cdot \Sigma \vdash \texttt{let}\ \overline{\beta x} = e_1\ \texttt{in}\ e_2 : (\overline{\beta_2 \tau_2}, \epsilon_1 \cup \epsilon_2, \gamma_1 \cup \gamma_2)}$$

By IH

$$\Delta \cdot \emptyset \cdot \Sigma' \vdash \bar{v} : (\overline{\beta_1' \tau_1}, \epsilon_1', \gamma_1')$$

where $\overline{\beta_1'} \sqsubseteq \overline{\beta_1}$, $\epsilon_1' \subseteq \epsilon_1$, $\gamma_1' \subseteq \gamma_1$, $\Sigma \subseteq \Sigma'$, and $\Sigma' \vdash \mu'$.

By auxiliary lemma 3.2.7 we have that $\Delta \cdot \emptyset \cdot \Sigma' \vdash e_2 : (\overline{\beta_2 \tau_2}, \epsilon_2, \gamma_2)$. By the substitution lemma we have

$$\Delta \cdot \emptyset \cdot \Sigma' \vdash e_2[\bar{x} \mapsto \bar{v}] : (\overline{\beta_2' \tau_2}, \epsilon_2, \gamma_2)$$

where $\overline{\beta_2'} \sqsubseteq \overline{\beta_2}$, since $\overline{\beta_1'} \sqsubseteq \overline{\beta_1}$. By IH

$$\Delta \cdot \emptyset \cdot \Sigma'' \vdash r : (\overline{\beta_2'' \tau_2}, \epsilon_2', \gamma_2')$$

where $\overline{\beta_2''} \sqsubseteq \overline{\beta_2}$ (by transitivity), $\epsilon_2' \subseteq \epsilon_2 \subseteq \epsilon_1 \cup \epsilon_2$, $\gamma_2' \subseteq \gamma_2 \subseteq \gamma_1 \cup \gamma_2$, $\Sigma \subseteq \Sigma''$, and $\Sigma'' \vDash \mu''$.

case (EvalApp). Semantics evaluation ends up with

$$\frac{\delta(f) = (\overline{x : \beta \tau}, e) \qquad \|\overline{x : \pi}\| = \|\bar{v}\| \qquad \delta \cdot \mu \cdot e[\bar{x} \mapsto \bar{v}] \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot f(\bar{v}) \Downarrow \mu' \cdot r}$$

We distinguish two cases depending on which rule was used at the conclusion of the type derivation.

- case (TApp): typing derivation ends up with

$$\frac{\begin{array}{c}\theta = [\overline{\alpha} \mapsto \overline{\tau}] \qquad \rho = [\bar{x} \mapsto \bar{a}] \\ \Delta(f) = \forall \overline{\alpha}.(\overline{x : \beta \tau}) \to (\overline{\beta_e \tau_e}, \epsilon_e, \gamma_e) \qquad \forall i.\, \Delta \cdot \emptyset \cdot \Sigma \vdash \nu_i : (\beta_i' \tau_i \theta, \emptyset, \emptyset) \qquad \beta_i' \sqsubseteq \beta_i\end{array}}{\Delta \cdot \emptyset \cdot \Sigma \vdash f\ (\bar{v}) : (\overline{\beta_e \tau_e}, \epsilon_e, \gamma_e)}$$

  We know that $\Delta \cdot [\bar{x} : \overline{\beta \tau}] \cdot \Sigma \vdash e : (\overline{\beta_e \tau_e}, \epsilon_e)$ is derivable. By the substitution lemma

$$\Delta \cdot \emptyset \cdot \Sigma \vdash e[\bar{x} \mapsto \bar{v}] : (\overline{\beta_e' \tau_e}, \epsilon_e, \gamma_e)$$

  where $\overline{\beta_e'} \sqsubseteq \overline{\beta_e}$, since $\overline{\beta_i'} \sqsubseteq \overline{\beta_i}$. By IH

$$\Delta \cdot \emptyset \cdot \Sigma' \vdash r : (\overline{\beta_e'' \tau_e}, \epsilon_e', \gamma_e')$$

  where $\overline{\beta_e''} \sqsubseteq \overline{\beta_e}$, $\epsilon_e' \subseteq \epsilon_e$, $\gamma_e' \subseteq \gamma_e$, $\Sigma \subseteq \Sigma'$, and $\Sigma' \vdash \mu'$.

- case (TAppGhostArg): similar to the previous case. The only difference is that the mask of r obtained after applying the substitution lemma and the induction hypotheses is trivially less ghost than the mask of $\text{👻}(\sigma\theta\rho)$.

cases (EvalFun), (EvalRec), (EvalIfTrue), (EvalIfFalse), and (EvalTry). Straightforward induction over the sub-expressions.

case (EvalTryAbort). Semantics evaluation ends up

$$\frac{\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r \qquad \texttt{abort}\ r \qquad \forall i.\, E_i \neq r}{\delta \cdot \mu \cdot \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow \mu' \cdot r}$$

We can type an abort response with any type so we trivially have that

$$\Delta \cdot \emptyset \cdot \Sigma' \vdash r : (\overline{\pi}, \emptyset, \emptyset)$$

case (EvalTryExn). Semantics evaluation ends up with

$$\frac{\delta \cdot \mu \cdot e_0 \Downarrow \mu'' \cdot \texttt{raise}\ E'\bar{v} \qquad E' = E_i \qquad \forall j.\, j < i \rightarrow E' \neq E_j \qquad \delta \cdot \mu'' \cdot e_i[\bar{x} \mapsto \bar{v}] \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow \mu' \cdot r}$$

The typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_0 : (\overline{\pi},\ \epsilon,\ \gamma) \qquad \forall i.\, E_i : \overline{\pi_i} \qquad \forall i.\, \Delta \cdot \Gamma + [\bar{x} : \overline{\pi_i}] \cdot \Sigma \vdash e_i : \sigma_i}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} : (\overline{\pi},\ \epsilon - \overline{\texttt{raises}\ E_i},\ \gamma - \overline{\texttt{raises}\ E_i}) \cup \bigcup_i \sigma_i}$$

We have trivially that

$$\Delta \cdot \emptyset \cdot \Sigma'' \vdash \texttt{raise}\ E'\bar{v} : (\overline{\pi}, \emptyset, \emptyset)$$

where $\Sigma \subseteq \Sigma''$, and $\Sigma'' \vdash \mu''$. By auxiliary lemma 3.2.7 we have that $\forall i. \Delta \cdot \Sigma'' \cdot [\bar{x} : \overline{\pi_i}] \vdash e_i : \sigma_i$. By the substitution lemma we have

$$\Delta \cdot \emptyset \cdot \Sigma'' \vdash e_i[\bar{x} \mapsto \bar{v}] : (\overline{\beta_i \tau_i}, \epsilon_i, \gamma_i)$$

where $\overline{\beta_i}$ is a less ghost mask than the mask of σ_i, since the sequence $\bar{v}$ is typed with a completely regular mask. By IH

$$\Delta \cdot \emptyset \cdot \Sigma' \vdash r : (\overline{\beta_i' \tau_i}, \emptyset, \emptyset)$$

where $\overline{\beta_i'} \sqsubseteq \overline{\beta_i}$, $\Sigma \subseteq \Sigma'$, and $\Sigma' \vdash \mu'$.

cases (EvalGet) and (EvalAssign). Trivial.

case (EvalErr). Impossible.

□

Using the progress and preservation results, we can now prove the desired *type soundness* theorem. We state that a well-typed closed expression cannot evaluate down to an error. This is strongly inspired by Theorem 2 of the paper *Pretty-Big Step Semantics*, by Arthur Charguéraud [30].

Theorem 3.2.9 (Type Soundness)**.** *If* $\Delta \cdot \emptyset \cdot \Sigma \vdash e : \sigma$ *then*

1. $\neg(\delta \cdot \mu \cdot e \Downarrow \mu' \cdot \texttt{err})$
2. $\neg(\delta \cdot \mu \cdot e \Downarrow^{co} \mu' \cdot \texttt{err})$

Proof.

1. By induction on $\Delta \cdot \emptyset \cdot \Sigma \vdash e : \sigma$, we apply the progress lemma to establish that a well-typed sub-expression can be (co-)evaluated. Since `err` is not a well-typed expression, it is never the case that `err` is evaluated. If a well-typed sub-expression evaluates down to a semantic result, then we use preservation to show that it evaluates down to a well-typed result. Since `err` is not a well-typed expression, it cannot be the result of an evaluation.
2. By lemma 3.2.3, either $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot r$ or $\delta \cdot \mu \cdot e \Downarrow^{co} \texttt{div}$. If it is the latter, there is nothing to prove. In the former case, the proof follows by the previous point.

□

3.3 Discussion and Related Work

A language with ghost code. In this chapter, we presented and formalized KidML, a programming language of the ML family whose most distinguish feature is the interaction between ghost code and stateful traits, such as assignments and divergence. The presence of ghost code makes KidML suitable to be used as the programming language of a deductive verification framework. In the activity of applying deductive techniques to the verification of programs, we often need to introduce auxiliary elements, *e.g.*, functions or variables, with the purpose to make the proof effort much easier and sometimes, namely in an automated proof setting, to make it possible. This is exactly the role of ghost code. Ghost elements are removed by a code extraction mechanism, which generates a correct-by-construction, executable program.

The notion of ghost code dates back to the origins of deductive verification itself. The first acknowledged use of *auxiliary variables*, the most primitive form of ghost code, is due to Lucas in 1968 [103]. From that moment on, ghost code evolved into a standard component of deductive verification tools. We can cite VCC [38], Dafny [97], Viper [114], and Why3 itself as verification platforms that allow users to augment a program with some form of ghost code. The crucial property that all these tools statically guarantee is that ghost code does not interfere with regular data. This is also the case with KidML, with our type system featuring a clear separation between regular and ghost effects, as well as the global condition of admissible effects. In particular, we forbid assignments over regular locations when these are aliased with ghost ones.

A language with effects. KidML is a stateful language. It features memory assignment, raise and catch of exceptions, and divergence. The simultaneous use of imperative and ghost code is what makes the design of KidML type system a challenging task. In our design of such a type system, we chose to treat effects via oracle functions. This is mostly evident in two situations: when it comes to prove the termination of a certain expression using the `CheckTermination` predicate (we recall that the evaluation of some piece of ghost code must always terminates, *i.e.*, a ghost expression can never be assigned the divergence effect); in the characterization of admissible effects via the `adm` predicate, where we state that no assignment on a regular location is allowed when this is aliased with a ghost one, without giving a precise definition of what it means for two memory locations to be aliased.

A comprehensive treatment of effects in the WhyML language is given by Jean-Christophe Filliâtre, Léon Gondelman [72, Chap.3] and Andrei Paskevich [63]. The authors present a type and effect system with *regions*, an artifact that is used to statically track all the memory aliases occurring in a given program. The developed type system uses *singleton* regions and gives strong guarantees about the separation of function arguments and return value. Each type definition is annotated with a set of regions, one for each mutable component of such a type. In order to statically know all the existing aliases in the program, this approach cannot be applied to arbitrary pointer-based data structures, such as an union-find implementation[4], since the set of involved regions is not statically bound. In the design of KidML, we abstract away many of the details of this precise characterization of effects and memory aliases, since these are orthogonal to our intended use of the language. It would be interesting to study if the proposed approach can be applied to a language of the size of KidML.

Differences with respect to WhyML. The KidML language draws inspiration from WhyML. Features like ghost code, recursive function definitions, stateful computations, top-level definition of record types, and exceptions treatment can be found in both languages. In fact, KidML is very

[4] In Sec. 5.2, we describe an approach that allows us to prove a union-find data structure in the WhyML language.

close to the internal representation of a WhyML program. For instance, our separation in different syntactic categories of local variable binding and local (recursive) functions definitions can also be found in the internal representation of WhyML. There are, nevertheless, some WhyML interesting features that we did not include in the design of KidML. We detail some of main differences between KidML and WhyML in the following.

A key aspect in the practicality of functional programming languages is the presence of *pattern matching* [95, 105]. This is a mostly appreciated feature when it comes to manipulate recursively-defined data types, *e.g.*, algebraic data types. WhyML features top-level definition of algebraic data types, as well as a `match..with` construction to destruct values of such types [121]. We do not include pattern matching in KidML, as we believe that this would unnecessarily complicate the design and formalization of the language, without adding expressiveness power.

One can rely on the WhyML module system to break the program components into smaller individual units. These units can then be linked together using the Why3 data refinement mechanism. In Chap. 6, we provide several examples of programs implemented and proved using Why3, where a key component is the use of WhyML modules to separate a client code from the implementation details of a provided module. The KidML language does not feature any kind of separation on the program components, other than the local definition of functions. Equipping KidML with a module system and a module refinement mechanism, in the style of WhyML, would certainly approach KidML to a more realistic programming language.

Finally, an important aspect of WhyML is that it features a lightweight form of higher-order programming. One can pass functions as arguments of other functions, return functions as the result of some computation, and define anonymous functions, as long as all of those remain stateless functions. Additionally, as it is commonly the case in a higher-order setting, WhyML allows the user to partially apply some functional symbol, which generates a new function that can be manipulated, *e.g.*, to be passed as an argument to a higher-order traversal procedure. In KidML, every function application is total, *i.e.*, the programmer must always supply the exact number of attended arguments. To extend KidML with some form of higher-order programming, we would need to add functions as a possible result of evaluation, and we most certainly would have to reconsider the design of our type with effects system, in order to limit the interaction between higher-order programming and stateful code.

The finger pointing at the moon is not the moon.

Buddhist saying

4 Extraction

In this chapter, we design a code extraction procedure for KidML programs. The main task of extraction is the removal of any ghost element from the source code. Extraction is guided by the KidML type system, which guarantees that we only apply our extraction function to regular expressions, *i.e.*, expressions that are meaningful from an operational point of view. We present our extraction function in Sec. 4.1. One novelty of our extraction function, with respect to other existing similar presentations [64], is that we erase as much as possible, *i.e.*, without compromising the soundness of the extracted code, any trace of ghost code. This adds some difficulty in our presentation and formalization, but we believe this is worth the effort in order to obtain a more efficient extracted code.

An important aspect of an extracted program is that it preserves the behavior of the original source program. This holds both from a typing, as well as from a semantics point of view. The former means that a well-typed regular expression is extracted into a well-typed expression. In particular, the extracted expression is typed with the same regular effects and with an empty set of ghost effects. The latter means that if the original source code diverges, then the extracted code diverges as well; and if the original source code evaluates down to a semantics result, then the extracted expression evaluates down to the extraction of that result. In Sec. 4.2 and 4.3, we present the proofs of typing and semantics preservation, respectively.

The extraction procedure we present in this chapter is a representative part of the Why3 extraction mechanism, which we have implemented in the course of this thesis. In Sec. 4.4, we describe the implemented extraction machinery, with an highlight on the adopted architecture, which clearly separates code translation from the printing phase, and the use of the new Why3 `extract` command. Sec. 4.5 concludes this chapter with some related work and discussion. We detail on some of the main differences between the extracted function defined in this chapter and the one we actually implemented.

4.1 Extraction Function

In this section, we define our extraction function over KidML types, top-level declarations (exceptions and record-types), and finally over well-typed regular expressions. In the following, we denote our extraction function by $\mathcal{E}_{\overline{\beta}}(\cdot)$, meaning extraction is parameterized by a mask $\overline{\beta}$.

4.1.1 Extraction of Types

We begin by defining what it means to extract a type π, *i.e.*, a type of the form $\overline{\beta'\tau}$. The idea is to remove some of the type components, according to the mask passed as an argument to the extraction function.

Definition 4.1.1 (Extraction of π)**.** We define the extraction of type π by induction over its structure, as follows:

$\overline{\beta'} \sqsubseteq \overline{\beta}$ $\qquad$ $\boxed{\mathcal{E}_{\overline{\beta}}(\overline{\beta'\tau})}$

$$\mathcal{E}_{\overline{\beta}}(\overline{\beta'\tau}) \triangleq \begin{cases} \texttt{Unit} & \text{if } \forall i.\ \beta_i = \texttt{ghost} \\ \overline{\beta'\tau}_{\restriction\overline{\beta}} & \text{otherwise} \end{cases}$$

The pre-condition on the left ensures that we only apply extraction with a mask $\overline{\beta}$ of same length and that is *at least as ghost as* mask $\overline{\beta'}$. The notation $\overline{\beta'\tau}_{\restriction\overline{\beta}}$ means that we *filter* the elements of sequence $\overline{\beta'\tau}$ with respect to the regular elements of mask $\overline{\beta}$. In other words, we keep the $\beta'_i\tau_i$ pairs such that $\beta_i = \texttt{reg}$ and all the ghost components of type $\overline{\beta'\tau}$ are removed. For instance, the result of $\mathcal{E}_{(\texttt{ghost},\,\texttt{reg})}(\texttt{reg Int}, \texttt{reg Bool})$ is the type $\texttt{reg Bool}$. As we shall see later, this is useful when extracting a function body against the type declared in the function signature. We lift function $\mathcal{E}_{\overline{\beta}}(\cdot)$ to the level of σ types, as follows:

Definition 4.1.2 (Extraction of σ)**.**

$\overline{\beta'} \sqsubseteq \overline{\beta}$ $\qquad$ $\boxed{\mathcal{E}_{\overline{\beta}}(\overline{\beta'\tau}, \epsilon, \gamma)}$

$$\mathcal{E}_{\overline{\beta}}(\overline{\beta'\tau}, \epsilon, \gamma) \triangleq (\mathcal{E}_{\overline{\beta}}(\overline{\beta'\tau}), \epsilon, \emptyset)$$

The most important point about this definition is that the extracted type contains no ghost effects. This conforms to the fact that extracted expressions cannot contain any such effect, since every piece of ghost code is removed. We present, as well, the following auxiliary definition:

Definition 4.1.3 (Ghost status of σ)**.** We define the ghost status of a type σ as follows:

$\boxed{\mathcal{G}(\sigma)}$

$$\mathcal{G}(\overline{\beta\tau}, \epsilon, \gamma) \triangleq \begin{cases} \texttt{ghost} & \text{if } \forall i.\ \beta_i = \texttt{ghost} \wedge \epsilon = \emptyset \\ \texttt{reg} & \text{otherwise} \end{cases}$$

When defining extraction for KidML expressions, we will be using $\mathcal{G}(\cdot)$ to decide which sub-expressions we must extract. Following this definition, an *extractable* expression is one that is typed with a type σ such that $\mathcal{G}(\sigma) = \texttt{reg}$. An expression is thus extractable if its mask is not entirely ghost, or it is a stateful expression. Note that, for instance, an expression to which we assign type $(\texttt{Unit}, \emptyset, \emptyset)$ is considered a `ghost` expression, and so it can be completely removed.

4.1.2 Extraction of Top-level Declarations

KidML top-level declarations include the declaration of an exception name and arguments, as well as the declaration of new record types. We define the extraction of such KidML symbols, as follows:

Definition 4.1.4 (Extraction of top-level declarations)**.**

$$\mathcal{E}_{\overline{\beta}}(\texttt{type}\ \mathsf{T}\overline{\alpha} = \{\overline{\mathsf{f} : \beta'\tau}\}) \triangleq \texttt{type}\ \mathsf{T}\overline{\alpha} = \{\overline{\mathsf{f} : \beta'\tau}_{\restriction\overline{\beta'}}\}$$

$$\mathcal{E}_{\overline{\beta}}(\texttt{exception}\ \mathsf{E} : \overline{\pi}) \triangleq \texttt{exception}\ \mathsf{E} : \mathcal{E}(\overline{\pi})$$

When it comes to extract top-level declarations, the argument mask is meaningless. We keep it in the above definition just for completeness purposes. The extraction of a record-type declaration removes from the type definition all the ghost fields. The notation $\overline{\mathsf{f} : \beta'\tau}_{\overline{\beta'}}$ stands for the filter of the record fields with respect to mask $\overline{\beta}$. Let us consider, for instance, the following type declaration:

```
type c α = { x: reg Int; y: ghost α }
```
KidML

The result of extracting such a type is as follows:

```
type c α = { x: reg Int }
```
KidML

Note that we do not remove type variables from the type definition, even if these are no longer used after extraction. The unused variables types act as *ghost types* after extraction[1]. In the following, we always use notation $\restriction\overline{\beta}$ to refer to the filtering of some sequence with respect to mask $\overline{\beta}$. Finally, in the extraction of exception declarations, $\mathcal{E}(\overline{\pi})$ stands for the removal of all $\beta_i\tau_i$ pairs such that $\beta_i = \texttt{ghost}$. For instance, the exception declaration of page 44 is extracted to the following:

```
exception Return : reg Int
```
KidML

If all arguments of an exception E are declared ghost, then E has no arguments after extraction.

4.1.3 Extraction of Expressions

We define now our procedure to extract KidML expressions. We separate this definition into two distinct functions. First, we define the extraction of semantic results that satisfy the `abort` predicate, as follows:

Definition 4.1.5 (Extraction of `abort` results)**.** Let r be a semantic result such that `abort` r. We define the extraction of r as follows:

`abort` r $\qquad\boxed{\mathcal{E}_{\overline{\beta}}(e)}$

$$\mathcal{E}_{\overline{\beta}}(\texttt{absurd}) \triangleq \texttt{absurd}$$

$$\mathcal{E}_{\overline{\beta}}(\texttt{div}) \triangleq \texttt{div}$$

$$\mathcal{E}_{\overline{\beta}}(\texttt{raise}\ \mathsf{E}\bar{v}) \triangleq \texttt{raise}\ \mathsf{E}\bar{v}_{\restriction\mathcal{M}(\mathsf{E})}$$

The interesting aspect about the extraction of such expressions is that it completely ignores the argument mask. For the first two cases, the extraction function actually acts like the identity function. For the extraction of `raise` expressions, we filter the sequence $\bar{v}$ of arguments using $\mathcal{M}(\mathsf{E})$, the mask of exception E. Such a mask corresponds to the sequence of ghost status assigned to the arguments in the exception declaration. For instance, if we return to the previous example of the `Return` exception, result of extracting the KidML program

[1] It is an interesting aspect that ghost code is generating ghost types.

```
exception Return : reg Int, ghost Bool                    KidML

raise Return (42, false)
```

is the following one:

```
exception Return : reg Int                                KidML

raise Return 42
```

We can observe here the coherence in the number and type of arguments of the extracted exception declaration, and the extracted `raise` expression. The extraction of an exception applied to a sequence $\bar{a}$ of atomic variables follows the exact same approach:

$$\mathcal{E}_{\overline{\beta}}(\texttt{raise}\ E\,\bar{a}) \triangleq \texttt{raise}\ E\,\bar{a}_{\restriction \mathcal{M}(E)}$$

We now move to the definition of the extraction function over KidML expressions. The complete definition of this function is given in Fig. 4.1. The conditions on the top-left corner of the definition specify that this function can only be applied to well-typed regular KidML expressions. Moreover, the mask $\overline{\beta}$ given as an argument of extraction, and the mask $\overline{\beta'}$ assign by the KidML type system to the expression being extracted, must verify the property $\overline{\beta'} \sqsubseteq \overline{\beta}$. Throughout the definition of $\mathcal{E}_{\overline{\beta}}(e)$, we use notation $\mathcal{E}(e)$ to denote the extraction of expression e under its own mask. In the following, we elaborate on the extraction of each syntactic category.

Extraction of sequence of atomic expressions. Extracting a sequence $\bar{a}$ of atoms amounts to a filter of this sequence according to mask $\overline{\beta}$. Given the pre-condition of this function, $\overline{\beta}$ is at least as ghost as the mask of $\bar{a}$, hence every ghost variable of $\bar{a}$ does not occur in $\bar{a}_{\restriction \overline{\beta}}$. If we consider, for instance, the extraction of expression

```
(x, 42, x)                                                KidML
```

where `x` is a ghost variable, only two masks can be used: either $\overline{\beta} = \texttt{ghost, reg, ghost}$, in which case the result of extraction is the constant `42`; or $\overline{\beta} = \texttt{ghost}$, *i.e.*, $\overline{\beta}$ is a completely ghost mask, in which case the extracted expression corresponds to the unit value `()`.

Extraction of function definitions. When it comes to extract an expression of the form

$$\texttt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2$$

if `f` is a ghost function, the result of extraction is simply the extraction of expression e_2 under mask $\overline{\beta}$. On the other hand, if `f` is regular, we recursively extract the body e_1 with the mask given in type σ. We assume σ to be equal to $(\overline{\beta'\tau}, \epsilon, \gamma)$. We also extract the arguments and the return type σ. The following KidML program

```
fun f (x: reg Int) : (ghost Int, ∅, ∅) = x + 1 in          KidML
fun g (y: ghost Int) : ((ghost Int, reg Int), ∅, ∅) = (42, 73) in
e
```

gets extracted into

```
fun g () : (reg Int, ∅, ∅) = 73 in E_β̄(e)                  KidML
```

Every application of `f` inside expression `e` is typed as a ghost expression, by rule **(TApp)**. Consequently, there remain no occurrences of `f` in $\mathcal{E}_{\overline{\beta}}(e)$. Extraction of recursive definitions follows the exact same pattern.

$\Delta \cdot \Gamma \cdot \Sigma \vdash e : (\overline{\beta' \tau}, \epsilon, \gamma),\ \mathcal{G}(\overline{\beta'}, \epsilon) = \mathtt{reg},\ \overline{\beta'} \sqsubseteq \overline{\beta}$ $\quad \boxed{\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e)}$

$$\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(\bar{a}) \triangleq \bar{a}_{\restriction \overline{\beta}}$$

$$\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}\left(\begin{array}{l}\mathtt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x:\pi}) : \sigma = e_1 \\ \mathtt{in}\ e_2\end{array}\right) \triangleq \begin{cases} \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_2) & \text{if } \mathcal{G}(\sigma) = \mathtt{ghost} \\ \mathtt{fun}\ f\langle\overline{\alpha}\rangle\ (\mathcal{E}(\overline{x:\pi})) : \mathcal{E}(\overline{\beta'\tau}, \epsilon, \gamma) = \mathcal{E}^{\overline{\beta'}}_{\overline{\beta'}}(e_1) & \\ \mathtt{in}\ \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_2) & \text{otherwise} \end{cases}$$

$$\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}\left(\begin{array}{l}\mathtt{rec}\ f\langle\overline{\alpha}\rangle(\overline{x:\pi}) : \sigma = e_1 \\ \mathtt{in}\ e_2\end{array}\right) \triangleq \begin{cases} \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_2) & \text{if } \mathcal{G}(\sigma) = \mathtt{ghost} \\ \mathtt{rec}\ f\langle\overline{\alpha}\rangle\ (\mathcal{E}(\overline{x:\pi})) : \mathcal{E}(\overline{\beta'\tau}, \epsilon, \gamma) = \mathcal{E}^{\overline{\beta'}}_{\overline{\beta'}}(e_1) & \\ \mathtt{in}\ \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_2) & \text{otherwise} \end{cases}$$

$$\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(\mathtt{let}\ \overline{\beta' x} = e_1\ \mathtt{in}\ e_2) \triangleq \begin{cases} \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_2) & \text{if } \mathcal{G}(e_1) = \mathtt{ghost} \\ \mathcal{E}_{\mathtt{ghost}}(e_1) & \text{if } \mathcal{G}(e_2) = \mathtt{ghost} \\ \mathtt{let}\ \mathtt{reg}\ \_ = \mathcal{E}_{\mathtt{ghost}}(e_1)\ \mathtt{in}\ \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_2) & \text{if } \overline{\beta'} = \mathtt{ghost} \wedge \epsilon_1 \neq \emptyset \\ \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_2) & \text{if } \bar{x} \notin \mathrm{FV}(\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_2)) \wedge \epsilon_1 = \emptyset \\ \mathtt{let}\ \overline{\mathtt{reg}\ x}_{\restriction \overline{\beta'}} = \mathcal{E}^{\overline{\beta'}}_{\overline{\beta'}}(e_1)\ \mathtt{in}\ \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_2) & \text{otherwise} \end{cases}$$

$$\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(f\langle\overline{\tau}\rangle\ (\bar{a})) \triangleq \mathtt{let}\ \overline{\mathtt{reg}\ x} = f\langle\overline{\tau}\rangle\ (\bar{a}_{\restriction \overline{\beta'}})\ \mathtt{in}\ \bar{x}_{\restriction \overline{\beta}} \quad \text{if } \Delta(f) = (\overline{x : \beta'\tau}) \to \sigma$$

$$\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(\mathtt{loop}\ e) \triangleq \mathtt{loop}\ \mathcal{E}(e)$$

$$\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(\mathtt{if}\ a\ \mathtt{then}\ \mathtt{absurd}\ \mathtt{else}\ e_2) \triangleq \begin{cases} \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_2) & \text{if } \overline{\beta'} = \mathtt{ghost} \\ \mathtt{if}\ a\ \mathtt{then}\ \mathtt{absurd}\ \mathtt{else}\ \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_2) & \text{otherwise} \end{cases}$$

$$\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(\mathtt{if}\ a\ \mathtt{then}\ e_1\ \mathtt{else}\ \mathtt{absurd}) \triangleq \begin{cases} \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_1) & \text{if } \overline{\beta'} = \mathtt{ghost} \\ \mathtt{if}\ a\ \mathtt{then}\ \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_1)\ \mathtt{else}\ \mathtt{absurd} & \text{otherwise} \end{cases}$$

$$\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(\mathtt{if}\ a\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) \triangleq \mathtt{if}\ a\ \mathtt{then}\ \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_1)\ \mathtt{else}\ \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_2)$$

$$\mathcal{E}_{\mathtt{ghost}}(a.f) \triangleq ()$$

$$\mathcal{E}_{\mathtt{reg}}(a.f) \triangleq a.f$$

$$\mathcal{E}_{\mathtt{ghost}}(\{\overline{f = a}\}) \triangleq ()$$

$$\mathcal{E}_{\mathtt{reg}}(\{\overline{f = a}\}) \triangleq \{\overline{f = a}\}_{\restriction \mathcal{M}(\overline{f})}$$

$$\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(a_l.f \leftarrow a_r) \triangleq a_l.f \leftarrow a_r$$

$$\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(\mathtt{raise}\ E\bar{a}) \triangleq \mathtt{raise}\ E\bar{a}_{\restriction \mathcal{M}(E)}$$

$$\mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(\mathtt{try}\ e_0\ \mathtt{with}\ \overline{E\bar{x} \Rightarrow e}) \triangleq \mathtt{try}\ \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(e_0)\ \mathtt{with}\ \mathcal{E}^{\overline{\beta}}_{\overline{\beta}}(\overline{E\bar{x} \Rightarrow\ e})$$

Figure 4.1: Extraction of Expressions.

Extraction of local binding expressions. The extraction of `let..in` expressions demands a little more of attention. We distinguish five different possible cases when it comes to extract an expression of the form `let` $\overline{\beta' x} = e_1$ `in` e_2. First, if e_1 expression is completely ghost, we proceed with the extraction of expression e_2. For instance, the KidML expression

```
let ghost x = 0 in 42
```
KidML

is simply translated to the constant `42`.

The next case is triggered when e_2 is a ghost expression. We know that e_1 is a stateful expression, otherwise the whole `let..in` would be typed as ghost expression. In this case, the result of e_1 is meaningless, and so we extract this expression under a completely ghost mask. If we consider the following KidML program

```
let reg _ = a.c <- 42 in
ghost 0
```
KidML

where `c` is a regular field of some record type, it extracts to `a.c <- 42`.

The third case corresponds to the extraction of a `let..in` expression where e_2 is a regular expression and e_1 is a stateful expression, even if it is assigned a completely ghost mask. In such a case, we erase the results of e_1, by extracting it with a completely ghost mask, and then extract e_2 as expected. As an example, the following KidML program

```
let ghost x =
  let reg _ = a.c <- 42 in 0 in
73
```
KidML

where `a` is a regular expression, and `c` a regular field, gets extracted into

```
let reg _ = a.c <- 42 in
73
```
KidML

This highlights a subtle point of the KidML language and your extraction procedure: even if we declare `x` as ghost variable, it does not necessarily mean that such a variable is bound to a ghost expression. This differs from the following expression

```
let ghost x = ghost e
```
KidML

in the sense that now the whole expression e is typed as a ghost expression. In particular, there can be no regular effects in e and so the program

```
let ghost x =
  ghost (let reg _ = a.c <- 42 in 0) in
73
```
KidML

would be rejected by the KidML type system.

The case in which e_2 is a regular expression, e_1 is stateless expression, and the sequence $\bar{x}$ of locally bound variables does not occur in the result of extracting e_2, is an optimization of our extraction function. Let us consider the following KidML program:

```
fun f () : ((ghost Int, reg Int), ∅, ∅) =
  let reg x = ... (* terminating, but very long computation *) in
  (x, 0) in
f ()
```
KidML

We could extract such a program in the following one:

```
fun f () : (reg Int, ∅, ∅) =
  let reg x = ... (* terminating, but very long computation *) in
```
KidML

```
  0 in
f ()
```

which is a well-typed KidML program. However, we would be paying the price of executing a long computation whose value is only used within ghost code. In order to avoid such a situation, we extract the original program into

```
fun f () : (reg Int, ∅, ∅) = 0 in                                    KidML
f ()
```

The final case is, perhaps, the one that happens most commonly in practice. In this case, both e_1 and e_2 are regular expressions. We proceed by extracting e_2 with the argument mask of extraction, and e_1 with the mask assigned in the return type of `f`. The sequence of variables $\bar{x}$ is also filtered according to this mask. As an example of such case, let us consider the following KidML program:

```
fun f () : ((reg Int, ghost Int), ∅, ∅) =                            KidML
  let reg x, ghost y = 42, 0 in (x, y) in
f ()
```

This is extracted into the following one:

```
let f () : (reg Int, ∅, ∅) =                                         KidML
  let reg x = 42 in x in
f ()
```

Extraction of applications. The extraction rule of applications, although somewhat cryptic, is very easily explained using an example. Let us consider, for instance, the following KidML program:

```
fun f (y: ghost Int, x: reg Int) : ((reg Int, reg Int), ∅, ∅) = (x, x) in   KidML
fun g (z: reg Int) : ((ghost Int, reg Int), ∅, ∅) = f (z, 42) in ...
```

Extracting the definition of function `f` is not particularly challenging:

```
fun f (x: reg Int) : ((reg Int, reg Int), ∅, ∅) = (x, x)             KidML
```

The signature of function `g` is also straightforwardly translated into

```
let g (z: reg Int) : (reg Int, ∅, ∅)                                 KidML
```

When it comes to extract the definition of `g`, we need to extract the application `f (z, 42)` under the mask assigned in the return type of `g`. In order to take this mask into account, we locally bind the regular results of the application, and then filter the sequence of bound variables using the mask of the function return type. The extraction of function `g` is as follows:

```
let g (z: reg Int) : (reg Int, ∅, ∅) =                               KidML
  let reg x, reg y = f (42) in y
```

Extraction of loops. The extraction of a `loop e` expression is straightforward. We simply recursively extract `e`, the body of the loop, with its own mask. Since `e` is always typed as a `Unit` expression, this is actually the empty mask.

Extraction of conditional expressions. The extraction of `if..then..else` expressions follows a very simple approach, where sub-expressions are recursively extracted using the mask that is initially given as an argument. The interesting aspect of this rule is the case when one of the

branches is `absurd` and the test is a ghost expression. In that case, and according to typing rules (**TIfAbsurd1**) and (**TIfAbsurd2**), we do not need to keep the conditional structure in the extracted expression.

Extraction of access, creation, and assignment of record fields. The extraction of record-related expressions is straightforward. In the case of access and creation expressions, we can only a apply either a completely (singleton) ghost mask or a regular one. In the first case, both expressions are translated into the unit value (). On the other hand, extraction function just behaves as the identity function for access, and filters the pairs $f_i = a_i$ according to the mask $\mathcal{M}(\bar{f})$. This mask stands for the mask that is assigned to the fields in the declaration of the associated record type. For instance, the following KidML program

```
type t = { a: ghost Int; b: reg Int; c: ghost Int }

{ a = 2; b = 3; c = 5 }
```
KidML

is translated into

```
type t = { b: reg int }

{ b = 3 }
```
KidML

As we did for exceptions, we insist on the coherence between the extracted declaration of type `t` and the expression that creates a value of such type.

Given the pre-condition on the top-left corner of Fig. 4.1, we know we can only extract regular expressions. When it comes to extract assignment expressions, we know this can only be a regular assignment, which we must keep in the extracted program. Hence, the extraction function behaves as the identity function for assignments.

Extraction of exception raising. The extraction of a `raise` expression, where the exception is applied to a sequence $\bar{a}$ of atomic variables, is similar to the extraction of a semantics result of the form `raise` $E\bar{v}$. It ignores mask $\overline{\beta}$ and filters the sequence $\bar{a}$ according to the mask of exception E.

Extraction of exception handling. The last case in our definition of the extraction function concerns the extraction of `try..with` expressions. We first recursively extract the body of the handler, expression e_0, with mask $\overline{\beta}$. Next, we must extract each branch of the handler individually. Naively, we could simply apply the extraction function over each expression e_i. However, some of these might very well be ghost expressions. Let us take a moment in order to clarify such a possibility. Consider, for instance, the following KidML program:

```
exception E : ghost Int
type ref α = { c: α }

fun f (r: reg (ref Int)) : (ghost Int, ∅+writes r.c, ∅) =
  try r.c <- 42; 89
  with E x => x in
f ({ c = 0 })
```
KidML

Even if the only branch of this handler is a ghost expressions, rule (**TTry**) assigns a regular type to the whole `try..with` expression. This is due to the assignment `r.c <- 42`, a regular effect that we must preserve in the extracted program. To extract each branch of an exception handler, we proceed as follows. If the branch is a completely ghost expression, we replace it with the unit value,

$$\mathcal{G}(e_i) = \texttt{ghost} \qquad \mathcal{E}_{\overline{\beta}}(E\bar{x} \Rightarrow e_i) \triangleq E\bar{x}_{\restriction \mathcal{M}(E)} \Rightarrow ()$$

where $\mathcal{G}(e_i) = \texttt{ghost}$ is a notation shortcut which we use to state expression e_i is a ghost expression. Let us note that, even if e_i is a ghost expression, we must preserve the branch structure to catch exception E_i. Otherwise, if expression e_0 raises E_i, the extracted code would be introducing a regular effect that did not existed in the source code.

If e_i is a regular expression, we straightforwardly extract it using mask $\overline{\beta}$, as follows:

$$\mathcal{G}(e_i) = \texttt{reg} \qquad \mathcal{E}_{\overline{\beta}}(E\bar{x} \Rightarrow e_i) \triangleq E\bar{x}_{\restriction \mathcal{M}(E)} \Rightarrow \mathcal{E}_{\overline{\beta}}(e_i)$$

The result of extracting the above KidML program is as follows:

KidML

```
exception E
type ref α = { c: α }

fun f (r: reg (ref Int)) : (ghost Int, ∅+writes r.c, ∅) =
  try r.c <- 42
  with E -> () in
f ({ c = 0 })
```

Recursive calls to the extraction function respect pre-conditions. We complete our presentation of the extraction function by showing that, for every recursive call in the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$, we respect the pre-conditions given in the top-left corner of Fig. 4.1.

Lemma 4.1.1 (Extraction function respects pre-conditions)**.** *In the definition of Figure 4.1, for each recursive call $\mathcal{E}_{\overline{\beta}}(e)$ we have that $\Delta \cdot \Gamma \cdot \Sigma \vdash e : (\overline{\beta'\tau}, \epsilon, \gamma)$ s.t. $\mathcal{G}(\overline{\beta'}, \epsilon) = \texttt{reg}$ and $\overline{\beta'} \sqsubseteq \overline{\beta}$.*

Proof. Supposing the pre-conditions holds at entry, by case analysis on the structure of e, following the definition of the extraction function.

cases $(e \equiv \{\overline{f = a}\})$, $(e \equiv a.f)$, $(e \equiv \bar{a})$, $(e \equiv \texttt{absurd})$, $(e \equiv f(\bar{a}))$, $(e \equiv a_r.f \leftarrow a)$, and $(e \equiv \texttt{raise}\, E\bar{a})$. Nothing to prove.

case $(e \equiv \texttt{fun}\ f(\overline{x : \pi}) : (\overline{\beta'\tau}, \epsilon) = e_1\ \texttt{in}\ e_2)$. The typing derivation ends up with

$$\frac{\begin{array}{c}\Delta \cdot \Gamma + [\overline{x : \pi}] \cdot \Sigma \vdash e_1 : (\overline{\beta_1\tau}, \epsilon_1', \gamma_1') \qquad (\overline{\beta_1\tau}, \epsilon_1', \gamma_1')_{\restriction \Gamma + [\overline{x:\pi}]} \sqsubseteq (\overline{\beta_0\tau}, \epsilon_1, \gamma_1) \\ \Delta + [f : \forall\overline{\alpha}.(\overline{x : \pi}) \to (\overline{\beta_0\tau}, \epsilon_1, \gamma_1)] \cdot \Gamma \cdot \Sigma \vdash e_2 : (\overline{\beta'\tau}, \epsilon, \gamma) \qquad \overline{\alpha}\#\Gamma \qquad \overline{\alpha}\#\Delta\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : (\overline{\beta_0\tau}, \epsilon_1, \gamma_1) = e_1\ \texttt{in}\ e_2 : (\overline{\beta'\tau}, \epsilon, \gamma)}$$

We distinguish two sub-cases:

- sub-case $\mathcal{G}(\overline{\beta_0}, \epsilon_1) = \texttt{ghost}$: by hypotheses we have $\mathcal{G}(\overline{\beta'}, \epsilon) = \texttt{reg}$ and $\overline{\beta'} \sqsubseteq \overline{\beta}$, so the recursive call $\mathcal{E}_{\overline{\beta}}(e_2)$ respects the pre-conditions.

- sub-case $\mathcal{G}(\overline{\beta_0}, \epsilon_1) = \texttt{reg}$: as for the previous case, it is easy to conclude that the recursive call to $\mathcal{E}_{\overline{\beta_0}}(e_1)$ respects the pre-conditions.

  We have $(\overline{\beta_1\tau}, \epsilon_1', \gamma_1') \sqsubseteq (\overline{\beta_0\tau}, \epsilon_1, \gamma_1)$, in particular $\overline{\beta_1} \sqsubseteq \overline{\beta_0}$. We can thus conclude that the recursive call $\mathcal{E}_{\overline{\beta_0}}(e_1)$ respects the pre-conditions.

case $(e \equiv \texttt{rec}\ f(\overline{x : \pi}) : (\overline{\beta' \tau}, \epsilon, \gamma) = e_1\ \texttt{in}\ e_2)$**.** Similar to the previous case.

case $(e \equiv \texttt{let}\ \overline{\beta' x} = e_1\ \texttt{in}\ e_2)$**.** The typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_1 : (\overline{\beta_1 \tau_1},\ \epsilon_1,\ \gamma_1) \qquad \overline{\beta_1} \sqsubseteq \overline{\beta'} \qquad \Delta \cdot \Gamma + [\overline{x : \beta' \tau_1}] \cdot \Sigma \vdash e_2 : (\overline{\pi_2},\ \epsilon_2,\ \gamma_2)}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{let}\ \overline{\beta' x} = e_1\ \texttt{in}\ e_2 : (\overline{\pi_2},\ \epsilon_1 \cup \epsilon_2,\ \gamma_1 \cup \gamma_2)}$$

We distinguish five sub-cases:

- sub-case $\mathcal{G}(\overline{\beta_1}, \epsilon_1) = \texttt{ghost}$: by hypotheses, we have $\mathcal{G}(\overline{\pi_2}, \epsilon_2) = \texttt{reg}$ and so the call to $\mathcal{E}_{\overline{\beta}}(e_2)$ respects the pre-conditions.
- sub-case $\mathcal{G}(\overline{\pi_2}, \epsilon_2) = \texttt{ghost}$: expression e_1 must be an effectful expression, $\mathcal{G}(\overline{\beta_1}, \epsilon_1) = \texttt{reg}$. We trivially have that $\overline{\beta_1} \sqsubseteq \texttt{ghost}$, so the call $\mathcal{E}_{\texttt{ghost}}(e_1)$ respects the pre-conditions.
- sub-case $\overline{\beta'} = \texttt{ghost}$ and $\texttt{obs}(\epsilon_1)$: by hypotheses we have $\mathcal{G}(\overline{\pi_2}, \epsilon_2) = \texttt{reg}$. Expression e_1 is an effectful expression, thus the call $\mathcal{E}_{\texttt{ghost}}(e_1)$ respects the pre-conditions. Expression e_2 is a regular expression, and so the call $\mathcal{E}_{\overline{\beta}}(e_2)$ respects the pre-conditions.
- sub-case $\bar{x} \notin \mathrm{FV}(\mathcal{E}_{\overline{\beta}}(e_2))$ and $\neg\texttt{obs}(e_1)$: similar to the case $\mathcal{G}(\overline{\beta_1}, \epsilon_1) = \texttt{ghost}$.
- otherwise, we have $\mathcal{G}(\overline{\beta_1}, \epsilon_1) = \texttt{reg}$ and $\mathcal{G}(\overline{\pi_2}, \epsilon_2) = \texttt{reg}$. Since $\overline{\beta_1} \sqsubseteq \overline{\beta'}$, we conclude that the call $\mathcal{E}_{\overline{\beta'}}(e_1)$ respects the pre-conditions. We can also easily conclude that the call $\mathcal{E}_{\overline{\beta}}(e_2)$ respects the pre-conditions.

case $(e \equiv \texttt{loop}\, e')$ We distinguish two sub-cases according to the typing rule used at the bottom of the derivation:

- case **(TLoop)**: the typing derivation ends up with

$$\frac{\begin{array}{c}\texttt{CheckTermination}(\texttt{loop}\, e) \\ \Delta \cdot \Gamma \cdot \Sigma \vdash e' : (\texttt{Unit},\ \epsilon,\ \gamma)\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{loop}\, e' : (\overline{\texttt{reg}\, \tau},\ \epsilon,\ \gamma)}$$

  The call $\mathcal{E}(e')$ trivially respects the pre-conditions, since e' is a unit expression and we extract e' with its own mask.

- case **(TLoopDiv)**: similar to the previous case.

case $(e \equiv \texttt{if}\ a\ \texttt{then absurd else}\ e_2)$**.** We distinguish two sub-cases according to the typing rule used at the bottom of the derivation:

- case **(TIfAbsurd1)**: the typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash a : (\beta' \texttt{Bool}, \emptyset, \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_2 : (\overline{\beta_2 \tau, \epsilon_2, \gamma_2})}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{if}\ a\ \texttt{then absurd else}\ e_2 : (\overline{\beta_2 \tau}, \epsilon_2, \gamma_2)}$$

  where $\overline{\beta_2} \sqsubseteq \overline{\beta}$ and $\mathcal{G}(\overline{\beta_2}, \epsilon_2) = \texttt{reg}$. The recursive call $\mathcal{E}_{\overline{\beta}}(e_2)$ thus respects the pre-conditions.

- case **(TIf)**: typing derivation ends up with

$$\frac{\begin{array}{c}\Delta \cdot \Gamma \cdot \Sigma \vdash a : (\texttt{reg}\, \texttt{Bool},\ \emptyset,\ \emptyset) \\ \Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{absurd} : (\texttt{reg}\tau, \emptyset, \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_2 : (\overline{\beta_2 \tau_2}, \epsilon_2, \gamma_2)\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{if}\ a\ \texttt{then absurd else}\ e_2 : (\overline{\beta_2 \tau_2}, \epsilon_2, \gamma_2)}$$

  The call $\mathcal{E}(\texttt{absurd})$ trivially respects the pre-conditions. By hypotheses $\overline{\beta_2} \sqsubseteq \overline{\beta}$ and $\mathcal{G}(\overline{\beta_2}, \epsilon_2) = \texttt{reg}$ so the call $\mathcal{E}_{\overline{\beta}}(e_2)$ respects the pre-conditions.

case ($e \equiv \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ \texttt{absurd}$)**.** Similar to the previous one.

case ($e \equiv \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2$)**.** The typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash a : (\texttt{reg}\ \texttt{Bool}, \emptyset, \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_1 : (\overline{\beta_1 \tau}, \epsilon_1, \gamma_1) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_2 : (\overline{\beta_2 \tau}, \epsilon_2, \gamma_2)}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : (\overline{(\beta_1 \sqcup \beta_2)\tau}, \epsilon_1 \cup \epsilon_2, \gamma_1 \cup \gamma_2)}$$

where $\mathcal{G}(\overline{\beta_1 \sqcup \beta_2}, \epsilon_1 \cup \epsilon_2) = \texttt{reg}$ and $\overline{\beta_1 \sqcup \beta_2} \sqsubseteq \overline{\beta}$. In particular,

$$\mathcal{G}(\overline{\beta_1}, \epsilon_1) = \texttt{reg} \qquad \mathcal{G}(\overline{\beta_2}, \epsilon_2) = \texttt{reg} \qquad \overline{\beta_1} \sqsubseteq \overline{\beta} \qquad \overline{\beta_2} \sqsubseteq \overline{\beta}$$

The recursive calls $\mathcal{E}_{\overline{\beta}}(e_1)$ and $\mathcal{E}_{\overline{\beta}}(e_2)$ thus respect the pre-conditions. The call $\mathcal{E}(a)$ also respects the pre-conditions since a is a regular expression and we use its own mask to extract it.

case ($e \equiv \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e'}$)**.** The typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_0 : (\overline{\pi},\ \epsilon,\ \gamma) \qquad \forall i.\ E_i : \overline{\pi_i} \qquad \forall i.\ \Delta \cdot \Gamma + [\bar{x} : \overline{\pi_i}] \cdot \Sigma \vdash e_i : \sigma_i}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} : (\overline{\pi},\ \epsilon - \overline{\texttt{raises}\ E_i},\ \gamma - \overline{\texttt{raises}\ E_i}) \cup \bigcup_i \sigma_i}$$

where $\mathcal{G}(\overline{\beta_0}, \epsilon) = \texttt{reg}$, $\overline{\beta_0} \sqsubseteq \overline{\beta}$, and $\forall i. \overline{\beta_i} \sqsubseteq \overline{\beta}$. It is easy to conclude that the call $\mathcal{E}_{\overline{\beta}}(e_0)$ respects the pre-conditions. The call $\mathcal{E}_{\overline{\beta}}(\overline{E\bar{x} \Rightarrow e})$ performs an individual extraction of the branches. For each ghost branch, we do not call the extraction function, so there is no violation of the pre-conditions. On the other hand, for each regular branch e_i, we have $\mathcal{G}(\overline{\beta_i}, \epsilon_1) = \texttt{reg}$ and so the call $\mathcal{E}_{\overline{\beta}}(e_i)$ respects the pre-conditions.

□

4.2 Typing Preservation under Extraction

In this section, we prove that our extraction function produces only well-typed, regular KidML expressions. Before demonstrating such a theorem, let us introduce some auxiliary properties about the masks comparison operator $\sqsubseteq$ and the masks union operator $\sqcup$. First, we show that $\sqsubseteq$ defines a preorder, as established by the following property:

Property 4.2.1 (Relation $\sqsubseteq$ is a preorder)**.**

1. $\beta \sqsubseteq \beta$.
2. *if* $\beta_1 \sqsubseteq \beta_2$ *and* $\beta_2 \sqsubseteq \beta_3$ *, then* $\beta_1 \sqsubseteq \beta_3$.

Proof.

1. by the definition of $\sqsubseteq$.
2. by case analysis on β_3.

□

We prove, as well, that $\sqcup$ is an Associative, Commutative, and Idempotent operator:

Property 4.2.2 (Operator $\sqcup$ is ACI)**.**

1. $(\beta_1 \sqcup \beta_2) \sqcup \beta_3 = \beta_1 \sqcup (\beta_2 \sqcup \beta_3)$.

2. $\beta_1 \sqcup \beta_2 = \beta_2 \sqcup \beta_1$.

3. $\beta \sqcup \beta = \beta$.

Proof.

1. by case analysis on β_1.
2. by case analysis on β_1 and sub-cases on β_2.
3. by the definition of $\sqcup$.

□

The following lemma establishes that for all masks β and β' such that $\beta \sqsubseteq \beta'$, then the result of $\beta \sqcup \beta'$ is always equal to the β' mask:

Lemma 4.2.3. *If* $\beta \sqsubseteq \beta'$ *then* $\beta \sqcup \beta' = \beta'$.

Proof. By case analysis on β' and using the definitions of $\sqsubseteq$ and $\sqcup$. □

Finally, we show that the order relation $\sqsubseteq$ is stable under the $\sqcup$ operator:

Lemma 4.2.4 ($\sqsubseteq$ is stable under $\sqcup$)**.** *If* $\beta' \sqsubseteq \beta''$ *then* $\forall \beta.\ \beta' \sqcup \beta \sqsubseteq \beta'' \sqcup \beta$.

Proof. By case analysis on β.

- case $\beta \sqsubseteq \beta'$: $\beta \sqsubseteq \beta''$ holds (by the transitivity of $\sqsubseteq$) and so $\beta' \sqcup \beta \sqsubseteq \beta'' \sqcup \beta \Leftrightarrow \beta' \sqsubseteq \beta''$ (by auxiliary lemma 4.2.3).
- case $\beta' \sqsubseteq \beta \sqsubseteq \beta''$: $\beta' \sqcup \beta \sqsubseteq \beta'' \sqcup \beta \Leftrightarrow \beta \sqsubseteq \beta''$ (by auxiliary lemma 4.2.3).
- case $\beta'' \sqsubseteq \beta$: $\beta' \sqcup \beta \sqsubseteq \beta'' \sqcup \beta \Leftrightarrow \beta \sqsubseteq \beta$ (by auxiliary lemma 4.2.3), where the last relation holds by the reflexive property of $\sqsubseteq$.

□

Lemma 4.2.5 (Extraction preserves mask union)**.** $\mathcal{E}_\beta(\beta_1 \sqcup \beta_2) = \mathcal{E}_\beta(\beta_1) \sqcup \mathcal{E}_\beta(\beta_2)$.

Proof. By case analysis.

- $\beta = \texttt{ghost}$: $\mathcal{E}_{\texttt{ghost}}(\beta_1 \sqcup \beta_2) = \texttt{ghost} = (\texttt{ghost} \sqcup \beta_1) \sqcup (\texttt{ghost} \sqcup \beta_2)$
- $\beta = \texttt{reg}$: $\mathcal{E}_{\texttt{reg}}(\beta_1 \sqcup \beta_2) = \beta_1 \sqcup \beta_2 = (\texttt{reg} \sqcup \beta_1) \sqcup (\texttt{reg} \sqcup \beta_2)$

□

Corollary 4.2.5.1. $\mathcal{E}_{\overline{\beta}}(\overline{\beta}_1 \sqcup \overline{\beta}_2) = \mathcal{E}_{\overline{\beta}}(\overline{\beta}_1) \sqcup \mathcal{E}_{\overline{\beta}}(\overline{\beta}_2)$

Proof. By induction on the structure of β and then by case analysis.

- case $\overline{\beta}$ is a singleton mask: by lemma 4.2.5.
- case $\overline{\beta} = \beta_1\beta_2 \ldots \beta_n$, hence β_1 and β_2 are also singleton masks: by IH we have

$$\mathcal{E}_{\beta_2 \ldots \beta_n}(\beta_1^2 \ldots \beta_1^n \sqcup \beta_2^2 \ldots \beta_2^n) = \mathcal{E}_{\beta_2 \ldots \beta_n}(\beta_1^2 \ldots \beta_1^n) \sqcup \mathcal{E}_{\beta_2 \ldots \beta_n}(\beta_2^2 \ldots \beta_2^n)$$

  - sub-case $\beta_1 = \texttt{ghost}$: we conclude using the IH.

– sub-case $\beta_1 = \texttt{reg}$: since $\overline{\beta} \sqsubseteq \overline{\beta}_1 \sqcup \overline{\beta}_2$ then $\beta_1^1 \sqsubseteq \beta_1$ and $\beta_2^1 \sqsubseteq \beta_1$. This gives $\beta_1^1 = \beta_2^1 = \beta_1 = \texttt{reg}$. The result of $\mathcal{E}_{\beta_1}(\beta_1^1 \sqcup \beta_2^1)$ is then $\texttt{reg}$ and using the IH we build the expected result:

$$\texttt{reg}\mathcal{E}_{\beta_2 \ldots \beta_n}(\mathcal{E}_{\beta_2 \ldots \beta_n}(\beta_1^2 \ldots \beta_1^n) \sqcup \mathcal{E}_{\beta_2 \ldots \beta_n}(\beta_2^2 \ldots \beta_2^n))$$

□

To state the preservation theorem, we need the following auxiliary definitions:

Definition 4.2.1 (Typing context extraction)**.**

$\boxed{\mathcal{E}(\Gamma)}$

$$\begin{aligned} \mathcal{E}(\emptyset) &\triangleq \emptyset \\ \mathcal{E}(\Gamma + [x : \texttt{ghost}\ \tau]) &\triangleq \mathcal{E}(\Gamma) \\ \mathcal{E}(\Gamma + [x : \texttt{reg}\ \tau]) &\triangleq \mathcal{E}(\Gamma) + [x : \texttt{reg}\ \tau] \end{aligned}$$

Definition 4.2.2 (Typing functions context extraction)**.**

$\boxed{\mathcal{E}(\Delta)}$

$$\begin{aligned} \mathcal{E}(\emptyset) &\triangleq \emptyset \\ \mathcal{E}(\Delta + [f : \forall \overline{\alpha}.(\overline{x : \pi}) \to \sigma]) &\triangleq \begin{cases} \mathcal{E}(\Delta) & \text{if } \mathcal{G}(\sigma) = \texttt{ghost} \\ \mathcal{E}(\Delta) + [f : \forall \overline{\alpha}.(\mathcal{E}(\overline{x : \pi})) \to \mathcal{E}(\sigma)] & \text{otherwise} \end{cases} \end{aligned}$$

We do not need to define an extraction for a typing store context Σ. Since every location is assigned a regular type by our typing rules, such a function would just behave like the identity function. We finally state type preservation theorem as follows:

Theorem 4.2.6 (Type preservation under extraction)**.** *If* $\Delta \cdot \Gamma \cdot \Sigma \vdash e : (\overline{\beta' \tau}, \epsilon, \gamma)$*, where* $\mathcal{G}(\overline{\beta' \tau}, \epsilon) = \texttt{reg}$*, then for all* $\overline{\beta}$ *such that* $\overline{\beta'} \sqsubseteq \overline{\beta}$*,* $\mathcal{E}(\Delta) \cdot \mathcal{E}(\Gamma) \cdot \Sigma \vdash \mathcal{E}_{\overline{\beta}}(e) : \mathcal{E}_{\overline{\beta}}(\overline{\beta' \tau}, \epsilon, \gamma)$ *holds.*

Proof. Induction on the derivation $\Delta \cdot \Gamma \cdot \Sigma \vdash e : (\overline{\beta \tau'}, \epsilon, \gamma)$. Throughout this proof we shall write $\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma)$ instead of $\mathcal{E}(\Delta) \cdot \mathcal{E}(\Gamma) \cdot \Sigma$.

case (TAbar). By IH

$$\forall i.\ \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\beta_i}(a_i) : (\beta_i' \tau_i, \emptyset, \emptyset)$$

for each a_i s.t. $\mathcal{G}(a_i) = \texttt{reg}$ where $\beta_i' \sqsubseteq \beta_i$. In particular, any ghost variable or any ghost location in $\bar{a}$ is also removed in $\mathcal{E}(\Gamma)$ and $\mathcal{E}(\Sigma)$, respectively. By taking only the judgments for which $\beta_i = \texttt{reg}$ we get exactly the subset $\bar{a}_{\restriction \overline{\beta}}$. The following judgment is then a valid judgment:

$$\frac{\forall i.\ \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash a_i : (\texttt{reg}\ \tau_i, \emptyset, \emptyset)}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \bar{a}_{\restriction \overline{\beta}} : \mathcal{E}_{\overline{\beta}}(\overline{\texttt{reg}\ \tau}, \emptyset, \emptyset)}$$

case (TFun). The typing derivation ends up with

$$\frac{\Delta \cdot \Gamma + [\overline{x : \beta' \tau}] \cdot \Sigma \vdash e_1 : \sigma' \qquad \sigma'_{\restriction \Gamma + [\overline{x:pi}]} \sqsubseteq \sigma \qquad \Delta + [f : \forall \overline{\alpha}.(\overline{x : \beta' \tau}) \to \sigma] \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma'' \qquad \overline{\alpha} \notin \Gamma \qquad \overline{\alpha} \notin \Delta}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{fun}\ f\langle \overline{\alpha} \rangle\ (\overline{x : \beta' \tau}) : \sigma = e_1\ \texttt{in}\ e_2 : \sigma''}$$

We distinguish two sub-cases:

- sub-case $\mathcal{G}(\sigma) = \texttt{ghost}$: by the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$, we have

  $$\mathcal{E}_{\overline{\beta}}(\texttt{fun}\ f\langle \overline{\alpha} \rangle\ (\overline{x : \beta' \tau}) : \sigma = e_1\ \texttt{in}\ e_2) \triangleq \mathcal{E}_{\overline{\beta}}(e_2)$$

  By IH

  $$\mathcal{E}(\Delta + [f : (\overline{x : \beta' \tau}, \sigma)]) \cdot \mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\sigma'')$$

  which simplifies to

  $$\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\sigma'')$$

- sub-case $\mathcal{G}(\sigma) = \texttt{reg}$: by the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$ we have

  $$\mathcal{E}_{\overline{\beta}}(\texttt{fun}\ f\langle \overline{\alpha} \rangle (\overline{x : \beta' \tau}) : (\overline{\beta'' \tau}, \epsilon) = e_1 \texttt{in} e_2) \triangleq \texttt{fun}\ f\langle \overline{\alpha} \rangle \mathcal{E}(\overline{x : \beta' \tau}) : \mathcal{E}(\overline{\beta'' \tau}, \epsilon) = \mathcal{E}_{\overline{\beta''}}(e_1) \texttt{in} \mathcal{E}_{\overline{\beta}}(e_2)$$

  By IH

  $$\mathcal{E}(\Delta) \cdot \mathcal{E}(\Gamma + [\overline{x : \beta' \tau}]) \cdot \mathcal{E}(\Sigma) \vdash \mathcal{E}_{\overline{\beta''}}(e_1) : \mathcal{E}_{\overline{\beta''}}(\sigma')$$
  $$\mathcal{E}(\Delta + [f : \forall \overline{\alpha}.(\overline{x : \beta' \tau}) \to (\overline{\beta'' \tau},\ \epsilon,\ \gamma)]) \cdot \mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\sigma'')$$

  By auxiliary lemma 4.2.4 $\mathcal{E}_{\overline{\beta''}}(\sigma') \sqsubseteq \mathcal{E}_{\overline{\beta''}}(\sigma)$, so the following judgment is valid:

  $$\frac{\mathcal{E}(\Delta) \cdot \mathcal{E}(\Gamma) + [\mathcal{E}(\overline{x : \beta' \tau})] \cdot \mathcal{E}(\Sigma) \vdash \mathcal{E}_{\overline{\beta''}}(e_1) : \mathcal{E}_{\overline{\beta''}}(\sigma') \qquad \mathcal{E}(\Delta) + [f : \forall \overline{\alpha}.\mathcal{E}(\overline{x : \beta' \tau}) \to \mathcal{E}(\overline{\beta'' \tau},\ \epsilon,\ \gamma)] \cdot \mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\sigma'') \qquad \mathcal{E}_{\overline{\beta''}}(\sigma') \sqsubseteq \mathcal{E}(\overline{\beta'' \tau}, \epsilon) \qquad \overline{\alpha} \notin \mathcal{E}(\Gamma) \qquad \overline{\alpha} \notin \mathcal{E}(\Delta)}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \texttt{fun}\ f\langle \overline{\alpha} \rangle\ \mathcal{E}(\overline{x : \beta' \tau}) : \mathcal{E}(\overline{\beta'' \tau}, \epsilon) = \mathcal{E}_{\overline{\beta''}}(e_1)\ \texttt{in}\ \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\sigma'')}$$

cases (TRec) and (TRecDiv). Similar to the (TFun) case.

case (TApp). The typing derivation ends up with

$$\frac{\theta = [\overline{\alpha} \mapsto \overline{\tau}] \qquad \rho = [\overline{x} \mapsto \overline{a}] \qquad \Delta(f) = \forall \overline{\alpha}.(\overline{x : \beta' \tau}) \to (\overline{\beta_\sigma \tau_\sigma},\ \epsilon,\ \gamma) \qquad \forall i.\ \Delta \cdot \Gamma \cdot \Sigma \vdash a_i : (\beta_i'' \tau_i \theta,\ \emptyset,\ \emptyset) \qquad \forall i.\ \beta_i'' \sqsubseteq \beta_i'}{\Delta \cdot \Gamma \cdot \Sigma \vdash f\langle \overline{\tau} \rangle(\overline{a}) : (\overline{\beta_\sigma \tau_\sigma},\ \epsilon,\ \gamma)\theta\rho}$$

By the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$ we have

$$\mathcal{E}_{\overline{\beta}}(f\langle \overline{\tau} \rangle(\overline{a})) \triangleq \texttt{let}\ \overline{\texttt{reg}\ x} = f\langle \overline{\tau} \rangle(\overline{a}_{\restriction \overline{\beta''}})\ \texttt{in}\ \overline{x}_{\restriction \overline{\beta}}$$

In order to prove this case, we must build the following judgment:

$$\frac{\theta = [\overline{\alpha} \mapsto \overline{\tau}] \qquad \rho = [\overline{x} \mapsto \overline{a}] \qquad \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash f\langle \overline{\tau} \rangle(\overline{a}_{\restriction \overline{\beta'}}) : \mathcal{E}(\overline{\beta_\sigma, \tau_\sigma,\ \epsilon,\ \gamma})\theta\rho \qquad \mathcal{E}(\Delta) \cdot \mathcal{E}(\Gamma) + [\overline{x}_{\restriction \overline{\beta}} : \mathcal{E}_{\overline{\beta}}(\overline{\beta_\sigma \tau_\sigma})] \cdot \mathcal{E}(\Sigma) \vdash \overline{x}_{\restriction \overline{\beta}} : \mathcal{E}_{\overline{\beta}}(\overline{\beta_\sigma \tau_\sigma},\ \emptyset,\ \emptyset)}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \texttt{let}\ \overline{\texttt{reg}\ x} = f\langle \overline{\tau} \rangle(\overline{a}_{\restriction \overline{\beta''}})\ \texttt{in}\ \overline{x}_{\restriction \overline{\beta}} : \mathcal{E}_{\overline{\beta}}(\overline{\beta_\sigma \tau_\sigma},\ \epsilon,\ \gamma)\theta\rho} \tag{4.1}$$

First, by IH

$$\forall i.\, \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\beta_i'}(a_i) : \mathcal{E}_{\beta_i'}(\beta_i'' \tau_i,\, \emptyset,\, \emptyset)$$

for each a_i s.t. $\mathcal{G}(\beta_i'') = \texttt{reg}$. By taking only the judgments for which $\beta_i' = \texttt{reg}$, we get exactly the sequence $\bar{a}_{\restriction \overline{\beta'}}$. Since $\beta_i'' \sqsubseteq \beta_i'$, this a legal mask to extract each a_i.

Given that $\mathcal{G}(f) = \texttt{reg}$, we have

$$\mathcal{E}(\Delta)(f) = \forall \overline{\alpha}.\mathcal{E}(\overline{x : \beta' \tau}) \to \mathcal{E}(\overline{\beta_\sigma \tau_\sigma},\, \epsilon,\, \gamma)$$

so the following judgment is valid:

$$\frac{\begin{array}{c}\theta = [\overline{\alpha} \mapsto \overline{\tau}] \qquad \rho = [\bar{x} \mapsto \bar{a}] \\ \mathcal{E}(\Delta)(f) = \forall \overline{\alpha}.\mathcal{E}(\overline{x : \beta' \tau}) \to (\mathcal{E}(\overline{\beta_\sigma \tau_\sigma}),\, \epsilon,\, \emptyset) \qquad \forall i.\, \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\beta_i'}(a_i) : \mathcal{E}_{\beta_i'}(\beta_i'' \tau_i \theta,\, \emptyset,\, \emptyset)\end{array}}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash f\langle \overline{\tau} \rangle(\bar{a}_{\restriction \overline{\beta'}}) : (\mathcal{E}(\overline{\beta_\sigma, \tau_\sigma}),\, \epsilon,\, \emptyset)}$$

The premises $\mathcal{E}(\beta_i'') \sqsubseteq \mathcal{E}(\beta_i')$ are trivially true, since $\beta_i' = \beta_i'' = \texttt{reg}$.

Second, the judgment

$$\mathcal{E}(\Delta) \cdot \mathcal{E}(\Gamma) + [\bar{x}_{\restriction \overline{\beta}} : \mathcal{E}_{\overline{\beta}}(\overline{\beta_\sigma \tau_\sigma})] \cdot \mathcal{E}(\Sigma) \vdash \bar{x}_{\restriction \overline{\beta}} : (\mathcal{E}_{\overline{\beta}}(\overline{\beta_\sigma \tau_\sigma}),\, \emptyset,\, \emptyset)\theta\rho$$

is also valid, with $\overline{\beta}$ begin a legal mask since $\overline{\beta_\sigma} \sqsubseteq \overline{\beta}$. We can finally build the judgment (4.1), which concludes this case.

case (TIfAbsurd1). The typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash a : (\beta \texttt{Bool},\, \emptyset,\, \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{if } a \texttt{ then absurd else } e_2 : \sigma}$$

We distinguish two sub-cases:

- $\beta = \texttt{ghost}$: by the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$, we have $\mathcal{E}_{\overline{\beta}}(\texttt{if } a \texttt{ then } e_1 \texttt{ else } e_2) \triangleq \mathcal{E}_{\overline{\beta}}(e_2)$. By IH

  $$\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\sigma)$$

  which concludes this case.

- $\beta = \texttt{reg}$: by the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$, we have

  $$\mathcal{E}_{\overline{\beta}}(\texttt{if } a \texttt{ then } e_1 \texttt{ else } e_2) \triangleq \texttt{if } a \texttt{ then absurd else } \mathcal{E}_{\overline{\beta}}(e_2)$$

  By IH

  $$\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\sigma)$$

  so the following derivation is valid:

  $$\frac{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash a : (\texttt{reg Bool},\, \emptyset,\, \emptyset) \qquad \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash e_2 : \mathcal{E}_{\overline{\beta}}(\sigma)}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \texttt{if } a \texttt{ then absurd else } \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\sigma)}$$

case (TIfAbsurd2). Similar to the previous one.

case (TIf). By the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$, we have

$$\mathcal{E}_{\overline{\beta}}(\texttt{if } a \texttt{ then } e_1 \texttt{ else } e_2) \triangleq \texttt{if } \mathcal{E}(a) \texttt{ then } \mathcal{E}_{\overline{\beta}}(e_1) \texttt{ else } \mathcal{E}_{\overline{\beta}}(e_2)$$

By IH

$$\begin{gathered}\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}(a) : (\texttt{reg Bool}, \emptyset, \emptyset) \\ \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_1) : \mathcal{E}_{\overline{\beta}}(\sigma_1) \\ \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\sigma_2)\end{gathered}$$

By corollary 4.2.5.1 $\mathcal{E}_{\overline{\beta}}(\sigma_1 \cup \sigma_2) = \mathcal{E}_{\overline{\beta}}(\sigma_1) \cup \mathcal{E}_{\overline{\beta}}(\sigma_2)$. The following judgment is thus valid:

$$\frac{\begin{gathered}\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}(a) : (\texttt{reg Bool}, \emptyset, \emptyset) \\ \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_1) : \mathcal{E}_{\overline{\beta}}(\sigma_1) \qquad \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\sigma_2)\end{gathered}}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \texttt{if } \mathcal{E}(a) \texttt{ then } \mathcal{E}_{\overline{\beta}}(e_1) \texttt{ else } \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\sigma_1) \cup \mathcal{E}_{\overline{\beta}}(\sigma_2)}$$

case (TLet). We distinguish five different sub-cases:

- sub-case $\mathcal{G}(e_1) = \texttt{ghost}$: by the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$ we have $\mathcal{E}_{\overline{\beta}}(\texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2) \triangleq \mathcal{E}_{\overline{\beta}}(e_2)$.

  By IH

  $$\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2},\ \epsilon_2,\ \gamma_2)$$

  since $\mathcal{E}(\Gamma + [\bar{x} : \overline{\texttt{ghost } \tau_1}]) = \mathcal{E}(\Gamma)$.

  There are no observable effects in ϵ_1, *i.e.* $\epsilon_1 = \emptyset$, and the ghost effects are erased. Given that $\mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2}, \epsilon_1 \cup \epsilon_2, \gamma_1 \cup \gamma_2) = (\mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2}), \epsilon_1 \cup \epsilon_2, \emptyset) = (\mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2}), \epsilon_2, \emptyset)$, we conclude this case.

- sub-case $\mathcal{G}(e_2) = \texttt{ghost}$: by the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$ we have $\mathcal{E}_{\overline{\beta}}(\texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2) \triangleq \mathcal{E}_{\texttt{ghost}}(e_1)$.

  There are no observable effects in ϵ_2 and the ghost effects are erased. Since $\mathcal{G}(\overline{\beta_2}) = \texttt{ghost}$, the only admissible value for $\overline{\beta}$ is ghost, so $\mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2}, \epsilon_2, \gamma_2) = (\texttt{Unit}, \emptyset, \emptyset)$.

  By IH

  $$\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\texttt{ghost}}(e_1) : \mathcal{E}_{\texttt{ghost}}(\overline{\beta_1 \tau_1}, \epsilon_1, \gamma_1)$$

  which simplifies to $\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\texttt{ghost}}(e_1) : (\texttt{Unit}, \epsilon_1, \emptyset)$.

  Given that $\mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2}, \epsilon_1 \cup \epsilon_2, \gamma_1 \cup \gamma_2) = (\mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2}), \epsilon_1 \cup \epsilon_2, \emptyset) = (\texttt{Unit}, \epsilon_1, \emptyset)$, we conclude this case.

- sub-case $\mathcal{G}(\overline{\beta'}) = \texttt{ghost}$: by the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$ we have

  $$\mathcal{E}_{\overline{\beta}}(\texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2) \triangleq \texttt{let reg } \_ = \mathcal{E}_{\texttt{ghost}}(e_1) \texttt{ in } \mathcal{E}_{\overline{\beta}}(e_2)$$

  By IH

  $$\begin{gathered}\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\texttt{ghost}}(e_1) : (\texttt{Unit}, \epsilon_1, \emptyset) \\ \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2}, \epsilon_2, \gamma_2)\end{gathered}$$

  where the last judgment is valid since the identifier _ does not appear free in e_2 so it is not bound in Γ. The following judgment is valid:

  $$\frac{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\texttt{ghost}}(e_1) : (\texttt{Unit}, \epsilon_1, \emptyset) \qquad \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : (\mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2}), \epsilon_2, \emptyset)}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \texttt{let reg } \_ = \mathcal{E}_{\texttt{ghost}}(e_1) \texttt{ in } \mathcal{E}_{\overline{\beta}}(e_2) : (\mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2}), \epsilon_1 \cup \epsilon_2, \emptyset)}$$

- sub-case $\bar{x} \notin \mathrm{FV}(\mathcal{E}_{\overline{\beta}}(e_2))$: similar to the case $\mathcal{G}(e_1) = \texttt{ghost}$.
- finally, the last case in the definition of extraction for a `let..in` expression: by the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$ we have

$$\mathcal{E}_{\overline{\beta}}(\texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2) \triangleq \texttt{let } \overline{\texttt{reg}\, x}_{\restriction \overline{\beta'}} = \mathcal{E}_{\overline{\beta'}}(e_1) \texttt{ in } \mathcal{E}_{\overline{\beta}}(e_2)$$

By the typing relation we have

$$\Delta \cdot \Gamma \cdot \Sigma \vdash e_1 : (\overline{\beta_1 \tau_1}, \epsilon_1, \gamma_1) \qquad \Delta \cdot \Gamma + [\overline{x : \beta' \tau_1}] \cdot \Sigma \vdash e_2 : (\overline{\beta_2 \tau_2}, \epsilon_2, \gamma_2)$$

By IH

$$\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta'}}(e_1) : \mathcal{E}_{\overline{\beta'}}(\overline{\beta_1 \tau_1}, \epsilon_1, \gamma_1)$$
$$\mathcal{E}(\Delta) \cdot \mathcal{E}(\Gamma + [\overline{x : \beta' \tau_1}]) \cdot \mathcal{E}(\Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2}, \epsilon_2, \gamma_2)$$

where the last judgment simplifies to

$$\mathcal{E}(\Delta) \cdot \mathcal{E}(\Gamma) + [\bar{x}_{\restriction \overline{\beta'}} : \mathcal{E}_{\overline{\beta'}}(\overline{\beta_1 \tau_1})] \cdot \mathcal{E}(\Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : \mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2}, \epsilon_2, \gamma_2)$$

The following judgment is thus valid:

$$\frac{\begin{array}{c}\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta'}}(e_1) : (\mathcal{E}_{\overline{\beta'}}(\overline{\beta_1 \tau_1}), \epsilon_1, \emptyset) \\ \mathcal{E}(\Delta) \cdot \mathcal{E}(\Gamma) + [\bar{x}_{\restriction \overline{\beta'}} : \mathcal{E}_{\overline{\beta'}}(\overline{\beta_1 \tau_1})] \cdot \mathcal{E}(\Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_2) : (\mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2}), \epsilon_2, \emptyset)\end{array}}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \texttt{let } \overline{\texttt{reg}\, x}_{\restriction \overline{\beta'}} = \mathcal{E}(e_1) \texttt{ in } \mathcal{E}_{\overline{\beta}}(e_2) : (\mathcal{E}_{\overline{\beta}}(\overline{\beta_2 \tau_2}), \epsilon_1 \cup \epsilon_2, \emptyset)}$$

case (TRaiseReg). By the typing relation we know

$$E : (\beta_1' \tau_1 \dots \beta_n' \tau_n') \qquad \forall i.\, \Delta \cdot \Gamma \cdot \Sigma \vdash a_i : (\overline{\beta_i'' \tau_i}, \emptyset, \emptyset) \qquad \forall i.\, \beta_i'' \sqsubseteq \beta_i'$$

By the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$ we have $\mathcal{E}_{\overline{\beta}}(\texttt{raise}\, E \bar{a}) \triangleq \texttt{raise}\, E \bar{a}_{\restriction \mathcal{M}(E)}$, where $\mathcal{M}(E) = \overline{\beta'}$.
By IH

$$\forall i.\, \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\beta_i'}(a_i) : \mathcal{E}_{\beta_i'}(\overline{\beta_i'' \tau_i}, \emptyset, \emptyset)$$

considering only the extraction of a_i for which $\beta_i'' = \texttt{reg}$. By taking only the judgments where $\beta_i' = \texttt{reg}$ we get exactly the sequence $\bar{a}_{\restriction \overline{\beta'}}$. The following judgment is then valid:

$$\frac{E : \mathcal{E}(\beta_i' \tau_1 \dots \beta_n' \tau_n) \qquad \forall i.\, \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\beta_i'}(a_i) : \mathcal{E}_{\beta_i'}(\overline{\beta_i'' \tau_i}, \emptyset, \emptyset) \qquad \texttt{reg} \sqsubseteq \beta_i'}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \texttt{raise}\, E \bar{a}_{\restriction \overline{\beta'}} : (\texttt{reg}\, \tau, \emptyset + \texttt{raises}\, E, \emptyset)}$$

case (TLoop). The typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e : (\texttt{Unit}, \epsilon, \gamma)}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{loop}\, e : (\texttt{reg}\, \tau, \epsilon + \texttt{div}, \gamma)}$$

By IH

$$\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\texttt{reg}}(e) : (\texttt{Unit}, \epsilon, \emptyset)$$

The following judgment is valid:

$$\frac{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\texttt{reg}}(e) : (\texttt{Unit}, \epsilon, \emptyset)}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \texttt{loop}\, \mathcal{E}_{\texttt{reg}}(e) : (\texttt{reg}\, \tau, \epsilon + \texttt{div}, \emptyset)}$$

where $\texttt{loop}\, \mathcal{E}_{\texttt{reg}}(e) = \mathcal{E}_{\overline{\beta}}(\texttt{loop}\, e)$, for every $\overline{\beta}$.

case (TLoopDiv). Similar to the previous case.

case (TTry). The typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_0 : (\overline{\pi}, \epsilon, \gamma) \qquad \forall i.\ E_i : (\overline{\pi_i}) \qquad \Delta \cdot \Gamma + [\overline{x} : \overline{\pi_i}] \cdot \Sigma \vdash e_i : \sigma_i}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{try}\ e_0\ \texttt{with}\ \overline{E\overline{x} \Rightarrow e} : (\overline{\pi}, \epsilon - \overline{\texttt{raises}\ E_i}, \gamma - \overline{\texttt{raises}\ E_i}) \cup \bigcup_i \sigma_i}$$

where e and e_i' are real expressions. By IH

$$\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_0) : \mathcal{E}_{\overline{\beta}}(\overline{\pi}, \epsilon, \gamma)$$
$$\mathcal{E}(\Delta) \cdot \mathcal{E}(\Gamma + [\overline{x} : \overline{\pi_i}]) \cdot \mathcal{E}(\Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_i) : \mathcal{E}_{\overline{\beta}}(\sigma_i)$$

The last judgments simplify to

$$\mathcal{E}(\Delta) \cdot \mathcal{E}(\Gamma) + \mathcal{E}([\overline{x} : \overline{\pi_i}]) \cdot \mathcal{E}_{\overline{\beta}}(e_i) \vdash \mathcal{E}(\Sigma) : \mathcal{E}_{\overline{\beta}}(\sigma_i)$$

that is, the extraction of Γ, for each e_i' keeps only the x_j s.t. $\beta_{i_j} = \texttt{reg}$, which corresponds exactly to real arguments when extracting the exceptions E_i:

$$\mathcal{E}(E_i : (\beta_{i_1} \tau_{i_1} \dots \beta_{i_{n_i}} \tau_{i_{n_i}})) \triangleq E_i : \mathcal{E}(\beta_{i_1} \tau_{i_1} \dots \beta_{i_{n_i}} \tau_{i_{n_i}})$$

The following judgment is valid:

$$\frac{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e_0) : (\mathcal{E}_{\overline{\beta}}(\overline{\pi}), \epsilon, \emptyset) \qquad \mathcal{E}(\Delta) \cdot \mathcal{E}(\Gamma) + \mathcal{E}([\overline{x} : \overline{\pi_i}]) \cdot \mathcal{E}_{\overline{\beta}}(e_i) \vdash \mathcal{E}(\Sigma) : \mathcal{E}_{\overline{\beta}}(\sigma_i)}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \texttt{try}\ \mathcal{E}_{\overline{\beta}}(e)\ \texttt{with}\ \overline{E\overline{x}_{\restriction \mathcal{M}(E)} \Rightarrow e} : (\mathcal{E}_{\overline{\beta}}(\overline{\pi}), \epsilon - \overline{\texttt{raises}\ E_i}, \gamma - \overline{\texttt{raises}\ E_i}) \cup \bigcup_i \mathcal{E}_{\overline{\beta}}(\sigma_i)}$$

which concludes the case.

case (TRecord). We distinguish two sub-cases:

- sub-case $\overline{\beta} = \texttt{ghost}$: by the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$ we have $\mathcal{E}_{\texttt{ghost}}(\{\overline{f = a}\}) \triangleq ()$, which can be typed under any environment.
- sub-case $\overline{\beta} = \texttt{reg}$: by the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$ we have $\mathcal{E}_{\texttt{reg}}(\{\overline{f = a}\}) \triangleq \{\overline{f = a}\}_{\restriction \mathcal{M}(T)}$. The following judgment is valid:

  $$\frac{\texttt{type}\ T\overline{\alpha} = \mathcal{E}(\{\overline{f : \beta_f \tau_f}\}) \qquad \forall i.\ \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash a_i : (\beta_f \tau_{f_i}[\overline{\alpha} \mapsto \overline{\tau}], \emptyset, \emptyset) \qquad \forall i.\ \beta_i \sqsubseteq \beta_{f_i}}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash \{\overline{f = a}\} : (\texttt{reg}\ T\overline{\tau}, \emptyset, \emptyset)}$$

  by considering only the judgments $\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash a_i : (\beta_f \tau_{f_i}[\overline{\alpha} \mapsto \overline{\tau}], \emptyset, \emptyset)$ for which $\beta_{f_i} = \texttt{reg}$. Atoms a_i are not erased in $\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma)$.

case (TGet). We distinguish two sub-cases:

- sub-case $\overline{\beta} = \texttt{ghost}$: by the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$ we have $\mathcal{E}_{\texttt{ghost}}(a.f) \triangleq ()$, which can be typed under any environment.
- sub-case $\overline{\beta} = \texttt{reg}$: by the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$ we have $\mathcal{E}_{\texttt{reg}}(a.f) \triangleq a.f$. The following judgment is valid:

  $$\frac{\texttt{type}\ T\overline{\alpha} = \mathcal{E}(\{\dots,\ f : \texttt{reg}\ \tau_f,\ \dots\}) \qquad \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash a : (\texttt{reg}\ T\overline{\tau},\ \emptyset,\ \emptyset)}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash a.f : (\beta_f\ \tau_f[\overline{\alpha} \mapsto \overline{\tau}],\ \emptyset,\ \emptyset)}$$

  as atoms a_l and a_r are not erased in $\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma)$ and the field f is not erased in $\mathcal{E}(\{\dots,\ f : \texttt{reg}\ \tau_f,\ \dots\})$.

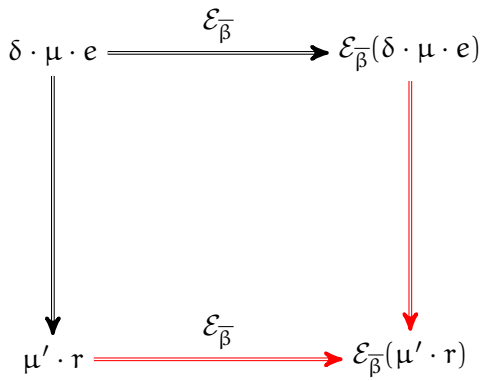


Figure 4.2: Preservation of Convergent Evaluation.

case (TAssign). By the definition of $\mathcal{E}_{\overline{\beta}}(\cdot)$ we have $\mathcal{E}_{\overline{\beta}}(a_l.f_i \leftarrow a_r) \triangleq a_l.f_i \leftarrow a_r$. Since $\mathcal{G}(a_l) = \texttt{reg}$, $\mathcal{G}(f_i) = \texttt{reg}$, and $\mathcal{G}(a_r) = \texttt{reg}$ the following judgment is valid:

$$\frac{\texttt{type}\ T\overline{\alpha} = \{\ldots, f_i : \texttt{reg}\ \tau_f, \ldots\} \quad \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash a_l : (\texttt{reg}\ T\overline{\tau}, \emptyset, \emptyset) \qquad \mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash a_r : (\texttt{reg}\ \tau_i[\overline{\alpha} \mapsto \overline{\tau}], \emptyset, \emptyset)}{\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma) \vdash a_l.f_i \leftarrow a_r : (\texttt{Unit}, \emptyset + \texttt{writes}\ a_r, \emptyset)}$$

as atoms a_l and a_r are not erased in $\mathcal{E}(\Delta \cdot \Gamma \cdot \Sigma)$. □

4.3 Semantics Preservation

In this section, we prove that extracted expressions preserve the semantics behavior of the original KidML program. Since we can only extract regular expressions, we are only interested in the *regular behavior*, *i.e.*, regular results and regular effects. We prove this preservation property both for convergent and divergent evaluations.

4.3.1 Preservation of Convergent Evaluation

In this section, we prove that if the evaluation of a KidML program converges to a semantic result r, then the evaluation of the extracted program converges into the result of extracting r. To show this result, we use the technique of *forward simulation*, as depicted in Fig. 4.2, where black arrows correspond to hypotheses, while the red ones stand for conclusions. Before stating this main result, we need the following auxiliary property:

Lemma 4.3.1 (Extraction preserves substitution)**.** *If* $\Delta \cdot \Gamma + [\bar{x} : \overline{\beta' \tau_1}] \cdot \Sigma \vdash e : (\overline{\beta'' \tau}, \epsilon, \gamma)$ *and* $\Delta \cdot \Gamma \cdot \Sigma \vdash \bar{v} : (\overline{\texttt{reg}\ \tau_1}, \emptyset, \emptyset)$*, then* $\mathcal{E}_{\overline{\beta}}(e_2)[\bar{x} \mapsto \bar{v}] = \mathcal{E}_{\overline{\beta}}(e_2[\bar{x} \mapsto \bar{v}])$*, for all* $\overline{\beta}$ *such that* $\overline{\beta''} \sqsubseteq \overline{\beta}$*.*

Proof. By straightforward induction on the structure of expression e. □

We also introduce the following auxiliary definitions:

Definition 4.3.1 (Extraction of stores)**.**

$\mathcal{E}(\mu)$

$$\begin{aligned} \mathcal{E}(\emptyset) &\triangleq \emptyset \\ \mathcal{E}(\mu[l \mapsto \{\overline{f = v}\}]) &\triangleq \mathcal{E}(\mu)[l \mapsto \{\overline{f = v}\}_{\restriction \mathcal{M}(\bar{f})}] \end{aligned}$$

Definition 4.3.2 (Extraction of procedures environment)**.**

$\Delta \vDash \delta$ $\boxed{\mathcal{E}(\delta)}$

$$\mathcal{E}(\emptyset) \triangleq \emptyset$$

$$\mathcal{E}(\delta[\mathtt{f} \mapsto (\overline{\mathtt{x} : \pi}, e)]) \triangleq \begin{cases} \mathcal{E}(\delta) & \text{if } \mathcal{G}(\mathtt{f}) = \mathtt{ghost} \\ \mathcal{E}(\delta)[\mathtt{f} \mapsto (\mathcal{E}(\overline{\mathtt{x} : \pi}), \mathcal{E}_{\overline{\beta'}}(e))] & \text{otherwise,} \\ & \text{where } \Delta(\mathtt{f}) = \forall \overline{\alpha}.\ (\overline{\mathtt{x} : \pi}) \to (\overline{\beta' \tau},\ \epsilon,\ \gamma) \end{cases}$$

In the definition of $\mathcal{E}(\delta)$, we use notation $\mathcal{G}(\mathtt{f}) = \mathtt{ghost}$ as a shortcut to state that $\mathtt{f}$ is a ghost function. Since we only extract well-typed programs, we only consider the evaluation of extracted programs under a well-typed procedures environment. In particular, we use the information stored for a function $\mathtt{f}$ in δ, in order to retrieve the mask of the returning type and extract the body e_1. The preservation theorem is stated as follows:

Theorem 4.3.2 (Convergent behavior preservation)**.** *If* $\Delta \cdot \Gamma \cdot \Sigma \vdash e : (\overline{\beta \tau'}, \epsilon, \gamma)$ *and* $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot \mathtt{r}$*, such that* $\mathtt{r} \neq \mathtt{absurd}$*,* $\Sigma \vDash \mu$*, and* $\Delta \vDash \delta$*, then* $\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(\mathtt{r})$*, for all* $\overline{\beta}$ *such that* $\overline{\beta'} \sqsubseteq \overline{\beta}$*.*

Proof. By induction on the semantics evaluation $\delta \cdot \mu \cdot e \Downarrow \mu' \cdot \mathtt{r}$. We rule out `absurd` since we suppose that we only extract verified KidML programs. This means that all `absurd` points in the code are proved to be unreachable, hence a KidML expression cannot evaluate down to such a semantics result.

cases (EvalVBar) and (EvalRaise). Trivial.

cases (EvalDiv), (EvalGhost), and (EvallAbsurd). Impossible

case (EvalLoop). The semantics evaluation ends up with

$$\frac{\delta \cdot \mu \cdot \mathtt{let\ reg\_} = e\ \mathtt{in\ loop}\ e \Downarrow \mu' \cdot \mathtt{r}}{\delta \cdot \mu \cdot \mathtt{loop}\ e \Downarrow \mu' \cdot \mathtt{r}}$$

We distinguish two sub-cases:

- case (EvalLetAbort): we can build the following derivation:

$$\dfrac{\dfrac{\delta \cdot \mu \cdot e \Downarrow \mu' \cdot \mathtt{r} \qquad \mathtt{abort\ r}}{\delta \cdot \mu \cdot \mathtt{let\ reg\_} = e\ \mathtt{in\ loop}\ e \Downarrow \mu' \cdot \mathtt{r}}\ \textsc{EvalLetAbort}}{\delta \cdot \mu \cdot \mathtt{loop}\ e \Downarrow \mu' \cdot \mathtt{r}}\ \textsc{EvalLoop}$$

  We distinguish two sub-sub-cases, according to the typing rule used at the bottom of the derivation:

  – case (TLoop): the typing derivation ends up with

$$\frac{\mathtt{CheckTermination}(\mathtt{loop}\ e) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e : (\mathtt{Unit},\ \epsilon,\ \gamma)}{\Delta \cdot \Gamma \cdot \Sigma \vdash \mathtt{loop}\ e : (\overline{\mathtt{reg}\ \tau},\ \epsilon,\ \gamma)}$$

  By IH

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}(e) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}(\mathtt{r})$$

We can build the following derivation:

$$\dfrac{\dfrac{\delta \cdot \mu \cdot \mathcal{E}(e) \Downarrow \mu' \cdot \mathcal{E}(r) \qquad \texttt{abort}\ \mathcal{E}(r)}{\delta \cdot \mu \cdot \texttt{let reg\_} = \mathcal{E}(e)\ \texttt{in loop}\ \mathcal{E}(e) \Downarrow \mu' \cdot \mathcal{E}(r)}\ \textsc{EvalLetAbort}}{\delta \cdot \mu \cdot \texttt{loop}\ \mathcal{E}(e) \Downarrow \mu' \cdot \mathcal{E}(r)}\ \textsc{EvalLoop}$$

which completes the case.

– case **(TLoopDiv)**: similar to the previous case. We may have an expression whose evaluation terminates, but for which oracle `CheckTermination` is not able to prove termination. Nonetheless, the semantics derivation is exactly the same as in the **(TLoop)** case, since our evaluation judgment does not care about the divergence effect assigned by the typing relation.

- case **(EvalLet)**: we can build the following derivation:

$$\dfrac{\dfrac{\delta \cdot \mu \cdot e \Downarrow \mu'' \cdot () \qquad \delta \cdot \mu'' \cdot \texttt{loop}\ e[\_ \mapsto ()] \Downarrow \mu'' \cdot r}{\delta \cdot \mu \cdot \texttt{let reg\_} = e\ \texttt{in loop}\ e \Downarrow \mu' \cdot r}\ \textsc{EvalLet}}{\delta \cdot \mu \cdot \texttt{loop}\ e \Downarrow \mu' \cdot r}\ \textsc{EvalLoop}$$

Let us note that expression e can only evaluate down to the unit value $()$, since it is a well-typed expression. By IH

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu') \cdot \mathcal{E}(e) \Downarrow \mathcal{E}(\mu') \cdot ()$$
$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu'') \cdot \mathcal{E}_{\overline{\beta}}(\texttt{loop}\ e) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)$$

We can build the following derivation:

$$\dfrac{\dfrac{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu') \cdot \mathcal{E}(e) \Downarrow \mathcal{E}(\mu'') \cdot () \qquad \mathcal{E}(\delta) \cdot \mathcal{E}(\mu'') \cdot \mathcal{E}_{\overline{\beta}}(\texttt{loop}\ e) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{let reg\_} = \mathcal{E}(e)\ \texttt{in}\ \mathcal{E}_{\overline{\beta}}(\texttt{loop}\ e) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)}\ \textsc{EvalLet}}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{loop}\ \mathcal{E}(e) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)}\ \textsc{EvalLoop}$$

where $\mathcal{E}_{\overline{\beta}}(\texttt{loop}\ e) = \texttt{loop}\ \mathcal{E}(e)$. This completes this case.

case (EvalLetAbort). The semantics evaluation ends up with

$$\dfrac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot r \qquad \texttt{abort}\ r}{\delta \cdot \mu \cdot \texttt{let}\ \overline{\beta' x} = e_1\ \texttt{in}\ e_2 \Downarrow \mu' \cdot r}$$

The typing derivation ends up with

$$\dfrac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_1 : (\overline{\beta_1 \tau_1},\ \epsilon_1,\ \gamma_1) \qquad \overline{\beta_1} \sqsubseteq \overline{\beta'} \qquad \Delta \cdot \Gamma + [\overline{x : \beta' \tau_1}] \cdot \Sigma \vdash e_2 : (\overline{\pi_2},\ \epsilon_2,\ \gamma_2)}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{let}\ \overline{\beta' x} = e_1\ \texttt{in}\ e_2 : (\overline{\pi_2},\ \epsilon_1 \cup \epsilon_2,\ \gamma_1 \cup \gamma_2)}$$

Since $\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot r$ such that $\texttt{abort}\ r$, where $r \neq \texttt{absurd}$, then we know e_1 is always an effectful regular expression, *i.e.*, it cannot be the case that e_1 is a completely ghost expression. We can thus apply the IH and get

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_1) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r) \tag{4.2}$$

for all $\overline{\beta}$, such that $\overline{\beta_1} \sqsubseteq \overline{\beta}$. The prove proceeds by case analysis on the definition of the extraction function:

- case $\mathcal{G}(e_1) = \texttt{ghost}$: impossible;

- case $\mathcal{G}(e_2) = \texttt{ghost}$: by instantiating $\overline{\beta}$ with $\texttt{ghost}$ in equation (4.2) we complete this case, since
$$\mathcal{E}_{\overline{\beta_2}}(\texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2) \triangleq \mathcal{E}_{\texttt{ghost}}(e_1)$$
for all $\overline{\beta_2}$ such that $\overline{\pi_2} \sqsubseteq \overline{\beta_2}$;
- case $\overline{\beta'} = \texttt{ghost}$ and $\epsilon_1 \neq \emptyset$: by the definition of $\mathcal{E}(\cdot)$
$$\mathcal{E}_{\overline{\beta_2}}(\texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2) \triangleq \texttt{let reg } \_ = \mathcal{E}_{\texttt{ghost}}(e_1) \texttt{ in } \mathcal{E}_{\overline{\beta_2}}(e_2)$$
We can build the following derivation:
$$\frac{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\texttt{ghost}}(e_1) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\texttt{ghost}}(r) \qquad \texttt{abort } \mathcal{E}_{\texttt{ghost}}(r)}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{let reg } \_ = \mathcal{E}_{\texttt{ghost}}(e_1) \texttt{ in } \mathcal{E}_{\overline{\beta_2}}(e_2) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\texttt{ghost}}(r)}$$
which completes this case;
- case $\epsilon_1 = \emptyset$: impossible;
- finally, the last case is the one where
$$\mathcal{E}_{\overline{\beta_2}}(\texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2) \triangleq \texttt{let } \overline{\texttt{reg } x}_{|\overline{\beta'}} = \mathcal{E}_{\overline{\beta'}}(e_1) \texttt{ in } \mathcal{E}_{\overline{\beta_2}}(e_2)$$
By instantiating $\overline{\beta}$ with $\overline{\beta'}$ in equation (4.2), we can build the following judgment:
$$\frac{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta'}}(\mathcal{E}(\mu') \cdot e_1) \Downarrow \mathcal{E}_{\overline{\beta'}}(r) \qquad \texttt{abort } \mathcal{E}_{\overline{\beta'}}(r)}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{let } \overline{\texttt{reg } x}_{|\overline{\beta'}} = \mathcal{E}_{\overline{\beta'}}(e_1) \texttt{ in } \mathcal{E}_{\overline{\beta_2}}(e_2) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta'}}(r)}$$
which completes this case.

case (EvalLet). The semantics evaluation ends up with
$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot \bar{v} \qquad \delta \cdot \mu' \cdot e_2[\bar{x} \mapsto \bar{v}] \Downarrow \mu'' \cdot r}{\delta \cdot \mu \cdot \texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2 \Downarrow \mu'' \cdot r}$$
The typing derivation ends up with
$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_1 : (\overline{\beta_1 \tau_1},\ \epsilon_1,\ \gamma_1) \qquad \overline{\beta_1} \sqsubseteq \overline{\beta'} \qquad \Delta \cdot \Gamma + [\bar{x} : \overline{\beta' \tau_1}] \cdot \Sigma \vdash e_2 : (\overline{\pi_2},\ \epsilon_2,\ \gamma_2)}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2 : (\overline{\pi_2},\ \epsilon_1 \cup \epsilon_2,\ \gamma_1 \cup \gamma_2)}$$
The proof proceeds by case analysis on the definition of the extraction function:

- if $\mathcal{G}(e_1) = \texttt{ghost}$: by the substitution lemma
$$\Delta \cdot \Gamma \cdot \Sigma \vdash e_2[\bar{x} \mapsto \bar{v}] : (\overline{\pi_2}, \epsilon_2, \gamma_2)[\bar{x} \mapsto \bar{v}]$$
By IH
$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_2[\bar{x} \mapsto \bar{v}]) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)$$
By the auxiliary lemma 4.3.1
$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_2)[\bar{x} \mapsto \bar{v}] \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)$$
which completes this case, since $\mathcal{E}_{\overline{\beta}}(\texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2) \triangleq \mathcal{E}_{\overline{\beta}}(e_2)$ and $\mathcal{E}_{\overline{\beta}}(e_2)[\bar{x} \mapsto \bar{v}] = \mathcal{E}_{\overline{\beta}}(e_2)$, and so there can be no occurrence of $\bar{x}$ in $\mathcal{E}_{\overline{\beta}}(e_2)$.

- if $\mathcal{G}(e_2) = \texttt{ghost}$: by the definition of $\mathcal{E}(\cdot)$

$$\mathcal{E}_{\overline{\beta}}(\texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2) \triangleq \mathcal{E}_{\texttt{ghost}}(e_1)$$

  Since $\mathcal{G}(e_1) = \texttt{reg}$, by IH

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\texttt{ghost}}(e_1) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\texttt{ghost}}(\bar{v})$$

  which completes this case;

- if $\overline{\beta'} = \texttt{ghost}$ and $\texttt{obs}(\epsilon_1)$: by the definition of $\mathcal{E}(\cdot)$

$$\mathcal{E}_{\overline{\beta}}(\texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2) \triangleq \texttt{let reg } \_ = \mathcal{E}_{\texttt{ghost}}(e_1) \texttt{ in } \mathcal{E}_{\overline{\beta}}(e_2)$$

  By the substitution lemma

$$\Delta \cdot \Gamma \cdot \Sigma \vdash e_2[\bar{x} \mapsto \bar{v}] : (\overline{\pi_2}, \epsilon_2, \gamma_2)[\bar{x} \mapsto \bar{v}]$$

  By IH

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\texttt{ghost}}(e_1) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\texttt{ghost}}(\bar{v})$$
$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(e_2[\bar{x} \mapsto \bar{v}]) \Downarrow \mathcal{E}(\mu'') \cdot \mathcal{E}_{\overline{\beta}}(r)$$

  where $\mathcal{E}_{\texttt{ghost}}(\bar{v}) = ()$. By the auxiliary lemma 4.3.1

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(e_2)[\bar{x} \mapsto \bar{v}] \Downarrow \mathcal{E}(\mu'')\mathcal{E}_{\overline{\beta}}(r)$$

  Since no occurrence of $\bar{x}$ can happen in $\mathcal{E}_{\overline{\beta}}(e_2)$, we have

$$\begin{aligned} \mathcal{E}_{\overline{\beta}}(e_2[\bar{x} \mapsto \bar{v}]) &= \mathcal{E}_{\overline{\beta}}(e_2) \\ &= \mathcal{E}_{\overline{\beta}}(e_2)[\_ \mapsto ()] \end{aligned}$$

  We can build the following judgment:

$$\frac{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\texttt{ghost}}(e_1) \Downarrow \mathcal{E}(\mu') \cdot () \qquad \mathcal{E}(\delta) \cdot \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(e_2)[\_ \mapsto ()] \Downarrow \mathcal{E}(\mu'') \cdot \mathcal{E}_{\overline{\beta}}(r)}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{let reg } \_ = \mathcal{E}_{\texttt{ghost}}(e_1) \texttt{ in } \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow \mathcal{E}(\mu'') \cdot \mathcal{E}_{\overline{\beta}}(r)}$$

  which completes this case;

- if $\bar{x} \notin \mathrm{FV}(\mathcal{E}_{\overline{\beta}}(e_2))$ and $e_1 = \emptyset$: similar to the case $\mathcal{G}(e_1) = \texttt{ghost}$;

- finally, the last case is the one where

$$\mathcal{E}_{\overline{\beta_2}}(\texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2) \triangleq \texttt{let } \overline{\texttt{reg } x}_{\restriction \overline{\beta'}} = \mathcal{E}_{\overline{\beta'}}(e_1) \texttt{ in } \mathcal{E}_{\overline{\beta_2}}(e_2)$$

  By the substitution lemma

$$\Delta \cdot \Gamma \cdot \Sigma \vdash e_2[\bar{x} \mapsto \bar{v}] : (\overline{\pi_2}, \epsilon_2, \gamma_2)[\bar{x} \mapsto \bar{v}]$$

  By IH

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta'}}(e_1) \Downarrow \bar{v}_{\restriction \overline{\beta'}}$$
$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu')\mathcal{E}_{\overline{\beta}}(e_2[\bar{x} \mapsto \bar{v}]) \Downarrow \mathcal{E}(\mu'') \cdot \mathcal{E}_{\overline{\beta}}(r)$$

  By the auxiliary lemma 4.3.1

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(e_2)[\bar{x} \mapsto \bar{v}] \Downarrow \mathcal{E}(\mu'') \cdot \mathcal{E}_{\overline{\beta}}(r)$$

There is no occurrence of ghost variables from $\bar{x}$ in $\mathcal{E}_{\overline{\beta}}(e_2)$, so the following holds:

$$\mathcal{E}_{\overline{\beta}}(e_2)[\bar{x} \mapsto \bar{v}] = \mathcal{E}_{\overline{\beta}}(e_2)[\bar{x}_{\restriction \overline{\beta'}} \mapsto \bar{v}_{\restriction \overline{\beta'}}]$$

We can build the following derivation:

$$\frac{\begin{array}{c}\mathcal{E}(\delta)\cdot\mathcal{E}(\mu)\cdot\mathcal{E}_{\overline{\beta'}}(\mathcal{E}(\mu')\cdot e_1) \Downarrow \mathcal{E}(\mu')\cdot\bar{v}_{\restriction\overline{\beta'}} \\ \mathcal{E}(\delta)\cdot\mathcal{E}(\mu')\cdot\mathcal{E}_{\overline{\beta}}(e_2)[\bar{x}_{\restriction\overline{\beta'}} \mapsto \bar{v}_{\restriction\overline{\beta'}}] \Downarrow \mathcal{E}(\mu'')\cdot\mathcal{E}_{\overline{\beta}}(r)\end{array}}{\mathcal{E}(\delta)\cdot\mathcal{E}(\mu)\cdot \texttt{let}\ \overline{\texttt{reg}\ x}_{\restriction\overline{\beta'}} = \mathcal{E}_{\overline{\beta'}}(e_1)\ \texttt{in}\ \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow \mathcal{E}(\mu'')\cdot\mathcal{E}_{\overline{\beta}}(r)}$$

which completes this case.

case (EvalApp). The semantics evaluation ends up with

$$\frac{\delta(f) = (\overline{x:\pi}, e) \qquad \|\overline{x:\pi}\| = \|\bar{v}\| \qquad \delta\cdot\mu\cdot e[\bar{x}\mapsto\bar{v}] \Downarrow \mu'\cdot r}{\delta\cdot\mu\cdot f(\bar{v}) \Downarrow \mu'\cdot r}$$

Since the whole application is a regular expression, the only rule we could have used at the bottom of the typing derivation is (**TApp**), as follows:

$$\frac{\begin{array}{c}\theta = [\overline{\alpha}\mapsto\overline{\tau}] \qquad \rho = [\bar{x}\mapsto\bar{a}] \\ \Delta(f) = \forall\overline{\alpha}.(\overline{x:\beta\tau'}) \to \sigma \qquad \forall i.\ \Delta\cdot\Gamma\cdot\Sigma \vdash a_i : (\beta_i'\tau_i'\theta,\ \emptyset,\ \emptyset) \qquad \forall i.\ \beta_i' \sqsubseteq \beta_i\end{array}}{\Delta\cdot\Gamma\cdot\Sigma \vdash f\langle\overline{\tau}\rangle\ (\bar{a}) : \sigma\theta\rho}$$

By the definition of $\mathcal{E}(\cdot)$, we have

$$\mathcal{E}_{\overline{\beta}}(f\langle\overline{\tau}\rangle\ (\bar{a})) \triangleq \texttt{let}\ \overline{\texttt{reg}\ x} = f\langle\overline{\tau}\rangle\ (\bar{a}_{\restriction\overline{\beta'}})\ \texttt{in}\ \bar{x}_{\restriction\overline{\beta}}$$

We distinguish two sub-cases:

- in case $e[\bar{x}\mapsto\bar{v}]$ evaluates down to a result r such that `abort` r: we have the body e of f is typed with

  $$\Delta\cdot\Gamma + [\bar{x} : \overline{\beta_i'\tau_i}\theta]\cdot\Sigma \vdash e : \sigma$$

  By the substitution lemma

  $$\Delta\cdot\Gamma\cdot\Sigma \vdash e[\bar{x}\mapsto\bar{v}] : \sigma$$

  where $\sigma = (\overline{\beta''},\ \epsilon,\ \gamma)$. By IH

  $$\mathcal{E}(\delta)\cdot\mathcal{E}(\mu)\cdot\mathcal{E}_{\overline{\beta''}}(e[\bar{x}\mapsto\bar{v}]) \Downarrow \mathcal{E}(\mu')\cdot\mathcal{E}_{\overline{\beta''}}(r)$$

  By the auxiliary lemma 4.3.1

  $$\mathcal{E}(\delta)\cdot\mathcal{E}(\mu)\cdot\mathcal{E}_{\overline{\beta''}}(e)[\bar{x}\mapsto\bar{v}] \Downarrow \mathcal{E}(\mu')\cdot\mathcal{E}_{\overline{\beta''}}(r)$$

  Since there cannot be any occurrence of the ghost variables in $\bar{x}$ in $\mathcal{E}_{\overline{\beta''}}(e)$, we have

  $$\mathcal{E}_{\overline{\beta''}}(e)[\bar{x}\mapsto\bar{v}] = \mathcal{E}_{\overline{\beta''}}(e)[\bar{x}_{\restriction\overline{\beta'}} \mapsto \bar{v}_{\restriction\overline{\beta'}}]$$

  We can build the following derivation:

  $$\textsc{EvalLetAbort}\ \frac{\textsc{EvalApp}\ \dfrac{\begin{array}{c}\mathcal{E}(\delta)\cdot\mathcal{E}(\mu)\cdot\mathcal{E}_{\overline{\beta''}}(e)[\bar{x}_{\restriction\overline{\beta'}} \mapsto \bar{v}_{\restriction\overline{\beta'}}] \Downarrow \mathcal{E}(\mu')\cdot\mathcal{E}_{\overline{\beta''}}(r) \\ \mathcal{E}(\delta) = (\mathcal{E}(\overline{x:\pi}), \mathcal{E}_{\overline{\beta''}}(e)) \qquad \|\mathcal{E}(\overline{x:\pi})\| = \|\bar{v}_{\restriction\overline{\beta'}}\|\end{array}}{\mathcal{E}(\delta)\cdot\mathcal{E}(\mu)\cdot f\langle\overline{\tau}\rangle\ (\bar{v}_{\restriction\overline{\beta'}}) \Downarrow \mathcal{E}(\mu')\cdot\mathcal{E}_{\overline{\beta''}}(r)} \qquad \texttt{abort}\ \mathcal{E}_{\overline{\beta''}}(r)}{\mathcal{E}(\delta)\cdot\mathcal{E}(\mu)\cdot \texttt{let}\ \overline{\texttt{reg}\ x} = f\langle\overline{\tau}\rangle\ (\bar{a}_{\restriction\overline{\beta'}})\ \texttt{in}\ \bar{x}_{\restriction\overline{\beta}} \Downarrow \mathcal{E}(\mu')\cdot\mathcal{E}_{\overline{\beta''}}(r)}$$

  which completes this case.

- in case $e[\bar{x} \mapsto \bar{v}]$ evaluates down to a sequence of values $\overline{v_r}$: using the same reasoning as in the previous case, we have

$$\mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta''}}(e)[\bar{x}_{\restriction \overline{\beta'}} \mapsto \bar{v}_{\restriction \overline{\beta'}}] \Downarrow \mathcal{E}(\mu') \cdot \overline{v_r}_{\restriction \overline{\beta''}}$$

The following trivially holds:

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu') \cdot \overline{v_r}_{\restriction \overline{\beta}} \Downarrow \mathcal{E}(\mu') \cdot \overline{v_r}_{\restriction \overline{\beta}}$$

We have that $\bar{x}_{\restriction \overline{\beta}}[\bar{x}_{\restriction \overline{\beta''}} \mapsto \overline{v_r}_{\restriction \overline{\beta''}}] = \overline{v_r}_{\restriction \overline{\beta}}$, where $\overline{\beta''} \sqsubseteq \overline{\beta}$. This holds since the sequence $\bar{x}_{\restriction \overline{\beta''}}$ corresponds to the regular results of application, and $\bar{x}_{\restriction \overline{\beta}}$ and $\bar{v}_{\restriction \overline{\beta}}$ are restrictions of $\bar{x}_{\restriction \overline{\beta''}}$ and $\bar{v}_{\restriction \overline{\beta''}}$, respectively. We can build the following derivation:

$$\textsc{EvalApp}\ \frac{\begin{array}{c}\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta''}}(e)[\bar{x}_{\restriction \overline{\beta'}} \mapsto \bar{v}_{\restriction \overline{\beta'}}] \Downarrow \mathcal{E}(\mu') \cdot \overline{v_r}_{\restriction \overline{\beta''}} \\ \mathcal{E}(\delta) = (\mathcal{E}(\overline{x : \pi}), \mathcal{E}_{\overline{\beta''}}(e)) \qquad \|\mathcal{E}(\overline{x : \pi})\| = \|\bar{v}_{\restriction \overline{\beta'}}\|\end{array}}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot f\langle\overline{\tau}\rangle\ (\bar{v}_{\restriction \overline{\beta'}}) \Downarrow \mathcal{E}(\mu') \cdot \overline{v_r}_{\restriction \overline{\beta''}} \qquad \mathcal{E}(\delta) \cdot \mathcal{E}(\mu') \cdot \overline{v_r}_{\restriction \overline{\beta}} \Downarrow \mathcal{E}(\mu') \cdot \overline{v_r}_{\restriction \overline{\beta}}}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{let}\ \overline{\texttt{reg}\ x} = f\langle\overline{\tau}\rangle\ (\bar{a}_{\restriction \overline{\beta'}})\ \texttt{in}\ \bar{x}_{\restriction \overline{\beta}} \Downarrow \mathcal{E}(\mu') \cdot \overline{v_r}_{\restriction \overline{\beta}}}\ \textsc{EvalLet}$$

which completes this case.

case (EvalFun). The semantics evaluation ends up with

$$\frac{\delta[f \mapsto (\overline{x : \pi}, e_1)] \cdot \mu \cdot e_2 \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{fun}\ f(\overline{x : \pi}) : \sigma = e_1\ \texttt{in}\ e_2 \Downarrow \mu' \cdot r}$$

The typing derivation ends up with

$$\frac{\begin{array}{c}\Delta \cdot \Gamma + [\overline{x : \pi}] \cdot \Sigma \vdash e_1 : \sigma' \qquad \sigma'_{\restriction \Gamma + [\overline{x:\pi}]} \sqsubseteq (\overline{\beta'\tau}, \epsilon, \gamma) \\ \Delta + [f : \forall\overline{\alpha}.(\overline{x : \pi}) \to (\overline{\beta'\tau}, \epsilon, \gamma)] \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma'' \qquad \overline{\alpha} \# \Gamma \qquad \overline{\alpha} \notin \Delta\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : (\overline{\beta'\tau}, \epsilon, \gamma) = e_1\ \texttt{in}\ e_2 : \sigma''}\ (\textsc{TFun})$$

We distinguish two sub-cases, according to the ghost status of function f:

- if f is a ghost function: in this case, by the definition of $\mathcal{E}(\cdot)$:

$$\mathcal{E}_{\overline{\beta}}(\texttt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : (\overline{\beta'\tau}, \epsilon, \gamma) = e_1\ \texttt{in}\ e_2) \triangleq \mathcal{E}_{\overline{\beta}}(e_2)$$

By IH

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)$$

where f is not contained in $\mathcal{E}(\delta)$ since it is a ghost function. This completes the case.

- if f is a regular function: in this case, by the definition of $\mathcal{E}(\cdot)$:

$$\begin{array}{l}\mathcal{E}_{\overline{\beta}}(\texttt{fun}\ f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : (\overline{\beta'\tau}, \epsilon, \gamma) = e_1\ \texttt{in}\ e_2) \quad \triangleq \\ \texttt{fun}\ f\langle\overline{\alpha}\rangle\ (\mathcal{E}(\overline{x : \pi})) : \mathcal{E}(\overline{\beta'\tau}, \epsilon, \gamma) = \mathcal{E}_{\overline{\beta'}}(e_1)\ \texttt{in}\ \mathcal{E}_{\overline{\beta}}(e_2)\end{array}$$

By IH

$$\mathcal{E}(\delta[\mapsto (\overline{x : \pi}, e_1)]) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)$$

where $\mathcal{E}(\delta[f \mapsto (\overline{x : \pi}, e_1)]) = \mathcal{E}(\delta)[f \mapsto (\mathcal{E}(\overline{x : \pi}), \mathcal{E}_{\overline{\beta'}}(e_1))]$. We can build the following derivation:

$$\frac{\mathcal{E}(\delta)[f \mapsto (\mathcal{E}(\overline{x : \pi}), \mathcal{E}_{\overline{\beta'}}(e_1))] \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{fun}\ f(\overline{x : \pi}) : \mathcal{E}(\overline{\beta'\tau}, \epsilon, \gamma) = \mathcal{E}_{\overline{\beta'}}(e_1)\ \texttt{in}\ \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)}$$

which completes this case.

case (EvalRec). Similar to the previous case. As for the (**TLoopDiv**) case, the divergence effect does not affect the evaluation derivation.

case (EvalIfFalse). The semantics evaluation ends up with

$$\frac{\delta \cdot \mu \cdot e_2 \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{if false then } e_1 \texttt{ else } e_2 \Downarrow \mu' \cdot r}$$

We know that $r \neq \texttt{absurd}$, so we only need to distinguish two cases according to the typing rule used at the bottom of the derivation:

- case (**TIfAbsurd1**): the typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{false} : (\beta\texttt{Bool}, \emptyset, \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{if false then absurd else } e_2 : \sigma}$$

  By IH

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)$$

  By the definition of $\mathcal{E}(\cdot)$:

$$\mathcal{E}_{\overline{\beta}}(\texttt{if false then absurd else } e_2) \triangleq \texttt{if false then absurd else } \mathcal{E}_{\overline{\beta}}(e_2)$$

  This is the only possibility, since the constant `false` is always typed as a regular variable. We can build the following derivation:

$$\frac{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{if false then } e_1 \texttt{ else } \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)}$$

  which completes this case.

- case (**TIf**): the typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{false} : (\texttt{reg Bool}, \emptyset, \emptyset) \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_1 : \sigma_1 \qquad \Delta \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma_2}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{if false then } e_1 \texttt{ else } e_2 : \sigma_1 \cup \sigma_2} \ (\textsc{TIf})$$

  By the definition of $\mathcal{E}(\cdot)$:

$$\mathcal{E}_{\overline{\beta}}(\texttt{if false then } e_1 \texttt{ else } e_2) \triangleq \texttt{if false then } \mathcal{E}_{\overline{\beta}}(e_1) \texttt{ else } \mathcal{E}_{\overline{\beta}}(e_2)$$

  Expressions e_1 and e_2 must be both typed as regular expressions. By IH

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)$$

  We can build the following derivation:

$$\frac{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{if false then } \mathcal{E}_{\overline{\beta}}(e_1) \texttt{ else } \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)}$$

  which completes this case.

case (EvalIfTrue). Similar to the previous case.

case (EvalTry). The semantics evaluation ends up with

$$\frac{\delta \cdot \mu \cdot e_0 \Downarrow \mu' \cdot \bar{v}}{\delta \cdot \mu \cdot \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow \mu' \cdot \bar{v}}$$

The typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_0 : (\overline{\pi},\ \epsilon,\ \gamma) \qquad \forall i.\ E_i : \overline{\pi_i} \qquad \forall i.\ \Delta \cdot \Gamma + [\bar{x} : \overline{\pi_i}] \cdot \Sigma \vdash e_i : \sigma_i}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} : (\overline{\pi},\ \epsilon - \overline{\texttt{raises}\ E_i},\ \gamma - \overline{\texttt{raises}\ E_i}) \cup \bigcup_i \sigma_i}$$

By the definition of $\mathcal{E}(\cdot)$:

$$\mathcal{E}_{\overline{\beta}}(\texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e'}) \triangleq \texttt{try}\ \mathcal{E}_{\overline{\beta}}(e_0)\ \texttt{with}\ \mathcal{E}_{\overline{\beta}}(\overline{E\bar{x} \Rightarrow\ e})$$

By IH

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_0) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(\bar{v})$$

We can build the following derivation:

$$\frac{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_0) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(\bar{v})}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{try}\ \mathcal{E}_{\overline{\beta}}(e_0)\ \texttt{with}\ \mathcal{E}_{\overline{\beta}}(\overline{E\bar{x} \Rightarrow\ e}) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(\bar{v})}$$

which completes this case.

case (EvalTryAbort). The semantics evaluation ends up with

$$\frac{\delta \cdot \mu \cdot e_0 \Downarrow \mu' \cdot r \qquad \texttt{abort}\ r \qquad \forall i.\ E_i \neq r}{\delta \cdot \mu \cdot \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow \mu' \cdot r}$$

The typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_0 : (\overline{\pi},\ \epsilon,\ \gamma) \qquad \forall i.\ E_i : \overline{\pi_i} \qquad \forall i.\ \Delta \cdot \Gamma + [\bar{x} : \overline{\pi_i}] \cdot \Sigma \vdash e_i : \sigma_i}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} : (\overline{\pi},\ \epsilon - \overline{\texttt{raises}\ E_i},\ \gamma - \overline{\texttt{raises}\ E_i}) \cup \bigcup_i \sigma_i}$$

By the definition of $\mathcal{E}(\cdot)$:

$$\mathcal{E}_{\overline{\beta}}(\texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e'}) \triangleq \texttt{try}\ \mathcal{E}_{\overline{\beta}}(e_0)\ \texttt{with}\ \mathcal{E}_{\overline{\beta}}(\overline{E\bar{x} \Rightarrow\ e})$$

By IH

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_0) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)$$

We can build the following derivation:

$$\frac{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_0) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r) \qquad \texttt{abort}\ \mathcal{E}_{\overline{\beta}}(r) \qquad \forall i.\ E_i \neq \mathcal{E}_{\overline{\beta}}(r)}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{try}\ \mathcal{E}_{\overline{\beta}}(e_0)\ \texttt{with}\ \mathcal{E}_{\overline{\beta}}(\overline{E\bar{x} \Rightarrow e}) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)}$$

which completes this case.

case (EvalTryExn). The semantics evaluation ends up with

$$\frac{\delta \cdot \mu \cdot e_0 \Downarrow \mu'' \cdot \texttt{raise}\ E'\bar{v} \qquad E' = E_i \qquad \forall j.\, j < i \to E' \neq E_j \qquad \delta \cdot \mu'' \cdot e_i[\bar{x} \mapsto \bar{v}] \Downarrow \mu' \cdot r}{\delta \cdot \mu \cdot \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow \mu' \cdot r}$$

The typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_0 : (\overline{\pi},\ \epsilon,\ \gamma) \qquad \forall i.\, E_i : \overline{\pi_i} \qquad \forall i.\, \Delta \cdot \Gamma + [\bar{x} : \overline{\pi_i}] \cdot \Sigma \vdash e_i : \sigma_i}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} : (\overline{\pi},\ \epsilon - \overline{\texttt{raises}\ E_i},\ \gamma - \overline{\texttt{raises}\ E_i}) \cup \bigcup_i \sigma_i}$$

By the definition of $\mathcal{E}(\cdot)$:

$$\mathcal{E}_{\overline{\beta}}(\texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e'}) \triangleq \texttt{try}\ \mathcal{E}_{\overline{\beta}}(e_0)\ \texttt{with}\ \mathcal{E}_{\overline{\beta}}(\overline{E\bar{x} \Rightarrow\ e})$$

We distinguish two sub-cases, according to the ghost status of e_i:

- in case e_i is a regular expression: by IH

  $$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_0) \Downarrow \mathcal{E}(\mu'') \cdot E'\bar{v}_{\restriction \mathcal{M}(E')}$$
  $$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu'') \cdot \mathcal{E}_{\overline{\beta}}(e_i[\bar{x} \mapsto \bar{v}]) \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)$$

  By the auxiliary lemma 4.3.1

  $$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu'') \cdot \mathcal{E}_{\overline{\beta}}(e_i)[\bar{x} \mapsto \bar{v}] \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)$$

  and since there can only be regular occurrences of $\bar{x}$ in $\mathcal{E}_{\overline{\beta}}(e_i)$

  $$\mathcal{E}_{\overline{\beta}}(e_i)[\bar{x} \mapsto \bar{v}] = \mathcal{E}_{\overline{\beta}}(e_i)[\bar{x}_{\restriction \mathcal{M}(E')} \mapsto \bar{v}_{\restriction \mathcal{M}(E')}]$$

  We can build the following derivation:

  $$\frac{\begin{array}{c}\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_0) \Downarrow \mathcal{E}(\mu'') \cdot E'\bar{v}_{\restriction \mathcal{M}(E')} \\ E' = E_i \qquad \forall j.\, j < i \to E' \neq E_j \qquad \mathcal{E}(\delta) \cdot \mathcal{E}(\mu'') \cdot \mathcal{E}_{\overline{\beta}}(e_i)[\bar{x}_{\restriction \mathcal{M}(E')} \mapsto \bar{v}_{\restriction \mathcal{M}(E')}] \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)\end{array}}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow \mathcal{E}(\mu') \cdot \mathcal{E}_{\overline{\beta}}(r)}$$

  which completes this case.

- in case e_i is a ghost expression: by IH

  $$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_0) \Downarrow \mathcal{E}(\mu'') \cdot E'\bar{v}_{\restriction \mathcal{M}(E')}$$

  By the definition of $\mathcal{E}(\cdot)$

  $$\mathcal{E}_{\overline{\beta}}(E'\bar{x} \Rightarrow e_i) \triangleq E'\bar{x}_{\restriction \mathcal{E}'} \Rightarrow ()$$

  We know that $\overline{\beta} = \texttt{ghost}$ and that $\mu'' = \mu'$, otherwise e_i would be an effectful expression and could not be considered as a ghost expression. We then have

  $$\mathcal{E}_{\overline{\beta}}(r) = ()$$

  We can build the following derivation

  $$\frac{\begin{array}{c}\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_0) \Downarrow \mathcal{E}(\mu'') \cdot E'\bar{v}_{\restriction \mathcal{M}(E')} \\ E' = E_i \qquad \forall j.\, j < i \to E' \neq E_j \qquad \mathcal{E}(\delta) \cdot \mathcal{E}(\mu'') \cdot () \Downarrow \mathcal{E}(\mu'') \cdot ()\end{array}}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{try}\ e_0\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \Downarrow \mathcal{E}(\mu'') \cdot ()}$$

  which completes this case.

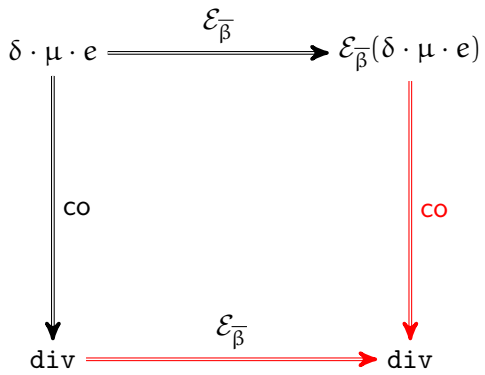


Figure 4.3: Preservation of Divergent Evaluation.

case (EvalRecord). There are only two possible masks to extract such an expression. Either $\overline{\beta} = \texttt{ghost}$, in which case

$$\mathcal{E}_{\texttt{ghost}}(\{\, \overline{\mathsf{f} = \nu} \,\}) \triangleq ()$$

We have $\mathcal{E}_{\texttt{ghost}}(l) = ()$ thus, the result follows immediately from the trivial evaluation

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot () \Downarrow \mathcal{E}(\mu) \cdot ()$$

Otherwise $\overline{\beta} = \texttt{reg}$, in which case

$$\mathcal{E}_{\texttt{reg}}(\{\, \overline{\mathsf{f} = \nu} \,\}) \triangleq \{\, \overline{\mathsf{f} = \nu} \,\}_{\restriction \mathcal{M}(\overline{\mathsf{f}})}$$

Since $\mathcal{E}_{\texttt{reg}}(l) = l$, we can build the following derivation

$$\frac{l \notin \texttt{dom}(\mathcal{E}(\mu)) \qquad \mathcal{E}(\mu') = \mathcal{E}(\mu)[l \mapsto \{\, \overline{\mathsf{f} = \nu} \,\}_{\restriction \mathcal{M}(\overline{\mathsf{f}})}]}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \{\, \overline{\mathsf{f} = \nu} \,\}_{\restriction \mathcal{M}(\overline{\mathsf{f}})} \Downarrow \mathcal{E}(\mu') \cdot l}$$

which completes this case.

case (EvalGet). Similar to the previous case, even easier when $\overline{\beta} = \texttt{reg}$. For both cases, location l is always in $\texttt{dom}(\mathcal{E}(\mu))$.

case (EvalAssign). Trivial: by the hypotheses of the theorem, such an assignment stands for a regular effect, so the extraction function does not affect this expression.

case (EvalErr). Impossible. □

4.3.2 Preservation of Divergent Evaluation

We focus now on the proof that a divergent KidML program is extracted into a program that diverges as well. Ghost code cannot diverge, hence divergence is an effect that we preserve in the extracted program. We use our co-inductive evaluation judgment and, once again, the technique of forward simulation to prove such a result, as illustrated in Fig. 4.3. This preservation theorem is state as follows:

Theorem 4.3.3 (Divergent behavior preservation)**.** *If* $\Delta \cdot \Gamma \cdot \Sigma \vdash e : (\overline{\beta\tau'}, \epsilon, \gamma)$ *and* $\delta \cdot \mu \cdot e \Downarrow^{co} \texttt{div}$*, such that* $\Sigma \vDash \mu$*, and* $\Delta \vDash \delta$*, then* $\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e) \Downarrow^{co} \texttt{div}$*, for all* $\overline{\beta}$ *such that* $\overline{\beta'} \sqsubseteq \overline{\beta}$*.*

Proof. By co-induction on the semantics evaluation $\delta \cdot \mu \cdot e \Downarrow^{co} \texttt{div}$. This proof actually follows the same structure as the proof of theorem 4.3.2 so we do not detail all the cases here. We present just the cases for the evaluation of `let..in` expressions.

case (CoEvalLoop). Semantics co-evaluation ends up with

$$\frac{}{\delta \cdot \mu \cdot \texttt{let reg \_} = e \texttt{ in loop } e \Downarrow^{\texttt{co}} \texttt{div}}$$

Since the evaluation of this expression diverges, the rule used at the bottom of typing derivation is (**TLoopDiv**), as follows:

$$\frac{\begin{array}{c}\neg\texttt{CheckTermination}(\texttt{loop } e) \\ \Delta \cdot \Gamma \cdot \Sigma \vdash e : (\texttt{Unit},\ \epsilon,\ \gamma)\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{loop } e : (\overline{\texttt{reg } \tau},\ \epsilon + \texttt{div},\ \gamma)}$$

We distinguish two sub-cases:

- sub-case (**CoEvalLetAbort**): we can build the following derivation:

$$\dfrac{\dfrac{\delta \cdot \mu \cdot e \Downarrow^{\texttt{co}} \texttt{div} \qquad \texttt{abort div}}{\delta \cdot \mu \cdot \texttt{let reg \_} = e \texttt{ in loop } e \Downarrow^{\texttt{co}} \texttt{div}}\ (\textsc{CoEvalLetAbort})}{\delta \cdot \mu \cdot \texttt{loop } e \Downarrow^{\texttt{co}} \texttt{div}}\ (\textsc{CoEvalLoop})$$

  By co-inductive hypothesis

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}(e) \Downarrow^{\texttt{co}} \texttt{div}$$

  We can build the following derivation:

$$\dfrac{\dfrac{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}(e) \Downarrow^{\texttt{co}} \texttt{div} \qquad \texttt{abort div}}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{let reg\_} = \mathcal{E}(e) \texttt{ in loop } \mathcal{E}(e) \Downarrow^{\texttt{co}} \texttt{div}}\ (\textsc{CoEvalLetAbort})}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{loop } \mathcal{E}(e) \Downarrow^{\texttt{co}} \texttt{div}}\ (\textsc{CoEvalLoop})$$

  which completes this case.

- sub-case (**CoEvalLet**): we can build the following derivation:

$$\dfrac{\dfrac{\delta \cdot \mu \cdot e \Downarrow \mu' \cdot () \qquad \mu' \cdot \texttt{loop } e \Downarrow^{\texttt{co}} \texttt{div}}{\delta \cdot \mu \cdot \texttt{let reg \_} = e \texttt{ in loop } e \Downarrow^{\texttt{co}} \texttt{div}}\ (\textsc{CoEvalLet})}{\delta \cdot \mu \cdot \texttt{loop } e \Downarrow^{\texttt{co}} \texttt{div}}\ (\textsc{CoEvalLoop})$$

  By co-induction hypotheses

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu') \cdot \texttt{loop } \mathcal{E}(e) \Downarrow^{\texttt{co}} \texttt{div}$$

  We can build the following derivation:

$$\dfrac{\dfrac{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot e \Downarrow () \qquad \mathcal{E}(\delta) \cdot \mathcal{E}(\mu') \cdot \texttt{loop } \mathcal{E}(e) \Downarrow^{\texttt{co}} \texttt{div}}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu') \cdot \texttt{let reg \_} = \mathcal{E}(e) \texttt{ in loop } \mathcal{E}(e) \Downarrow^{\texttt{co}} \texttt{div}}\ (\textsc{CoEvalLet})}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{loop } \mathcal{E}(e) \Downarrow^{\texttt{co}} \texttt{div}}\ (\textsc{CoEvalLoop})$$

  which completes this case.

case (CoEvalLetAbort). Semantics co-evaluation ends up with

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow^{co} \texttt{div} \qquad \texttt{abort div}}{\delta \cdot \mu \cdot \texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2 \Downarrow^{co} \texttt{div}}$$

In this case, expression e_1 diverges causing the whole `let..in` expression to diverge. The typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_1 : (\overline{\beta_1' \tau_1},\ \epsilon_1,\ \gamma_1) \qquad \overline{\beta_1'} \sqsubseteq \overline{\beta} \qquad \Delta \cdot \Gamma + [\overline{x : \beta' \tau_1}] \cdot \Sigma \vdash e_2 : (\overline{\pi_2},\ \epsilon_2,\ \gamma_2)}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2 : (\overline{\pi_2},\ \epsilon_1 \cup \epsilon_2,\ \gamma_1 \cup \gamma_2)}$$

By co-induction hypothesis

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta'}}(e_1) \Downarrow^{co} \texttt{div}$$

We now proceed by case analysis on the definition of the extraction function. Since e_1 is a stateful expression, the first and the third cases of the `let..in` extraction rule cannot happen. For the remaining three cases, we can always use rule (CoEvalLetAbort) to the complete the case. For instance, if e_2 is a regular expression, we complete the case with the following derivation:

$$\frac{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta'}}(e_1) \Downarrow^{co} \texttt{div} \qquad \texttt{abort div}}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{let } \overline{\texttt{reg } x}_{\restriction \overline{\beta'}} = \mathcal{E}_{\overline{\beta'}}(e_1) \texttt{ in } \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow^{co} \texttt{div}}$$

case (CoEvalLet). Semantics co-evaluation ends up with

$$\frac{\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot \bar{v} \qquad \delta \cdot \mu' \cdot e_2[\bar{x} \mapsto \bar{v}] \Downarrow^{co} \texttt{div}}{\delta \cdot \mu \cdot \texttt{let } \overline{\beta x} = e_1 \texttt{ in } e_2 \Downarrow^{co} \texttt{div}}$$

In this case, expression e_2 diverges causing the whole `let..in` expression to diverge. The evaluation of e_1 terminates in the sequence of values $\bar{v}$, so we have

$$\delta \cdot \mu \cdot e_1 \Downarrow \mu' \cdot \bar{v}$$

The typing derivation ends up with

$$\frac{\Delta \cdot \Gamma \cdot \Sigma \vdash e_1 : (\overline{\beta_1' \tau_1},\ \epsilon_1,\ \gamma_1) \qquad \overline{\beta_1'} \sqsubseteq \overline{\beta} \qquad \Delta \cdot \Gamma + [\overline{x : \beta' \tau_1}] \cdot \Sigma \vdash e_2 : (\overline{\pi_2},\ \epsilon_2,\ \gamma_2)}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{let } \overline{\beta' x} = e_1 \texttt{ in } e_2 : (\overline{\pi_2},\ \epsilon_1 \cup \epsilon_2,\ \gamma_1 \cup \gamma_2)}$$

By co-induction hypotheses

$$\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow^{co} \texttt{div}$$

We now proceed by case analysis on the definition of the extraction function. The first and third cases in the extraction of a `let..in` expression follow directly by the co-induction hypotheses. Since e_2 is a stateful expression, the second case is impossible. For the remaining two cases, we can always use rule (CoEvalLet) and the co-induction hypotheses to build a derivation that completes the case.

case (CoEvalRec). Semantics co-evaluation ends up with

$$\frac{\delta[f \mapsto (\overline{x : \pi}, e_1)] \cdot \mu \cdot e_2 \Downarrow^{co} \texttt{div}}{\delta \cdot \mu \cdot \texttt{rec } f\langle\overline{\alpha}\rangle(\overline{x : \pi}) : \sigma = e_1 \texttt{ in } e_2 \Downarrow^{co} \texttt{div}}$$

We distinguish two sub-cases, according to the typing rule used at the bottom of the derivation.

- sub-case **(TRec)**: typing derivation ends up with

$$\frac{\begin{array}{c}\Delta + [f : \forall\overline{\alpha}.(\overline{x:\pi}) \to \sigma] \cdot \Gamma + [\overline{x:\pi}] \cdot \Sigma \vdash e_1 : \sigma' \qquad \sigma'_{\restriction \Gamma + [\overline{x:\pi}]} \sqsubseteq \sigma \\ \texttt{CheckTermination}(\texttt{rec } f\langle\overline{\alpha}\rangle(\overline{x:\pi}) : \sigma = e_1) \\ \sigma = (\overline{\beta'\tau}, \epsilon, \gamma) \qquad \Delta + [f : \forall\overline{\alpha}.(\overline{x:\pi}) \to \sigma] \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma'' \qquad \overline{\alpha} \notin \Gamma \qquad \overline{\alpha} \notin \Delta\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{rec } f\langle\overline{\alpha}\rangle(\overline{x:\pi}) : \sigma = e_1 \texttt{ in } e_2 : \sigma''}$$

  By co-induction hypothesis

$$\mathcal{E}(\delta[f \mapsto (\overline{x:\pi}, e_1)]) \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow^{co} \texttt{div}$$

  In this case, a call to function f always terminates, hence this is not the source of divergence for the evaluation of expression e_2. In particular, f can be a ghost function. If that is the case, we have

$$\mathcal{E}(\delta[f \mapsto (\overline{x:\pi}, e_1)]) = \mathcal{E}(\delta)$$

  and the proof follows by the co-induction hypothesis, since the whole `rec..in` expression extracts to $\mathcal{E}_{\overline{\beta}}(e_2)$. Otherwise, f is a regular function and we have

$$\mathcal{E}(\delta[f \mapsto (\overline{x:\pi}, e_1)]) = \mathcal{E}(\delta)[f \mapsto (\mathcal{E}(\overline{x:\pi}), \mathcal{E}_{\overline{\beta'}}(e_1))]$$

  We can build the following derivation:

$$\frac{\mathcal{E}(\delta)[f \mapsto (\mathcal{E}(\overline{x:\pi}), \mathcal{E}_{\overline{\beta'}}(e_1))] \cdot \mathcal{E}(\mu) \cdot \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow^{co} \texttt{div}}{\mathcal{E}(\delta) \cdot \mathcal{E}(\mu) \cdot \texttt{rec } f\langle\overline{\alpha}\rangle(\mathcal{E}(\overline{x:\pi})) : \mathcal{E}(\sigma) = \mathcal{E}_{\overline{\beta'}}(e_1) \texttt{ in } \mathcal{E}_{\overline{\beta}}(e_2) \Downarrow^{co} \texttt{div}}$$

  which concludes this case.

- sub-case **(TRecDiv)**: typing derivation ends up with

$$\frac{\begin{array}{c}\Delta + [f : \forall\overline{\alpha}.(\overline{x:\pi}) \to \sigma] \cdot \Gamma + [\overline{x:\pi}] \cdot \Sigma \vdash e_1 : \sigma' \qquad \sigma'_{\restriction \Gamma + [\overline{x:\pi}]} \sqsubseteq \sigma \\ \neg\texttt{CheckTermination}(\texttt{rec } f\langle\overline{\alpha}\rangle(\overline{x:\pi}) : \sigma = e_1) \\ \sigma = (\overline{\beta\tau}, \epsilon + \texttt{div}, \gamma) \qquad \Delta + [f : \forall\overline{\alpha}.(\overline{x:\pi}) \to \sigma] \cdot \Gamma \cdot \Sigma \vdash e_2 : \sigma'' \qquad \overline{\alpha} \notin \Gamma \qquad \overline{\alpha} \notin \Delta\end{array}}{\Delta \cdot \Gamma \cdot \Sigma \vdash \texttt{rec } f\langle\overline{\alpha}\rangle(\overline{x:\pi}) : \sigma = e_1 \texttt{ in } e_2 : \sigma''}$$

  Since f is a stateful function, we can only apply the following rule of extraction:

$$\mathcal{E}_{\overline{\beta}}(\texttt{rec } f\langle\overline{\alpha}\rangle(\overline{x:\pi}) : \sigma = e_1 \texttt{ in } e_2 \triangleq \texttt{rec } f\langle\overline{\alpha}\rangle\ (\mathcal{E}((\overline{\beta'\tau}, \epsilon, \gamma))) : \mathcal{E}(\overline{\beta'\tau}, \epsilon, \gamma) = \mathcal{E}_{\overline{\beta'}}(e_1) \texttt{in } \mathcal{E}_{\overline{\beta}}(e_2))$$

  We complete this case by using the co-induction hypothesis and the rule **(CoEvalRec)** to build the derivation. This is similar to second sub-case of the **(CoEval)** case.

□

4.4 Extraction Machinery

In this section, we present the new Why3 extraction mechanism. We describe our implementation effort, with a focus on the new architecture of the mechanism, code optimizations, and interaction with drivers. We give, as well, an overview of the new Why3 `extract` command, with a detailed explanation of the command line options and different entry points for extraction.

4.4.1 Extraction Implementation.

We have re-implemented from scratch the Why3 extraction mechanism. This programming effort currently makes up to more than 2.5k of OCaml non-blank lines of code. We entirely re-designed the extraction mechanism architecture to clearly separate the code translation part from the code printing process. The WhyML internal abstract syntax tree is translated into an intermediate representation, an ML-like language, from which all the traces of ghost code and logical annotations have been removed. The implemented extraction function follows the definition we give in Fig. 4.1. Taking a regular WhyML expression and a mask as arguments, it produces an expression in the new intermediate representation. The following is an excerpt of the extraction function implementation:

OCaml
```
let rec extract mask e =
  assert (not (e_ghost e));
  assert (mask_sub e.e_mask mask));
  match e.e_node with
  | Econst _ | Evar _ when mask = MaskGhost  -> e_unit
  | Econst c                                 -> e_const c
  | Evar pv                                  -> e_var pv
  | Elet (LDvar (_, e1), e2) when e_ghost e1 -> extract mask e2
  | Elet (LDvar (_, e1), e2) when e_ghost e2 -> extract MaskGhost e1
    ...
```

Let us note the two `assert` expressions at the beginning of the function definition. These ensure that we never try to extract a ghost expression, and that the mask given as the mask of expression `e` is always a sub-mask, *i.e.*, it is *at most as ghost as* the mask given as an argument to the *extract* function. These assert expressions correspond exactly to the statement of lemma 4.1.1, and are used as mean of *defensive programming*. The four cases in the definition of `extract` that we present here are also present in the definition of Fig. 4.1. When `mask` is completely ghost, we extract constants and variables into `e_unit`, a smart constructor of the intermediate language that build the unit value. On the other hand, if the mask is visible, we call smart constructors `e_const` and `e_var` to extract, respectively, constants and program variables. Finally, the remaining two cases correspond to the first two cases in extraction of a `let..in` expression. In the first one, we extract only sub-expression e_2 since e_1 is ghost. In the second case, we extract e_1 with a ghost mask, as we are only interested in keeping the side effects produced by this expression and not its result. For both cases of `let..in` extraction, the assert expressions on top of the function body, together with the `when` clauses of each branch, ensure that each recursive call of `extract` is done on regular expressions. Moreover, we know that for the second case e_1 is an effectful expression, otherwise the whole `let..in` expression would be considered ghost.

To print the extracted code we use a *printer*, an OCaml program that takes the intermediate representation and prints it into the desired programming language syntax. In the scope of this work, we wrote the printer for the OCaml language. This is an essential ingredient in our quest to use Why3 to produce verified OCaml programs. We wrote, as well, a CakeML [93] printer, in order to translate verified WhyML programs into the input language of a certified compiler. Extending Why3 with the support for the extraction towards a new programming language is now just a matter of writing a new printer. Giving the simplicity of the intermediate representation, when compared with the WhyML internal AST, writing such a printer is not a major task. In fact, Why3 currently supports extraction of C programs [136], using a printer conceived on top of our intermediate representation.

Optimizations. During the code translation phase, we perform some optimizations over the extraction result. An important optimization is the simplification of singleton record types. When

reasoning about data structures, we normally introduce a logical model for such a data structure. Such a model is typically introduced as ghost value and maintained as a field of a record type which contains the data structure. Let us consider the case of type `t`, the of permutations presented in Sec. 2.2.1. When we remove the two ghost fields, we end up with the following type definition:

WhyML
```
type t = { a: array63 }
```

The Why3 extraction mechanism inlines such type definitions, and simplifies the code that accesses this field or that creates a value of this record-type. This avoids a dynamic allocation for a single-fielded record.

Another important optimization performed during extraction is the elimination of superfluous `let..in` expressions. These are mainly introduced by the internal representation of WhyML in A-normal form. Even if locally binding every sub-expression, via what we call a *proxy* variable, does not change the meaning of the program, it becomes rather tedious to read a code containing all those extra bindings. However, not all `let..in` bindings introduced internally by Why3 can be removed. Let us consider, for instance, the following WhyML program:

WhyML
```
let conflict_assign () : unit
= let f (x y: int63) : int63
  = assert { y = 42 };
    y in
  let i = ref 42 in
  let _ = f (i := 0; !i) !i in
  assert { !i = 0 }
```

In Why3, it is specified that the order of arguments evaluation is done from right to left. The internal use of A-normal form automatically guarantees that this order of evaluation is respected:

WhyML (internal)
```
let conflict_assign () : unit =
  let f (x y: int63) : int63
  = assert { y = 42 };
    y in
  let i = ref 42 in
  let _ = let o = !i in
    let o1 = i := 0; !i in
    f o1 o in
  assert { !i = 0 }
```

In particular, we are able to prove the two given intermediate assertions. Naively, we could have extracted this program into the following OCaml code:

OCaml (extracted)
```
let conflict_assign () : unit =
  let f (x: int) (y: int) : int = y in
  let i = ref 42 in
  ignore (f (i := 0; !i) !i)
```

However, the order in which function arguments are evaluated in OCaml is not specified. This means that some versions of the OCaml compiler might start by evaluating expression `i := 0; !i`. In particular, function `f` is called with both arguments equal to zero, which breaks the intermediate assertion proved by Why3.

Interaction with drivers. Extraction drivers introduce textual substitutions for some of the WhyML symbols used in our development. For each type, function, or exception name encountered

during extraction, we check if such a symbol is defined in the driver. If this is the case, we replace every occurrence of such a symbol by the code provided in the driver. In the particular case of extraction to OCaml, we use the `ocaml64` driver to map some of the Why3 standard library elements to the corresponding OCaml standard library counterparts. The following is a piece of this driver:

Driver

```
module list.List
  syntax type      list   "%1 list"
  syntax function  Nil    "[]"
  syntax function  Cons   "%1 :: %2"
  syntax predicate is_nil "%1 = []"
end
```

In the example above, we show we translate the WhyML `List` module into the corresponding OCaml syntax. The use of constructors `Nil` and `Cons` are replaced, respectively, by the OCaml syntactic sugar `[]` and `_::_`. The Boolean function `is_nil` is directly replaced by a test against the empty list. Each `%i` correspond to a place-holder for the symbol's arguments. Since `list` is a polymorphic type, `%1` corresponds to the type arguments.

In order to avoid some potential faulty expressions, we enclose within parentheses every expression issued from the driver. This is, however, clearly not enough to prevent every potential problem rising from the driver. Each string can contain arbitrary code that is never verified, nor even type-checked, which makes every substitution a potential source of bugs in extracted code. Drivers are, thus, part of the trusted computing base of Why3. Nonetheless, a typical driver is supposed to only introduce very simple substitutions, which we can convince ourselves that are the source of any problem just by carefully reading the driver's contents.

4.4.2 The `why3 extract` command.

To extract a program from a WhyML development, the user must invoke the `extract` tool of the Why3 framework, directly from her terminal. The general usage of this command is as follows:

Terminal

```
why3 extract [options] -D <driver> [-o <dir|file>] [<file>.<Module>*.<symbol>?|-]
```

The available command options are the following:

- `--flat` (this option is activated by default): instructs Why3 to print the result of extraction into a single file, even if multiple WhyML modules/files are extracted;
- `--modular`: instructs Why3 to print each WhyML module targeted by extraction into a separated file; we can use this option to extract functorial code, as we shall present in Sec. 5.3.4; such an option cannot be combined with the `--flat` one;
- `--recursive`: instructs Why3 to perform a recursive extraction, *i.e.*, the dependencies of each WhyML symbol are also extracted; this option can be combined with both the `--flat` and `--modular` ones.

To perform recursive extraction, we have implemented a calculus of dependencies for our intermediate representation tree. It consists of an higher-order iterator that traverses bottom-up the abstract syntax tree issued from extraction, using Why3 standard library functions to fetch the definition of a symbol from its name.

The `-D` option of the command line is mandatory and expects the name of a driver. A single call to `extract` can be given multiple drivers; it suffices to prefix each one with an extra `-D`. The `-o` option specifies the output file to print the extraction result. Under `flat` extraction, if no output

file is given, the result is printed to standard output. This option is mandatory under a `modular` extraction, and one must specify a directory in which to print the different resulting files.

Finally, we can call the Why3 extraction on different *entry points*. Extraction can be applied to a complete `.mlw` file, in which case the contents of the whole file is extracted. Extraction can also be applied to a single module, in which case we give to the command line an argument of the form `f.M`, where module `M` is contained in the Why3 file named `f.mlw`. Last, we can also chose to extract the definition of a single WhyML symbol, *i.e.*, a function, a type, or an exception name. In this case, the `extract` command terminates by `f.M.t`, where symbol `t` is defined inside module `M`, which is contained in the `f.mlw` file.

4.5 Discussion and Related Work

Correct-by-construction programs via code extraction. The Coq proof assistant provides an extraction mechanism [101, 102] that can be used to extract a certified functional program from a proof term. The basis for Coq's extraction is a clear separation between propositions that are "computational informative" and propositions that have only "logical" contents [122]. In practice, this is done by marking the computational-relevant elements of a proof with sort `Set`, and the parts that are useless for computation with sort `Prop`. Coq's type system ensures that this marking is correct, which is similar to our use of KidML's type system to ensure the property of non-interference. Coq currently supports extraction towards OCaml, Haskell, and the Scheme language.

The Coq extraction mechanism has been successfully used to produce correct-by-construction implementations of industrial-size software. An example is CompCert [99], a realistic, optimizing, and formally verified C compiler. Coq is used to prove that the different steps of compilation preserve the semantics of the C source code, as well as to extract an executable OCaml implementation of the compiler. Another impressive example that follows this approach is the Verasco static analyzer [82]. This analyzer uses abstract interpretation [45] to search for run-time errors in C programs. Verasco is proved correct using Coq, which guarantees that programs that analyze without alarms are free of run-time errors. The OCaml sources of this project are, once again, obtained via the Coq extraction mechanism.

Another example of a verification framework featuring an extraction mechanism for ML-like programs is the FoCaLiZe atelier [73]. The FoCaLiZe environment presents a functional programming language with object-oriented features, on which the programmer can write specifications and formal proofs of programs. Currently, FoCaLiZe translates a proof to three languages: OCaml, which lets the user build an executable program; and Coq and Dedukti [6], in order to check the proofs.

Formalization of extraction. The formalization of our extraction function is greatly influenced by the previous work of Jean-Christophe Filliâtre, Léon Gondelman [72, Chap.2], and Andrei Paskevich [64]. In that work, the authors define a code ghost erasure procedure over GhostML, a small ML-like language. This erasure function is parameterized by a Boolean value and a GhostML expression. Whenever the first parameter is bottom, the whole expression is converted to the unit value. Otherwise, an extracted program is produced as a morphism of the source program. This closely resembles our use of masks during extraction, where bottom corresponds to an entirely ghost mask and top to a regular one. A proof of typing and semantics preservation is given for such erasure proceeding. Contrary to our approach, these proofs are done using a small-step semantics judgment. Another important difference is that a ghost expression is systematically replaced by the unit value, while our extraction function completely erases any traces of ghost code, whenever

possible. Moreover, an expression is either completely ghost or completely regular, which simplifies the design and soundness proof of the erasure procedure.

Differences with respect to the Why3 extraction. The extraction mechanism we present in this chapter is a significative sub-set of what we have actually implemented inside the Why3 framework. Our presentation of the extraction function closely follows the OCaml implementation that lives inside the Why3 source code. In the following, we detail on some of the implemented features that are missing in our formalized extraction function.

There are some WhyML syntactic constructions that we do not include in our presentation. The most evident is pattern matching and algebraic data types declaration. We chose to exclude pattern-matching as we believe this would only make our formalization heavier, without actually making it more interesting. Another distinguished feature of WhyML is its module system. The modular structure of a development can be replicated in the extracted code, via the `modular` option of the `extract` command. It could extend our extraction function to take into account the modular structure of a WhyML development, and it would be interesting to investigate how we would generalize our proof of semantic preservation to take non-closed programs into account.

An important difference between the implemented extraction mechanism and the one we present in this thesis is partial application of functions. While the KidML language only features complete application, WhyML allows the programmer to partially apply functional symbols. From a point of view of code extraction, this poses some interesting challenges. Let us consider the following WhyML program:

WhyML
```
let last_ghost_arg () =
  let f (x: int63) (ghost y: int63) = x / 0 in
  let partial = f 0 in
  42
```

Naively, we could expect the following extracted code:

OCaml (extracted)
```
let last_ghost_arg () =
  let f (x: int) = x / 0 in
  let partial = f 0 in
  42
```

At first view, and given the ghost status of argument `y`, this seems a fair result of extraction. However, the partial application of `f` must be considered carefully. The above WhyML program produces no run-time error, since `f` is never completely applied, hence we never evaluate the division by zero expression. On the other hand, `f` is extracted as a single-argument function thus, when expression `f 0` is evaluated, a division-by-zero error is raised. The extracted code cannot introduce any behavior that does not happen in the original WhyML code. To avoid this pitfall, whenever a WhyML function is partially applied and the remaining arguments are all ghost arguments, we encapsulate such an application with a function expecting a unit argument. For the above example, this is as follows:

OCaml (extracted)
```
let partial () = f 0 in
...
```

It would be interesting to investigate how we could extend our language and extraction mechanism to include partial function application, and observe how this would impact our formalization.

No one in the brief history of computing has ever written a piece of perfect software. It's unlikely that you'll be the first.

Andy Hunt

5

A Toolchain for Verified OCaml Programs

In this chapter, we present a new toolchain and methodology to produce verified OCaml programs. The core of our methodology is based on the material presented in the previous chapters, namely the use of the WhyML language to derive verified implementations, and the use of the Why3 extraction mechanism to obtain correct-by-construction OCaml code.

Our toolchain is part of a bigger project, named VOCaL [31], whose ambition is to provide a mechanically-verified library of efficient general-purpose data structures and algorithms, written in the OCaml language. One novelty of VOCaL[1] is the collaborative use of three different verification tools, namely Coq [140], the CFML tool [29], and Why3 itself. To reconcile these three tools together, given their distinct logic and programming frameworks, one of the main lines of work in the VOCaL project is the design of a specification language for OCaml, similar to what JML is for Java [24, 96], or ACSL for C [15]. Such a specification language is not tied to any particular verification tool, hence it can be used as an entry point for the three tools of the VOCaL project. An important aspect about this specification language is that it must be both mathematically rigorous and easy to understand by an OCaml programmer who is not necessarily a proof expert. This last point is crucial, since we attach specification elements to `.mli` files, which are traditionally used as documentation units on the behavior of the library code, as the user is not expected to look into the details of implementation. In such a way, the introduced specification cannot disturb the library user, but rather be a source of extra rigorous documentation, helping the programmer to understand the behavior of the library operations. In the following, we shall refer to this new specification as OSL, short for *OCaml Specification Language*.

This chapter is organized as follows. We present our methodology and toolchain in Sec. 5.1. In Sec. 5.2, we use the example of a verified union-find implementation as a complete illustration of all the steps in our methodology. Sec. 5.3 completes the union-find case study by presenting challenging aspects that arise during the proof of OCaml programs, and how we deal with them. We conclude with some discussion and related work in Sec. 5.5.

[1] `https://vocal.lri.fr/`

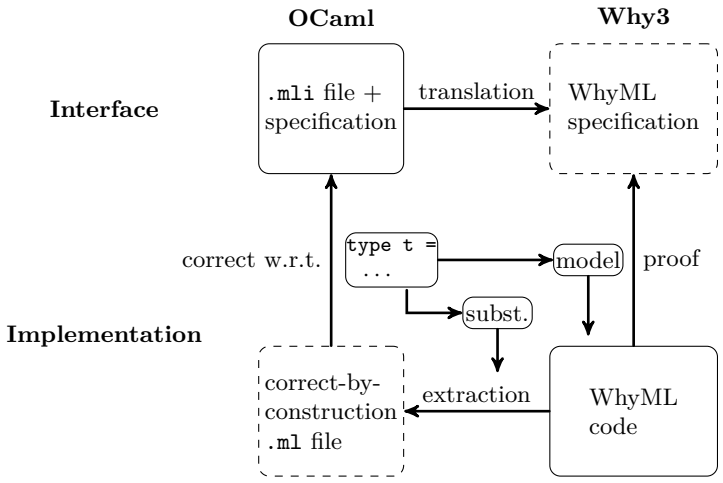


Figure 5.1: Methodology diagram.

5.1 Methodology

There are various different approaches to tackle the development of formally verified programs. The most direct and, perhaps, most widespread method is to augment an existing mainstream programming language with specification annotations (function contracts, loop invariants, etc.) and prove the conformance of the code to the specification, normally through an intermediate verification platform. Some examples of such approach include the VeriFast [81] and KeY [2] tools for Java, the Frama-C [87] and VCC [38] (using Boogie [10] as an intermediate language) frameworks for the C language, GNATprove [39] (using Why3 as an intermediate language) for Ada/SPARK, and CFML (using Coq as an intermediate language) for OCaml. A major technical challenge resulting from this approach is the need to encode a significant fragment of a real-life programming language, which was most likely not designed with verification in mind, into a suitable program logic. Other than the hardness of defining such an encoding, the generated verification conditions may be rather complex which makes the proof difficult for both automated or interactive tools.

A different approach consists in developing formally verified code in a dedicated verification environment and then translate it to an existing programming language. Via this translation mechanism one produces a so called *correct-by-construction* program. Following this approach, one can cite PVS [119], Coq , the B method [1], F* [139], Dafny [97], and Why3. This approach works well for self-contained developments, *e.g.*, the CompCert [99] verified C compiler, but faces some problems when it comes to integrate the verified code into a larger development. The automatically generated code is typically a clobbered mess while the original source code, written in the specific language of the verification platform, is normally incomprehensible to a common programmer.

Our methodology to use the Why3 framework to producing verified OCaml code is an attempt to reconcile these two approaches, avoiding some of the aforementioned drawbacks. The OCaml language has a number of features that make it a particularly well-suited target language for our approach. The separate compilation in OCaml is organized around the notions of *interface*, declarations of types and function signatures collected in a `.mli` file, and an *implementation* providing the definitions of types and functions, collected in a `.ml` file. We split our verification and implementation process into several steps. First, given an OCaml `.mli` file, we annotate declarations with specification elements such as function contracts, type invariants, etc., using the new OCaml specification language. From the annotated `.mli` file, we then automatically generate a corresponding Why3 input file, in which all annotations are translated into WhyML. This is done

using a new Why3 plugin, which we specifically developed as part of our methodology. The next step is to provide a verified WhyML implementation of the declared operations and types. This means that, besides implementing and verifying a WhyML program, we also establish its correctness with respect to the specification we first gave in the `.mli` file. Lastly, we use the Why3 extraction mechanism to translate our verified WhyML implementation into a correct-by-construction OCaml code, which we print to a `.ml` file.

An overview of our methodology is given in Fig. 5.1. In the diagram, the rows correspond to the different levels of abstraction (interface vs. implementation) and the columns correspond to environments (OCaml vs. Why3). The solid rectangles represent the user-written files, namely the annotated OCaml interface and the WhyML implementation, and the dashed rectangles represent the automatically generated files, namely the WhyML interface and the OCaml implementation. Whenever an OCaml type cannot be mapped directly to a WhyML type, due to use of mutable data beyond the reach of WhyML's type system [63], a custom memory model is built for this type. When it comes to translation of WhyML to OCaml, we return to the original OCaml type using a consistent substitution file. This is illustrated in the central part of the diagram.

In the remaining of this chapter, we go through several case studies in order to illustrate our methodology. We use the example of a union-find library to explain our verification and implementation workflow in detail. Other case studies are presented in Sec. 5.3.

5.2 A Case Study: Union-Find

To illustrate our methodology, let us consider the verification of a union-find library. Such a case study is a collaboratively development with Martin Clochard, Jean-Christophe Filliâtre, and Simão Melo de Sousa. We reuse the OCaml API from Arthur Charguéraud and François Pottier's proof [32]:

OCaml

```
type 'a elem                              (* type of the elements *)
val make  : 'a -> 'a elem                 (* a singleton class    *)
val get   : 'a elem -> 'a                 (* access the image     *)
val set   : 'a elem -> 'a -> unit         (* update the image     *)
val find  : 'a elem -> 'a elem            (* the representative   *)
val eq    : 'a elem -> 'a elem -> bool    (* in the same class?   *)
val union : 'a elem -> 'a elem -> unit    (* merge two classes    *)
```

We assume the interface to be enclosed in a file named `UnionFind.mli`. Polymorphic type `elem` stands for the type of elements of each equivalence class. In this API, a value of type `'a` is attached to every equivalence class. The `make` function takes a value of such type and returns a fresh `elem`, representing a new singleton class. Functions `get` and `set` are used, respectively, to retrieve and modify the information attached to an equivalent class. The return type of the `set` operation indicates that the equivalence class is updated via a side-effect, which lets us guess that function `set` performs some effectful computation. The remaining functions in the API are the expected ones for a union-find implementation: the classical function `find` takes an element of and returns the representative of the equivalent class to which the element belongs; function `eq` checks if two elements belong to the same equivalence class; finally, `union` merges two equivalence classes, also via a side-effect.

In the following sections, we detail on the OCaml and WhyML developments that we use to produce a correct-by-construction implementation of the union-find library. We show how we specify the interface of union-find, using the new OCaml specification language, and how we can use Why3 to build a verified implementation of the union-find operations. We prove that such an implementation conforms to the specified interface, through a proof of data refinement. Finally, we

use the Why3 extraction mechanism to mechanically generate a compilable OCaml implementation of the union-find library.

5.2.1 Specification

"With great proofs, comes great specification". Before engaging ourselves in the proof of the union-find implementation, we must first devise a formal specification for the union-find operations. Such a specification is added to the `UnionFind.mli` file, in a form of special comments starting with '`@`'. Being declared inside comments, specification elements are ignored by the OCaml compiler.

In order to specify to the API operations, we need to be able to reason about the elements in the union-find universe. To this end, we first introduce the following *ghost* type `uf`:

OCaml (annotated)

```
(*@ type 'a uf
      mutable model dom : 'a elem set
      mutable model rep : 'a elem -> 'a elem
      mutable model img : 'a elem -> 'a
      invariant forall x. mem x dom -> img x = img (rep x)
      invariant forall x. mem x dom -> rep x = rep (rep x)
      invariant forall x. mem x dom -> mem (rep x) dom *)
```

This is a mutable, abstract data type, whose contents we model through a set `dom`, a function `rep`, and a function `img`. We add, as well, three invariants to type `uf`. These ensure that the set `dom` is, indeed, partitioned by the relation "to have the same values by `rep`". In the following, we use type `uf` to derive the specification of the union-find operations.

We use a subset of the operations declared in the `UnionFind.mli` interface to illustrate the use of OSL. Let us use `make` as a first example. Such an operation is specified in OSL as follows:

OCaml (annotated)

```
val make : 'a -> 'a elem
(*@ e = make [uf: 'a uf] v
      modifies uf
      ensures  not (mem e (old (dom uf)))
      ensures  dom uf = old (dom uf) `union` {e}
      ensures  rep uf = (old (rep uf))[e <- e]
      ensures  img uf = (old (img uf))[e <- v] *)
```

A function specification is attached to a `val` symbol. The first line names the argument `v` and the returned value `e`, so that we can refer to them in the function contract. For the purpose of specification, function `make` receives an extra argument `uf` of type `uf 'a`, which we identify as a ghost argument using square brackets. The function contract is given in the form of pre- and postconditions, as well as `modifies` clauses to indicate a modification effect performed by the function. In this case, `modifies` accounts for the fact that we update the logical models `dom`, `rep`, and `img`, following the creation of a new equivalence class. Explicitly mentioning the modification effects of functions gives the programmer some extra understanding about the execution of the function, which cannot be stated in the OCaml interface. In the case of `make`, this tells the programmer that a call to `make` produces some side-effect. The postcondition states four different properties: (1) the element `e` did not belong to set `dom` before the function execution (the `old` keyword represents the value of its argument in the state prior to function execution), *i.e.*, we create, indeed, a fresh equivalence class; (2) we add `e` to set `dom`; (3) we extend `rep` with `e` as the representative element of the new equivalence class; (4) we extend `img` with `v` as the image of the new equivalence class. Note that we use the name of models as projections over the value `uf`.

Next, we define the following specification for the `find` function:

```
val find : 'a elem -> 'a elem                                    OCaml (annotated)
(*@ r = find [uf: 'a uf] e
      requires mem e (dom uf)
      modifies uf
      ensures  dom uf = old (dom uf)
      ensures  rep uf = old (rep uf)
      ensures  img uf = old (img uf)
      ensures  r = rep uf e *)
```

The above pre-condition requires that element `e` to be in the `dom` set, *i.e.*, the domain of elements in the union-find universe. The postcondition ensures that the `find` operation does not change the value of the logical models, and also that the returned value `r` stands for the representative element of the equivalence class of `e`. The most interesting part of this specification lies in the `modifies` clause. Even if we do not change the value of any of the models of type `uf`, function `find` internally performs a side-effect. This is nothing else than the effect of *path-compression*.

We complete our presentation of the union-find specification with the `union` function. This function is specified as follows:

```
(*@ predicate equiv (uf: 'a uf) (x: 'a elem) (y: 'a elem) :=     OCaml (annotated)
      rep uf x = rep uf y *)

val union : 'a elem -> 'a elem -> unit
(*@ union [uf: 'a uf] e1 e2
      requires mem e1 (dom uf)
      requires mem e2 (dom uf)
      modifies uf
      ensures  dom uf = old (dom uf)
      ensures  exists r. (r = old (rep uf e1) || r = old (rep uf e2)) &&
           forall x. mem x uf.dom ->
           rep uf x = (if old (equiv uf x e1 || equiv uf x e2) then r
                       else old (rep uf x))
        && img uf x = if old (equiv uf x e1 || equiv uf x e2) then img uf r
                      else old (img uf x) *)
```

We can only merge two existing elements, which we state in the pre-condition. When merging two equivalence classes, we do not allocate any new element, as stated in the first postcondition. The remaining of the `union` postcondition demands a little more of attention. The `union` operation takes two equivalence classes and shall map every element in the classes of `e1` and `e2` to the same representative value. The representative element of the merged class is either the representative value of `e1` or the representative value of `e2`, before the merging. We retrieve such an element using the existential quantification in the postcondition. We can then specify that, for every element `x` in the `dom` set, if `x` is in the same equivalence class as `e1` or `e2`, then `rep uf x` is updated to `r`. Otherwise, `x` just keeps the same representative as before. We update function `img` in a similar way. Finally, the `modifies` clause accounts for path-compression and the connection of elements to a new representative value.

Functions `get`, `set`, and `eq`, which we do not show here, have similar contracts to those of functions `make`, `find`, and `union`.

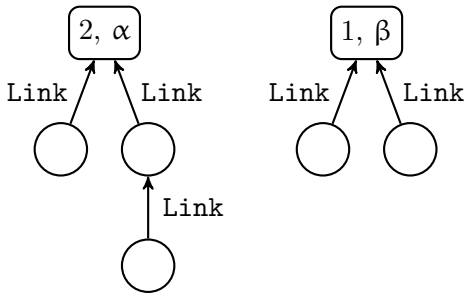
5.2.2 Verified Implementation

The next step is to implement and verify the union-find data structure. The OCaml implementation we target is based on the following data types:

```
type 'a content = Link of 'a elem | Root of int * 'a                    OCaml
and  'a elem    = 'a content ref
```

Each element is either a representative element (`Root`), containing a value of type `int`, which we refer to as the *rank* of the class, and a value of type `'a`, or a pointer (`Link`) to another element in the same equivalence class.

The picture on the right shows a possible state of the union-find universe, built from the given data types. The rectangular nodes stand for `Root` elements (the representatives of classes), whereas the round ones stand for `Link`. The leftmost class is of rank two, as the longest chain until the root features two `Link` nodes. We represent the information associated to each equivalence class using symbols α and β.



Memory model. Unfortunately, the given OCaml type definition cannot be directly translated to WhyML. The reason is that recursive mutable types are beyond the scope of Why3's type-and-effect discipline [63]. The solution is to resort to an explicit memory model, that is a set of types for pointers and memory together with the operations to allocate, read, and write memory. In this case, we introduce the following WhyML type to model the heaps contents:

```
type mem_ref 'a = private {                                              WhyML
  ghost mutable refs : loc_ref 'a -> option 'a;
}
```

where `loc_ref` is an abstract type to represent locations of OCaml's heap-allocated references of type `ref`. In this type of the heap, non-allocated locations are mapped to `None`, and each allocated location is mapped to `Some c`, for some value of type `'a`.

We declare some primitive functions to manipulate the heap and locations. For instance, the following returns a fresh and empty heap:

```
val ghost empty_memory () : mem_ref 'a                                   WhyML
  ensures { forall r. not (mem r result.dom) }
```

The specification of function `empty_memory` states that no location is allocated in the newly created heap. This is very easily expressed using set `dom`. To allocate a new reference, we use the following function:

```
val alloc_ref (ghost memory: mem_ref 'a) (v: 'a) : loc_ref 'a            WhyML
  writes  { memory }
  ensures { not (mem result (old memory).dom) }
  ensures { memory.refs = Map.set (old memory.refs) result (Some v) }
```

Let us note that, contrarily to `empty_memory`, function `alloc_ref` is not declared as a ghost function. This is due to the fact that `empty_memory` is used exclusively in ghost code, while we need to use the `alloc_ref` function within regular code of our union-find implementation. At extraction time, an expression of the form `alloc_ref m v` is replaced by `ref v`, the OCaml standard library function to create new references. Sec. 5.2.4 presents the details on how to the translate our memory model and type of locations to executable OCaml code.

To interact with the heap, we introduce functions `get_ref` and `set_ref` which, respectively, get the current value associated to a location and update the value pointed by a reference in the memory. These are specified as follows:

```
val get_ref (ghost memory: mem_ref 'a) (l: loc_ref 'a) : 'a                    WhyML
  requires { mem l memory.dom }
  ensures  { Some result = memory.refs[l] }

val set_ref (ghost memory: mem_ref 'a) (l: loc_ref 'a) (c: 'a) : unit
  requires { mem l memory.dom }
  writes   { memory }
  ensures  { memory.refs = (old memory.refs)[l <- Some c] }
```

Note that we pass the heap as a ghost argument, instead of declaring a single global variable to model the heap. The reason is two-fold. First, global variables must have monomorphic types. Second, by using "small heaps" passed through the chain of function calls as hidden arguments, we statically enforce separation between the heaps and avoid complicated frame conditions. This is yet another instance of Burstall's "component-as-array". Soundness of heap manipulation is guaranteed by the fact that `mem_ref` is a private data type that is updated through abstract functions.

Finally, we introduce the following function to compare the memory address of two locations:

```
val (==) (x y: loc_ref 'a) : bool                                              WhyML
  ensures { result <-> x=y }
```

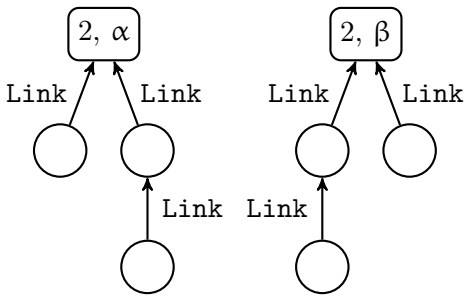
Different flavors of machine arithmetic. Having defined a particular memory model to our union-find implementation, we can now give a WhyML definition for types `content` and `elem`. This is as follows:

```
type rank = int63                                                              WhyML

type content 'a = Link (elem 'a) | Root rank 'a
type elem 'a    =  loc_ref (content 'a)
```

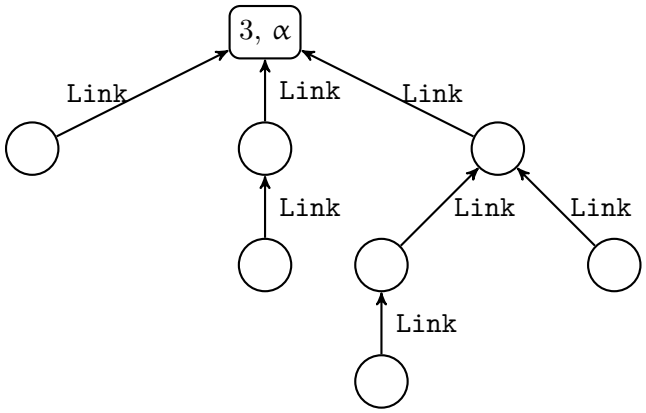
We use the `loc_ref` type of the memory model to represent references, and instantiate it here with an argument of type `content 'a`.

The interesting aspect about the above type definitions is the definition of type `rank`. This is an alias for type `int63`, the type of WhyML machine integers. As we describe in Chap. 2, type `int63` is directly extracted to the `int` type of OCaml, producing efficient code. From a proof perspective, however, the price to pay is to prove the absence of arithmetic overflows.

In Chap. 2, the proof that reference `n` does not overflow is made simpler because its value is bound by the length of the array `t`. There are, however, some situations where it is much more subtle to deal with machine arithmetic. Let us consider, for instance, the following two equivalence classes:



Performing a `union` operation over such equivalence classes, results in the following configuration of the union-find universe:



There is now a single equivalence class of rank three, which results from linking the root of the former rightmost class to the root of the leftmost one. In WhyML, we can implement the code performing this `union` operation as follows:

WhyML

```
let union (ghost memo: mem_ref (content 'a)) (x y: elem 'a) =
  let x, y = find memo x, find memo y in
  if x == y then x
  else match get_ref memo x, get_ref memo y with
    | Root rx vx, Root ry vy ->
        if rx = ry then begin
          set_ref memo y (Link x);
          set_ref memo x (Root (rx + 1) vx) end
        ...
```

Since the rank is of type `int63`, from such an operation Why3 generates the verification condition $\texttt{rx} + 1 \leq 2^{62} - 1$. Without any further knowledge on `rx`, this is simply not provable. One obvious solution would be to add such an inequality as a pre-condition of `union`. This would make the proof of overflow absence trivial, but this extra pre-condition would fatally pollute the proof of any client of `union`. Another solution would be to use arbitrary-precision integers, which would not require a proof of overflow absence. However, this would make the extracted code less efficient and would require the use of an external arbitrary-precision library to run the extracted code.

We adopt here the solution proposed by Martin Clochard, Jean-Christophe Filliâtre, and Andrei Paskevich [37]. The idea is as follows: unless we expect our program to have century-long runs, we can rest assured that an integer variable, whose value only grows by one at a time is, for all intents and purposes, safe from overflow. To materialize this meta-argument, the authors introduce a new type of integers, called *Peano integers*, with limited arithmetic operations. In Why3, this results in a new library `Peano`, with a type `t` for Peano integers, a zero constant `zero`, a successor function `succ`, and comparison function `lt` and `eq`, as follows:

WhyML

```
module Peano
  type t = abstract { v: int }

  val constant zero : t
    ensures { result.v = 0 }

  val succ (x: t) : t
    ensures { result.v = x.v + 1 }
```

```
  val lt (x y: t) : bool
    ensures { result <-> x.v < y.v }

  val eq (x y: t) : bool
    ensures { result <-> x.v = y.v }
end
```

We forgo the non-overflow precondition for `Peano.succ`, as there is no other way of producing a Peano integer than by starting at zero and incrementing it one by one, and so the 2^{62} limit is never reached in any real-life situation. In our verified implementation of union-find, we alias type `rank` with `Peano.t`, instead of `int63`:

```
type rank = Peano.t
```
WhyML

and change the implementation of `union` accordingly:

```
let union (ghost memo: mem_ref (content 'a)) (x y: elem 'a) =
  let x, y = find memo x, find memo y in
  if x == y then x
  else match get_ref memo x, get_ref memo y with
    | Root rx vx, Root ry vy ->
        if rx = ry then begin
          set_ref memo y (Link x);
          set_ref memo x (Root (Peano.succ rx) vx) end
```
WhyML

Since `Peano.succ` has no precondition, there is no proof of absence of overflow to be done. In particular, no extra pre-condition is required for function `union`.

When it comes to extract OCaml code, type `Peano.t` is translated into OCaml's type `int`, function `Peano.succ` into a machine addition, and `lt` and `eq` are translated, respectively, into the OCaml operations `<` and `=`. This is done by extending our `ocaml64` driver with the following:

```
module Peano
  syntax type t    "int"
  syntax val  succ "%1 + 1"
  syntax val  lt   "%1 < %2"
  syntax val  eq   "%1 = %2"
end
```
Driver

Implementation of data type `uf`. Having defined the memory model and the types `element` and `content`, we can implement and verify the union-find data structure. In particular, we have to implement the data type `uf`. It is a record data type that contains, in addition to the fields `dom`, `rep`, and `img`, the contents of the memory allocated by the union-find structure:

```
type uf 'a = {
  ghost mutable dom : set (elem 'a);
  ghost mutable rep : elem 'a -> elem 'a;
  ghost mutable img : elem 'a -> 'a;
  ghost         memo: mem_ref (content 'a);
}
```
WhyML

Encapsulation of the set of allocated pointers in type `uf` closely resembles the treatment of memory footprint in the context of dynamic frames [84–86]. Note that type `uf` is only useful for specification

purposes, as it contains only ghost fields. In order to specify the type `uf`, we keep the same type invariants as declared in `UnionFind.mli` interface:

```
invariant { forall x. mem x dom -> img x = img (rep x) }
invariant { forall x. mem x dom -> rep (rep x) = rep x }
invariant { forall x. mem x dom -> mem (rep x) dom }
```
WhyML

and add some new conditions to relate the contents of `memo` with those of set `dom`, and functions `rep` and `img`. First, we state that an `element` belongs to the union-find universe if and only if it is allocated,

```
invariant { forall x. mem x dom <-> allocated memo x }
```
WhyML

where `allocated` is defined as follows:

```
predicate allocated (memory: mem_ref (content 'a)) (x: elem 'a) =
  mem x memory.dom
```
WhyML

Next, we say that representative values are only allocated as `Root` elements:

```
invariant { forall x. mem x dom -> match memo.refs (rep x) with
              | Some (Root _ _) -> true
              | _               -> false end }
```
WhyML

Finally, every allocated element is part of a union-find chain:

```
invariant { forall x. match memo.refs x with
              | Some (Link y)   -> x <> y /\ allocated memo y /\ rep x = rep y
              | Some (Root _ v) -> img x = v /\ rep x = x
              | None            -> true end }
```
WhyML

If `x` points to a `Link y` node in memory, then `y` is different of `x` (no loops are allowed in union-find), element `y` is also allocated, and has the representative element as `x`; on the other, if `x` points to a `Root _ v` node, then the `img` of `x` is `v`, and `x` is its own representative element. This completes the specification of the union-find universe, with respect to the built memory model.

Verified implementation of union-find operations. After implementing the `uf` data type, we can now focus on the implementation of the union-find operations. As declared in the interface, all union-find functions receive a ghost parameter of type `uf` and then exploit it to perform read/write operations on memory. For instance, the `make` function is implemented and specified as follows:

```
let make (ghost uf: uf 'a) (v: 'a) : elem 'a
  ensures { not (mem result (old (dom uf))) }
  ensures { dom uf = add result (old (dom uf)) }
  ensures { rep uf = (old (rep uf))[result <- result] }
  ensures { img uf = (old (img uf))[result <- v] }
= let x = alloc_ref uf.memo (Root Peano.zero v) in
  uf.dom <- add x uf.dom;
  uf.rep <- uf.rep[x <- x];
  uf.img <- uf.img[x <- v];
  x
```
WhyML

We keep the same specification as given in the interface. The implementation of `make` is straightforward. More interestingly, we implement and specify function `find` as follows:

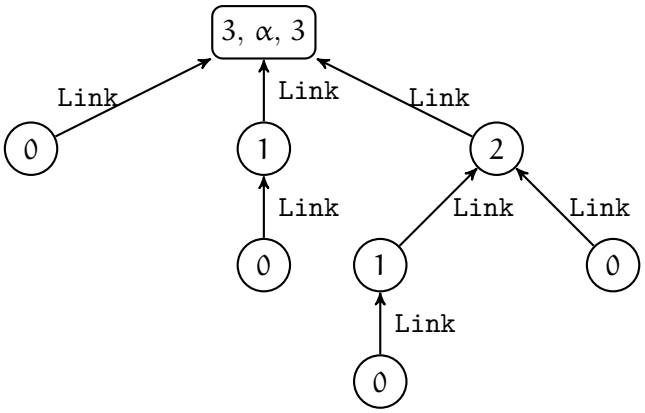
```
let rec find (ghost uf: uf 'a) (x: elem 'a) : elem 'a
  requires { mem x uf.dom }
```
WhyML

```
    ensures  { result = uf.rep x }
  = match get_ref uf.memo x with
    | Root _ _ -> x
    | Link y   -> let rx = find uf y in
                  set_ref uf.memo x (Link rx); rx
    end
```

Here, the specification is much simpler than the one we gave in the interface. Why3 is able to infer that the only field of `uf` modified by the execution of `find` is `memo`. There is, hence, no need to state that the values of `dom`, `rep`, and `img` are unchanged. The call to `set_ref` accounts for path compression, as it links every element in the equivalence class directly to the representative element computed by the recursive calls.

Proof of termination. Let us note that we have not, yet, supplied a termination measure for function `find`. In fact, proving the termination of this function is one of most challenging aspects in the proof of the union-find data structure.

We adopt here a solution that consists in maintaining a *distance* value associated to each element of an equivalence class. When following `Link` pointers, distances strictly increase, as we illustrate in the following diagram:



The distance values we assign to each element are of no particular interest, as long as they increase until we reach a `Root` node. We introduce, as well, a *maximal* value as an upper bound for distances. We materialize these two notions by adding to type `uf` two new fields `dst` and `maxd`, as follows:

```
type uf 'a = {                                                    WhyML
  ...
  ghost mutable dst : elem 'a -> int;
  ghost mutable maxd: int;
}
```

We also equip this data type with the following new invariants:

```
  invariant { forall x. match memo.refs x with                    WhyML
                | Some (Link y) -> dst x < dst y
                | _             -> true end }
  invariant { 0 <= maxd }
  invariant { forall x. mem x dom -> dst x <= maxd }
```

Such invariants state the increasing property of the `dst` values and that these are limited by the value of `maxd`. We straightforwardly update function `make` by adding the line

```
  uf.dst <- uf.dst[x <- 0]}                                          WhyML
```

to its body. This creates a new singleton equivalence class, with a zero-distance element. This trivially respects the `uf` type invariant, since we state `maxd` to be a non-negative value.

We can now specify and prove the termination of the `find` operation. This is as simple as follows:

```
  let rec find (ghost uf: uf 'a) (x: elem 'a) : elem 'a              WhyML
    ...
    variant  { uf.maxd - uf.dst x }
    ensures  { uf.dst result >= uf.dst x }
  = ...
```

The variant `uf.maxd - uf.dst x` decreases at each recursive call due to the increasing property of `dst`. The implementation requires absolutely no change. When we do path compression, we do not change the `dst` value. Every element in the chain of `x` is now linked to the representative element, which respects the invariant, as all distances are still strictly increasing.

Using `find`, we can provide an implementation for the `union` operation. This demands some attention when it comes to update the values of `dst` and `maxd`, in order to respect the invariant of type `uf`. First, we implement the `union` function as follows:

```
  let union (ghost uf: uf 'a) (x y: elem 'a) : unit                  WhyML
    requires { mem x uf.dom }
    requires { mem y uf.dom }
    ensures  { exists r. (r = old (rep uf x)) || (r = old (rep uf y)) /\
               forall z. mem z uf.dom ->
                 rep uf z = if old (equiv uf z x || equiv uf z y) then r
                            else old (rep uf z) /\
               forall z. mem z uf.dom ->
                 img uf z = if old (equiv uf z x || equiv uf z y) then img uf r
                            else old (img uf z) }
  = let (a, b) = find uf x, find uf y in
    link uf a b
```

We keep the exact same specification as in the `UnionFind.mli` interface file. The code of `union` first calls `find` to retrieve the representative elements `a` and `b` of the classes of `x` and `y`. Then, the auxiliary function `link` merges the classes of elements `a` and `b`. The implementation of `link` is where "all the fun happens". This function is responsible for updating the values of the logical models in `uf`, as well as the pointers in the heap. We give the following specification to such a function:

```
  let link (ghost uf: uf 'a)  (x y: elem 'a) : unit                  WhyML
    requires { mem x uf.dom /\ mem y uf.dom }
    requires { x = uf.rep x /\ y = uf.rep y }
    ensures  { exists r. (r = old (rep uf x)) || (r = old (rep uf y)) /\
               forall z. mem z uf.dom ->
                 rep uf z = if old (equiv uf z x || equiv uf z y) then r
                            else old (rep uf z) /\
               forall z. mem z uf.dom ->
                 img uf z = if old (equiv uf z x || equiv uf z y) then img uf r
                            else old (img uf z) }
```

This is very similar to the specification of function `union`, except for the pre-condition stating that `x` and `y` must be representative elements. The implementation is as follows: we first compare elements `x` and `y`, and if these point to the same location in memory we immediately return `x`,

```
= if x == y then x                                                    WhyML
```

on the other hand, if these are different pointers, we use function `get_ref` to get the value associated to `x` and `y`. If `x` and `y` are `Root` elements, then we merge the two equivalence classes according to the ranks. If `x` is the less-ranked representative element, then we link `x` to `y`, the representative element of the new class,

```
  | Root rx vx, Root ry vy ->                                          WhyML
      if Peano.lt rx ry then begin
        set_ref uf.memo x (Link y);
        uf.rep, uf.img, uf.maxd, uf.dst <-
          pure { fun z -> if uf.rep z = uf.rep x then y  else uf.rep z },
          pure { fun z -> if uf.rep z = uf.rep x then vy else uf.img z },
          uf.maxd + 1,
          uf.dst[y <- 1 + max (uf.dst x) (uf.dst y) ]
```

The merge of the two classes is done using the `set_ref` function, which changes in memory the value associated with `x`. The rest of the code updates fields `rep`, `img`, `maxd`, and `dst`. Syntax

```
x.f1,..., x.fn <- v1,..., vn                                          WhyML
```

stands for the *multiple assignment* operation, *i.e.*, we assign each field $\texttt{f}_\texttt{i}$ of record `x` to the value $\texttt{v}_\texttt{i}$. We increment the value of `maxd` and update `dst` such that `dst y` is equal to the maximum between `dst x` and `dst y`, plus one. This way, all distances are kept strictly increasing and smaller to the value of `maxd`, hence we preserve the invariant of type `uf`. The other cases in the pattern matching, *i.e.*, at least one of the elements is a `Link` node, represents an unreachable point in the code,

```
  else match get_ref uf.memo x, get_ref uf.memo y with               WhyML
    | Link _, Link _ -> absurd
```

which we can prove thanks to the extra supplied pre-condition.

The remaining of function `link` links `y` to `x`, and updates the fields of `uf` accordingly:

```
      end else begin                                                  WhyML
        set_ref uf.memo y (Link x);
        uf.rep, uf.img, uf.maxd, uf.dst <-
          pure { fun z -> if uf.rep z = uf.rep y then x  else uf.rep z },
          pure { fun z -> if uf.rep z = uf.rep y then vx else uf.img z },
          uf.maxd + 1,
          uf.dst[x <- 1 + MinMax.max (uf.dst x) (uf.dst y)];
        if Peano.eq rx ry then set_ref uf.memo x (Root (Peano.succ rx) vx)
      end
  end
```

If `x` and `y` are equally-ranked root elements, we know that the new class must be of an higher rank. In such a case, we update the value associated with `x` in memory to `Root (Peano.succ rx) vx`. This completes the proof of functions `union` and `link`.

We omit the implementation of functions `get`, `set`, and `eq`. These have all straightforward implementations and the contract of these functions follows exactly the one we give in the interface file. It is worth pointing out that all the generated verification conditions by Why3 for the

`union-find` operations are discharged fully automatically. No auxiliary lemma or intermediate assertion is ever needed.

A digression on specification styles. The way we specify the `uf` data type is crucial to complete the proof of the union-find data structure. The use of universal quantification and the specification in terms of model fields `dom`, `rep`, `img`, `dst`, and `maxd` is the secret to achieve a fully automated proof of correction of the implemented operations. In particular, proving termination is almost trivial thanks to the use of fields `maxd` and `dst`. We refer to such kind of invariants as *local invariants*.

We could, very well, specify type `uf` in terms of some *inductive* definition, introducing the notion of path and reachability between union-find elements. For instance, we could use the following definition of `path`:

```
inductive path (memo: memo 'a) (x y: elem 'a) (n: int) =                    WhyML
| Path_nil : forall mem: memo 'a, x: elem 'a, v: 'a, r, n.
             0 <= n -> memo.refs x = Some (Root r v) ->
             path memo x x n
| Path_cons: forall mem: memo 'a, x y z: elem 'a, n.
             0 <= n -> mem.refs x = Some (Link y) -> path mem y z n ->
             path mem x z (n+1)
```

The statement `path memo x y n` stands for "under heap `memo`, there is a path from `x` to `y` of length at most `n`". Then, we would have

```
forall x. mem x dom -> path memo x (rep x) maxd                              WhyML
```

as an invariant of the `uf` data type, which ensures the termination of function `find`. However, from a practical point of view, the use of inductive predicate `path` would fatally compromise the automation of the proof. At each assignment in the heap, we would have to re-establish the `path` invariant for every chain in the union-find universe (some are unchanged, some are shortened, etc.). This would require, undoubtedly, some auxiliary lemmas being proved by induction.

5.2.3 Proof of Refinement and Specification Inclusion

Once we have implemented and verified all operations, we move on to a proof of *specification inclusion*. This means proving that the specification of the verified union-find operations conforms to the specification written in the `.mli` file and translated to WhyML by the new Why3 plugin. A proof of specification inclusion is very similar to the use of Hoare Logic *consequence rule*. The following Hoare Logic rule

$$\frac{P \implies P' \qquad \{P'\}c\{Q'\} \qquad Q' \implies Q}{\{P\}c\{Q\}}\ \textsc{Consequence}$$

states that the Hoare triple $\{P\}c\{Q\}$ is valid if we can derive the Hoare triple $\{P'\}c\{Q'\}$, where P implies P and Q' implies Q. In fact, this rule allows us to strengthen the pre-condition and weaken the postcondition of program `c`. In a proof of specification inclusion, we lift such a reasoning to the level of functions. Let us consider, for instance, that we declare the following function `f` in the interface file

```
val f x1:τ1 ... xn:τn : τ                                                    OCaml (annotated)
(*@ r = f y1 ... ym
      requires P(y1, ..., ym)
      ensures  Q(y1, ..., ym, r) *)
```

and that we give the following implementation to `f`

```
let f (x'_1 : τ_1, ..., x'_n : τ_n) : τ                    WhyML
  requires { P'(x'_1, ..., x'_n) }
  ensures  { Q'(x'_1, ..., x'_n, result) }
= c
```

The specification inclusion proof of `f` amounts to proving the following implications:

$$\begin{aligned} P(\overline{x}) &\implies P'(\overline{x}) \\ Q'(\overline{x}, r) &\implies Q(\overline{x}, r) \end{aligned}$$

In Why3, a specification inclusion proof is done using the WhyML **clone** command. For the given example, we would write the following:

```
clone import Interface.Sig with val f = Impl.f                WhyML
```

where we assume function `f` is contained in file `interface.mli`, where `Sig` is the WhyML module automatically generated by the Why3 plugin, and the implementation of function `f` is contained in a module named `Impl`. Instead of directly generating the above verification conditions, Why3 generates the following piece of code:

```
let f_correct (x_1 : τ_1, ..., x_n : τ_n) : τ              WhyML
  requires { P(x_1, ..., x_n) }
  ensures  { Q(x_1, ..., x_n, result) }
= Impl.f (x_1, ..., x_n)
```

This way, it uses the Why3 verification conditions generator to generate the necessary verification conditions, avoiding the error-prone task of directly generating implications to prove specification inclusion.

In the case of our union-find WhyML development, the proof of inclusion is as simple as the following **clone** expression:

```
clone import UnionFind.Sig with                               WhyML
  type elem  = Impl.elem,
  type uf    = Impl.uf,
  val  make  = Impl.make,
  val  find  = Impl.find,
  val  eq    = Impl.eq ,
  val  get   = Impl.get,
  val  set   = Impl.set,
  val  union = Impl.union
```

For every symbol declared in the interface file `UnionFind.mli`, we give a substitution by the corresponding implementation symbol, which we assume to be defined in the WhyML module `Impl`. The result of the above **clone** statement is an instance of module `Sig`, modulo the given substitution, while generating verification conditions for specification inclusion.

Note that we also assign type symbols `elem` and `uf`, declared in file `UnionFind.mli`, with the corresponding implementations. If the types do not possess any invariant, this is just a textual substitution of the the type names. On the other hand, whenever types are equipped with some invariant, as it is the case of `uf`, there is a *refinement proof* to be done. Let us suppose that we have declared in the interface the following data type:

```
type t                                                        OCaml (annotated)
(*@ model f_1 : τ_1
    ...
    model f_n : τ_n
    invariant I(f_1, ..., f_n) *)
```

and have the following implementation of type `t`:

WhyML
```
type t = { f'1 : τ1; ... fn' : τn; }
  invariant { I'(f'1, ..., fn') }
```

In order to prove that the implementation of type `t` refines the specification given for type `t` in the interface, one must prove that the following condition holds:

$$\mathcal{I}'(\overline{\mathtt{f}'}) \implies \mathcal{I}(\overline{\mathtt{f}})$$

Other than generating the above verification condition, Why3 also checks that the sequence $\overline{\mathtt{f}}$ of fields declared in the interface is a sub-sequence of fields $\overline{\mathtt{f}'}$ given in the implementation, as well as the following additional condition on the ghost status of fields: a field declared as ghost in the interface can either be kept a ghost field in the implementation, or become a regular field; a field declared as regular can only be implemented as a regular field. In the case of type `uf`, the refinement proof is trivial since every field in the implementation keeps the same ghost status as declared in the interface, and the interface invariant is formed of three conjunctions that are also in the implementation invariant.

All the generated verification conditions for the specification inclusion and refinement proof of union-find are automatically discharged. This completes the specification, implementation and verification task of our methodology.

5.2.4 Extraction to OCaml

The last step in our methodology diagram consists in translating WhyML to OCaml, using Why3's extraction mechanism. From our memory model, we build the following custom driver:

Driver
```
module UnionFind.Impl
  syntax type loc_ref   "%1 ref"
  syntax val  (==)      "%1 == %2"
  syntax val  alloc_ref "ref %1"
  syntax val  get_ref   "!%1"
  syntax val  set_ref   "%1 := %2"
end
```

Note that the argument `%1` of functions in the driver does not refer to the heap. Indeed, because of its ghost status, this is erased by the extraction mechanism. The OCaml code extracted from the WhyML union-find implementation is given in Fig. 5.2.

With extraction, we close our methodology diagram. We can now use the OCaml compiler to compile the extracted code against the `UnionFind.mli` interface. From our toolchain point of view, this not only guarantees that the extracted code type checks with respect to what we declare in the interface file, but also that this implementation conforms to the specification included in `UnionFind.mli` file.

5.3 Challenges

We now illustrate our toolchain over several other examples. Each one presents a particular issue in the process of verifying OCaml code.

5.3.1 Non-verified Client Code

In the task of deductive verification, it is customary to annotate functions with some pre-conditions that prevent, for instance, run-time errors to happen during execution. Within the same formal

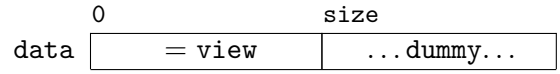
OCaml (extracted)

```
type 'a content =
| Link of ('a content) ref
| Root of int * 'a

let make (v: 'a) : ('a content) ref = let x = ref (Root (0, v)) in x

let rec find (x: ('a content) ref) : ('a content) ref =
  begin match !x with
  | Root (_, _) -> x
  | Link y -> let rx = find y in begin x := (Link rx); rx end
  end

let link (x: ('a content) ref) (y: ('a content) ref) : unit =
  if x == y then begin () end
  else
  begin
    begin match (!x, !y) with
    | (Root (rx, vx), Root (ry, vy)) ->
      if rx < ry then begin x := (Link y) end
      else
      begin
        begin
          y := (Link x); if rx = ry then begin x := (Root ((rx + 1), vx)) end
        end end
    | (_, _) -> assert false (* absurd *)
    end end

let union (x: ('a content) ref) (y: ('a content) ref) : unit =
  let a = find x in let b = find y in link a b
```

Figure 5.2: Union-find Extracted Code.

development, client code must respect these pre-conditions, which guarantees that every call to the verified function is safe. When we come out of our closed world of verified software, and translate a program proof into a compilable code with the purpose of distributing it, the panorama is completely different. Our verified implementation, that we so hardly obtained, can now be executed from a non-verified code base. In particular, some pre-conditions can be violated which might result in a run-time faulty behavior. In a perfect world, our correct-by-construction implementation should be defensive against non-verified client code, but without sacrificing the ability to call such an implementation from a verified piece of software.

In this section, we describe our proposal to deal with both verified and non-verified client code within our toolchain. Let us take the example of *vectors* [138, p. 136-137], also known as re-sizable or growing arrays, to illustrate this approach. A vector is a data structure that encapsulates an array which automatically expands or shrinks, when necessary. The following is a scheme of such a data structure:



The internal array is allocated with a certain capacity, which we might not be using entirely during

the lifetime of a vector. We say that there is a part of the vector filled with *significant* elements, which corresponds to the `view` section in the above scheme. We refer to the number of significant elements as the `size` of the vector. The rest of the internal array is filled with a `dummy` value. When one implements the vector data structure in OCaml, it is the `dummy` value that allows the GC to reclaim pointers of the

We can insert or remove elements from a vector, and the strategy of when to expand and when to shrink the internal array, is what makes vectors such an interesting data structure. A good implementation doubles the capacity when growing, and shrinks it (normally to half the capacity) whenever the number of elements is one-fourth of the array capacity. This way, inserting n elements in a vector takes $O(n)$ time, *i.e.*, each insertion operation has amortized constant time complexity (several proof techniques for such a complexity bound are given in Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein's *Introduction to Algorithms* textbook [43, chap. 17]). Given the use of a constant `dummy` value, when we shrink the internal array the OCaml's GC can reclaim pointers to values that aren't used anymore.

The following is a sub-set of a typical OCaml interface for the vector data structure:

OCaml

```
type 'a t

exception Empty

val create : ?capacity:int -> dummy:'a -> 'a t
val resize : 'a t -> int -> unit
val push   : 'a t -> 'a -> unit
val pop    : 'a t -> 'a
val get    : 'a t -> int -> 'a
val set    : 'a t -> int -> 'a -> unit
```

This interface file provides the polymorphic type `t` of vectors and the following operations: function `create` returns a zero-sized fresh vector, where the labeled argument `dummy` is the user-supplied value to fulfill the empty part of the array and the optional argument `capacity` stands for the initial capacity of the vector; function `resize` takes a vector `v` and an integer `n` as arguments and sets the size of `v` to `n`; function `push` inserts its second argument at the rightmost position of its first argument vector; finally, function `pop` removes and returns the rightmost element of a vector, or raises `Empty` if the vector contains no elements.

Following our methodology diagram, we provide a specification for the vector's interface. We give the following specification for type `t`:

OCaml (annotated)

```
type 'a t
(*@ mutable model view: 'a seq *)
(*@ invariant length view <= Sys.max_array_length *)
```

The vector data structure is modeled by a single mutable model field named `view`, exactly as in the previously shown scheme. This `view` field represents the sequence of significant elements of the vector. The sole restriction we place on this field is that its length never exceeds the `max_array_length` constant, defined in the OCaml System interface module. This limits the maximum length of the array, and this invariant guarantees that we never allocate an internal array bigger than `max_array_length`. The specification of `create` is straightforward, hence we do not detail it here. The specification of `resize` is more interesting:

OCaml (annotated)

```
val resize: 'a t -> int -> unit
(*@ resize a n
      checks   0 <= n <= Sys.max_array_length
```

```
      modifies a
      ensures  length a.view = n
      ensures  forall i. 0 <= i < min (length (old a.view)) n ->
                 a.view[i] = (old a.view)[i] *)
```

The `checks` keyword introduces a pre-condition that is meant to be verified at run-time. Instead of letting `resize` trying to allocate an array with a bad length, we dynamically test the value of `n` and stop the execution immediately if this is not within bounds. Concretely, we want an unsatisfied `checks` pre-condition to result in an `Invalid_argument` exception being raised, as is customary in OCaml libraries. When fed to our plugin, the above specification generates the following WhyML code:

WhyML

```
val unsafe_resize (a:t 'a) (n:int63) : unit
  requires { 0 <= n /\ n <= max_array_length }
  ensures  { length (view a) = n }
  ensures  { forall i. 0 <= i /\ i < min (length (view a1)) n ->
               (view a)[i] = (view a1)[i] }

val resize (a:t 'a) (n:int63) : unit
  ensures  { length (view a) = n }
  ensures  { forall i. 0 <= i /\ i < min (length (view a1)) n ->
               (view a)[i] = (view a1)[i] }
  raises   { Invalid_argument -> not (0 <= n /\ n <= max_array_length) }
```

The plugin introduces a new function named `unsafe_resize` where the `checks` condition is given in the form of a traditional pre-condition. We expect this function to be called from a verified client code, where we do not want to pay the price of unnecessary run-time tests but rather verify the given pre-condition. On the other hand, function `resize` turns the `checks` condition into a `raises` clause. This function dynamically tests the `checks` condition and raises `Invalid_argument` if such a condition is not met.

From an implementation perspective, we must provide both the `unsafe_resize` and the `resize` functions. The latter is straightforwardly implemented as follows:

WhyML

```
let resize (a: t 'a) (n: int63) : unit
  ensures { n = a.size }
  ensures { forall i. 0 <= i < min ((old a).size) n ->
              a.view[i] = (old a).view[i] }
  raises  { Invalid_argument -> not (0 <= n <= max_array_length) }
= if not (0 <= n <= max_array_length) then raise Invalid_argument;
  unsafe_resize a n
```

If we pass the test of the `checks` clause, then we are sure the pre-condition of `unsafe_resize` holds and so we can safely call it. As for the implementation of `resize`, this keeps the pre-condition as a **`requires`** clause:

WhyML

```
let unsafe_resize (a: t 'a) (n: int63) : unit
  requires { 0 <= n <= max_array_length }
  ensures  { n = a.size }
  ensures  { forall i. 0 <= i < min ((old a).size) n ->
               a.view[i] = (old a).view[i] }
= let n_old = length a.data in
  if n <= a.size then ... (* shrink *) ...
  else                ... (* grow   *) ...;
```

```
    a.size <- n
```

All the verification conditions generated for `resize` and `unsafe_resize` are automatically discharged. Providing a defensive and an unsafe version of the same function is already a usual practice in existing OCaml libraries, *e.g.*, the standard library functions `Array.get` and `Array.unsafe_get`.

When it comes to implement `push` and `pop`, these are verified functions and so we can call the unsafe version of `resize`. The implementation of `push` is as follows:

WhyML

```
let push (a: t 'a) (x: 'a) : unit
  requires { a.size < max_array_length }
  ensures  { a.size = (old a).size + 1 }
  ensures  { a.view[a.size - 1] = x }
  ensures  { forall i. 0 <= i < (old a).size -> a.view[i] = (old a).view[i] }
= let n = a.size in unsafe_resize a (n + 1);
  ...
```

The pre-condition of `push` and the type invariant of argument `a` imply the pre-condition of the call to `unsafe_resize`. All the verification conditions generated for `push` are proved by SMT solvers. We skip the specification and implementation of functions `get` and `set`, since these are straightforward.

Not all `requires` are born to be `checks`. Giving the presence of the `checks` in OSL, it seems tempting to get all **`requires`** conditions tested at run-time. Let us use an example to show why this is not always possible. In the following, we detail on the verified implementation of a binary search routine to find the insertion place of an element in an array. The following is the OCaml interface defining these two operations:

OCaml

```
val bisectr : ('a -> 'a -> int) -> 'a array -> int -> int -> 'a -> int
```

We give the following specification to function `bisectr`:

OCaml (annotated)

```
val bisectr : ('a -> 'a -> int) -> 'a array -> int -> int -> 'a -> int
(*@ r = binary_search_right cmp a fromi toi v
      checks   0 <= fromi <= toi <= Array.length a
      requires Order.is_pre_order cmp
      requires forall i j. fromi <= i <= j < toi -> cmp a.(i) a.(j) <= 0
      ensures              fromi <= r <= toi
      ensures  forall i.   fromi <= i < r   -> cmp a.(i) v <= 0
      ensures  forall i.   r     <= i < toi -> cmp a.(i) v >  0 *)
```

The first pre-condition states that the arguments `fromi` and `toi` must be valid indexes in the array. We give such a pre-condition in a `checks` clause, as this is definitely a property that we want and *can* test at zrun-time. The second pre-condition states that the argument `cmp` is a preorder (we assume here that predicate `is_pre_order` is defined in a module named `Order`, issued from a specification library). This cannot be checked at run-time and thus we keep it as a traditional **`requires`** clause. Finally, the third pre-condition indicates that the array `a` is sorted within the range `[fromi..toi[`. There is no practical constraint that would prevent us from writing a loop that scans the array and tests if the elements respect the order induced by function `cmp`. However, this would incur a linear cost, beyond the logarithmic cost of the binary search itself. Hence, we keep this as a **`requires`** pre-condition.

5.3.2 Higher-order Effectful Functions

In the example of the previous section, function `bisectr` is a higher-order function as this takes as an argument the comparison function `cmp`. In our specification (and in the subsequent proof), we implicitly assume function `cmp` to be pure. This assumption is somewhat justified, since we require `cmp` to implement a preorder, which would make it really difficult to specify and to use if it would depend on the state or, worse, had some side-effects.

In general, assuming that higher-order functions are only given pure arguments is not acceptable. A typical example of a stateful higher-order function is an iterator over the elements of a collection. In OCaml, it is idiomatic to provide, for some abstract type `t`, an `iter` function of the form

```
val iter : (elt -> unit) -> t -> unit                                    OCaml
```

A call to `iter f c` applies function `f` sequentially to each element of the collection `c`. This is only interesting if `f` performs some side-effects[2]. Providing a complete specification to this function would require a richer specification logic to account for function effects [28, 83], including abrupt termination if an exception is raised during iteration. This would lead to a rather unreadable contract for the OCaml programmer, sacrificing the principle of a simple specification language. Moreover, this would probably mean a richer logic than that of Why3, which would make it impossible to conduct our methodology within the Why3 framework.

We claim that the best specification for an `iter`-like function is an operationally equivalent program with a clear meaning to any OCaml programmer. If we consider the case of the `iter` function over arrays, we propose the following specification:

```
val iter : ('a -> unit) -> 'a array -> unit                    OCaml (annotated)
(*@ iter f a
      equivalent "for i = 0 to Array.length a - 1 do f a.(i) done" *)
```

The `equivalent` keyword of OSL is used to introduce a program with the same operational behavior as `iter`. In this case, it is almost the same as the standard library implementation of `iter`[3]. If we consider the more general case of some abstract collection, we can still give `iter` a specification, as follows:

```
val iter : ('a -> unit) -> 'a t -> unit                        OCaml (annotated)
(*@ iter f c
      equivalent "List.iter f (elements c)" *)
```

Function `elements` returns the list of the elements in the order they are traversed. In particular, we move the problem of specifying the iteration order to the specification of the `elements` function. In Chap. 6, we present several examples of `elements`-like functions, which we specify using our approach of characterizing the returned list with two predicates [67], one to identify valid prefixes and the other to identify complete lists. Combined with the semantics of `List.iter`, known by any OCaml programmer, the given `equivalent` clause fully specifies the behavior of `iter`. However, the program given in terms of `elements` is inefficient, since it builds an intermediate list unnecessarily. This makes it less suited to be used as an actual implementation of `iter`.

The use of keyword `equivalent` suggests a proof of program equivalence between the specification program and the actual implementation. Currently, Why3 does not provide such a possibility, and so we simply use the program given in the `equivalent` clause as the extraction result of function `iter`. This is not satisfactory, as we are not providing any formal guarantee about this piece

[2] In Chap. 6 we propose a way to tackle the specification and verification of effect-free iterators, *e.g.*, fold-like functions.

[3] The code of `Array.iter` uses `Array.unsafe_get` to access the elements of the array, since it is safe given the limits of the `for` loop.

of code. In Sec. 5.5, we discuss some related work on programs equivalence proofs and suggest some possible solutions to extend Why3 in order to conduct equivalence proofs.

5.3.3 Recursive Mutable Data Types

As seen in Sec. 5.2, the static discipline of Why3 limits the range of ephemeral data structures and heap-manipulating programs that can be encoded directly in WhyML. The solution is to resort to an explicit memory model, with a type of pointers and heap together with operations to interact with memory. In this section, we complement our presentation of pointer-based data structures from Sec. 5.2. We use our approach to systematically build memory models from arbitrarily, (possibly) recursive mutable OCaml data types. An example of such data type are precisely the `content` and `elem` types, on page 124. In this section, we present another example of a recursive mutable data type, whose definition is based on recently-introduced features of the OCaml language. Part of the ideas and material used in this section have been already presented in a JFLA (*Journées Francophones des Langages Applicatifs*) 2018 paper [68].

Since version 4.03 of the OCaml compiler[4], one can use records syntax to define the arguments of algebraic data type constructors. For instance, one can declare a type of mutable singly-linked lists as follows:

```
type 'a cell = Nil | Cons of { content: 'a; mutable next: 'a cell; }
```
OCaml

This type avoids the extra memory allocations that would happen had we used a record and an option type to define type `cell`, as follows:

```
type 'a cell = { content: 'a; mutable next: 'a cell option; }
```
OCaml

Other solutions amount at using recursive values or, even worse, the `Obj.magic` function [61]. With the `cell` type we can obtain a list representation similar to those of a Java or C code, where the `Nil` constructor works as the `null` pointer. However, being an algebraic data type, the OCaml type system ensures that we do not use `Nil` to access the fields of `content` or `next`.

We use the example of an in-place `mergesort` routine verification to illustrate the use of mutable singly-linked lists. Let us consider the following OCaml implementation:

```
let get_next = function Nil -> assert false | Cons { next } -> next

let split l = ...

let merge cmp l1 l2 = ...

let rec mergesort cmp l =
  if l1 == Nil || get_next l == Nil then l
  else let l2 = split l in
    let l1 = mergesort cmp l in
    let l2 = mergesort cmp l2 in
    merge cmp l1 l2
```
OCaml

For readability purposes, we omit the code of functions `split` and `merge`. Our goal here is to use Why3 to get a correct-by-construction version of function `mergesort`. Given the definition of type `cell`, we cannot directly encode such a type in WhyML and so the solution is, again, to introduce an explicit memory model. This is actually the approach adopted in tools using Why3 as an intermediate language, *e.g.*, Frama-C [87]. Contrary to such tools, though, we do not map every WhyML element to a common memory model. We build a specific memory model for type

[4] `https://caml.inria.fr/pub/docs/manual-ocaml/extn.html#sec272`

`cell` and keep using traditional WhyML types whenever possible. We begin by introducing the WhyML types corresponding to `cell` and memory locations, as follows:

```
type loc  'a                                                    WhyML
type cell 'a = Nil | Cons (loc 'a)
```

Next, we model the heap contents using the following record type:

```
type mem 'a = private {                                         WhyML
  mutable content : loc 'a -> option 'a;
  mutable next    : loc 'a -> option (cell 'a);
} invariant { forall l. content l = None <-> next l = None }
```

Fields `content` and `next` mimic to the two fields of constructor `Cons` argument. The type invariant ensures that both fields are simultaneously `None`. It is worth pointing out that the choice of introducing a heap type with several distinct fields is not innocent: this statically guarantees that a modification of one field does not affect the other, which makes the proof process much more easy. This idea dates back to 1972 and is due to Rod Burstall [25]. It is known under the designation of "component-as-array" memory model.

Similarly to what we did for the union-find example, we introduce several abstract functions to interact with the `mem` type. We do not show them here, as these very closely resemble the ones we defined for union-find.

Proof of function `mergesort`. We can now use the introduced memory model to implement, and then verify, a WhyML version of function `mergesort`. The complete code is give in Fig. 5.3.

Let us begin by explaining the definition of predicate `is_list` (lines 1–9). The idea is to model the elements from cell `from` until cell `to` (both inclusively) by means of a finite logical sequence of type `view 'a`. Such a type is defined as follows:

```
type view 'a = seq (loc 'a)                                     WhyML
```

For the case of an empty list (lines 1–3), we state both `from` and `to` point to same location in memory. On the other hand, if the list is non-empty, then `from` is a `Cons` node, its argument is the first element of sequence `s` (line 5), the list is finite and ends up with cell `to` (line 6), it does not contain any repeated element, hence no cycle (lines 7–8), and finally that the intermediate elements are those of sequence `s`. Next, predicate `frame_mem` (lines 11–13) states that a sequence `s` presents the same values for `next` and `contents` under memories `m1` and `m2`. From predicates `frame_mem` and `disjoint`, we can deduce the definition of predicate `frame` (lines 15 and 16). This states that, for every sequence `ss` disjoint from `s`, we have that the `next` and `contents` projections map the elements of `ss` to the same values in both memory `m1` and `m2`. This is useful to state that a part of the memory is not changed by the execution of some function, *i.e.*, we can *frame* the part of memory changed during execution.

We can now use predicates `is_list` and `frame` to give a contract to function `split` (lines 15–21): the first pre-condition requires `l1` and `l2` to point to well-formed nil-terminated lists; the second one states that we only split lists of at least two elements. The first postcondition ensures that the returned lists `l1` and `l2` are also well-formed nil-terminated lists, with respect to sequences `s1` and `s2`, respectively. Finally, we ensure that the only memory locations updated during the execution of `split` are those contained in sequence `s`.

Function `merge` presents a similar contract (lines 29–35). This function takes as arguments the memory, a `cmp` function defining a pre-order (line 29), and two lists sorted for the order relation induced by `cmp` (lines 30 and 31), disjoint from each other (line 32). The merge function returns a new sorted list `l` (line 33), whose elements are a permutation of the elements in lists `l1` and `l2` (line

```
predicate is_list (mem: mem 'a) (from : cell 'a) (s: view 'a) (to: cell 'a) =     WhyML
  let n = length s in
  n = 0 /\ from = to
\/
  n > 0 /\ from = Cons s[0] /\
  mem.next s[n-1] = Some to /\
  (forall i. 0 <= i < n -> Cons s[i] <> to) /\
  distinct s /\
  forall i. 0 <= i < n - 1 -> mem.next s[i] = Some (Cons s[i+1])

predicate frame_mem (m1 m2: mem 'a) (s: view 'a) =
  forall i. 0 <= i < length s ->
    m1.next s[i] = m2.next s[i] /\ m1.contents s[i] = m2.contents s[i]

predicate frame (m1 m2: mem 'a) (s: view 'a) =
  forall ss: view 'a. disjoint s ss -> frame_mem m1 m2 ss

let split (ghost mem: mem 'a) (l1: cell 'a) (ghost s: view 'a) :
          (s1: ghost view 'a, l2: cell 'a, s2: ghost view 'a)
  requires { is_list mem l1 s }
  requires { length s >= 2 }
  ensures  { is_list mem l1 s1 /\ is_list mem l2 s2 }
  ensures  { s = s1 ++ s2 }
  ensures  { frame (old mem) mem s }
= ...

let merge (ghost mem: mem 'a) (cmp: 'a -> 'a -> int63)
          (l1 l2: cell 'a) (ghost s1 s2: view 'a) : (l: cell 'a, s: ghost view 'a)
  requires { is_pre_order cmp }
  requires { is_list mem l1 s1 /\ sorted mem cmp s1 }
  requires { is_list mem l2 s2 /\ sorted mem cmp s2 }
  requires { disjoint s1 s2 }
  ensures  { is_list mem l s }
  ensures  { sorted mem cmp s /\ permut_all (s1 ++ s2) s  }
  ensures  { frame (old mem) mem s }
= ...

let rec mergesort (ghost mem: mem 'a) (cmp: 'a -> 'a -> int63)
                  (l: cell 'a) (ghost s: view 'a) : (r: cell 'a, s': ghost view 'a)
  requires { is_pre_order cmp }
  requires { is_list mem l s }
  variant  { length s }
  ensures  { is_list mem r s'}
  ensures  { permut_all s s'/\ sorted mem cmp s' }
  ensures  { frame (old mem) mem s }
= if l == Nil || get_next mem l == Nil then l, s
  else let s1, l2, s2 = split mem l s in
    let l1, s1 = mergesort mem cmp l s1 in
    let l2, s2 = mergesort mem cmp l2 s2 in
    merge mem cmp l1 l2 s1 s2
```

Figure 5.3: WhyML Implementation of Function `mergesort`.

34). We also ensure that the execution of `merge` only changes the part of the heap corresponding to locations of the `s1` and `s2` sequences (line 35).

Finally, we can give a suitable contract to function `mergesort` (lines 40–45) and prove that the implementation conforms to such a specification. From the specification of functions `split` and `merge`, we can easily prove the correctness of the body of `mergesort`. In fact, all the generated verification conditions by Why3 for this function are automatically discharged. The whole WhyML development of this in-place mergesort routine is composed of more than 200 lines of specification (auxiliary lemmas, function contracts, etc.), and by almost 200 lines of code (this includes ghost code).

5.3.4 Functors

The WhyML language features a module system rather different from that of OCaml. We can divide a WhyML development into several top-level modules, introduced with the **`module`** keyword; a module can be further divided into several sub-namespaces, introduced via the keyword **`scope`**. This is an important difference w.r.t. OCaml, where modules can feature sub-modules. Another distinguishing feature of WhyML is that it allows uninterpreted and defined symbols to appear in the same namespace. In this section, we explain how we use the WhyML module system to mimic some of the most interesting features of the OCaml language, namely *functors*. We use the Pairing Heaps [71, 118] data structure as our working example of specification, translation and extraction of an OCaml functor within our toolchain of verified OCaml programs.

Functors are used in OCaml to implement modules parameterized by other modules. A typical example is data structure that requires its elements to be equipped with an order relation, *e.g.*, a priority queue implementation. Within our toolchain, we are able to give a specification to an OCaml interface that declares such a data structure. Fig. 5.4 presents the complete specification of a priority queue interface file, which we name `PairingHeap.mli`. When we feed our plugin with the `PairingHeap.mli` file, it generates the following WhyML module:

WhyML

```
module Sig
  scope Make
    scope X
      type t

      function cmp t t : int63

      axiom is_pre_order : is_pre_order (fun (y0:t) (y1:t) -> cmp y0 y1)

      val compare (x:t) (y:t) : int63
        returns  { r -> r = cmp x y }
    end

    type elt = t

    type heap = private { ghost bag : bag elt }

    val empty (us:()) : heap
      returns  { h -> card (bag h) = 0 }
      returns  { h -> forall x:t. nb_occ x (bag h) = 0 }
    ...
  end
```

OCaml (annotated)

```
module Make
  type t

  (*@ function cmp: t -> t -> int *)
  (*@ axiom is_pre_order: Order.is_pre_order cmp *)

  val compare : t -> t -> int
    (*@ r = compare x y
          ensures r = cmp x y *)
end) : sig
  type elt = X.t

  type heap
  (*@ model bag : elt bag *)

  val empty : unit -> heap
  (*@ h = empty ()
        ensures card h.bag = 0
        ensures forall x. nb_occ x h.bag = 0 *)

  val is_empty : heap -> bool
  (*@ b = is_empty h
        ensures b <-> h.bag = empty_bag *)

  val merge : heap -> heap -> heap
  (*@ h = merge h1 h2
        ensures card h.bag = card h1.bag + card h2.bag
        ensures forall x. nb_occ x h.bag = nb_occ x h1.bag + nb_occ x h2.bag *)

  val insert : elt -> heap -> heap
  (*@ h' = insert x h
        ensures nb_occ x h'.bag = nb_occ x h.bag + 1
        ensures forall y. y <> x -> nb_occ y h'.bag = nb_occ y h.bag
        ensures card h'.bag = card h.bag + 1 *)

  (*@ predicate mem        (x: elt) (h: heap) := nb_occ x h.bag > 0 *)
  (*@ predicate is_minimum (x: elt) (h: heap) :=
        mem x h /\ forall e. mem e h -> X.cmp x e <= 0 *)

  (*@ function minimum: heap -> elt *)
  (*@ axiom min_def: forall h. 0 < card h.bag -> is_minimum (minimum h) h *)

  val find_min : heap -> elt
  (*@ x = find_min h
        requires card h.bag > 0
        ensures  x = minimum h *)

  val delete_min : heap -> heap
  (*@ h' = delete_min h
        requires card h.bag > 0
        ensures  let x = minimum h in nb_occ x h'.bag = nb_occ x h.bag - 1
        ensures  forall y. y <> minimum h -> nb_occ y h'.bag = nb_occ y h.bag
        ensures  card h'.bag = card h.bag - 1 *)
end
```

Figure 5.4: Pairing Heaps Specification.

```
end
```

Let us explain the hierarchy of modules and namespaces from the point of view of our plugin. The top-level module of a `.mli` file is translated by our plugin into the WhyML module `Sig`. Any sub-module declared in the same `.mli` file is recursively extracted into a WhyML namespace, which is the closest we get to OCaml sub-modules. File `PairingHeap.mli` features two sub-modules: on one hand, `Make` is a sub-module of the top-level module; on the other hand, the functor signature `X` is a sub-module of `Make`, so our plugin includes the definition of scope `X` inside the body of scope `Make`.

The next step is to provide a WhyML implementation with identical namespaces. The following is a sketch of our WhyML implementation of Pairing Heaps:

WhyML
```
module Impl
  scope Make
    scope X
      type t
      val function compare elt elt : int63
      axiom is_pre_order: is_pre_order compare
    end

    type tree = E | T X.t (list tree)
    type heap = { data : tree; ghost bag: bag X.t }

    let merge (h1 h2: heap) : heap = ...
    let delete_min (h: heap) : heap
  end
end
```

This a WhyML implementation of significant size so, for the sack of readability, we do not show it entirely here and focus only on the most relevant aspects. Let us begin by noting that every symbol declared in scope `X` is left undefined. As we shall later see in this section, this generates a functor argument at extraction time. The implementation task happens inside scope `Make`. Here, we can use symbols from `X` to provide implementations to our types and functions, just like one would do in an OCaml functor. For instance, type `tree` is defined in terms of type `X.t`.

A Pairing Heap is implemented as a multiway tree, as defined by type `tree`. The `heap` type encapsulates a field of type `tree`, together with the logical model of type `bag X.t`. This way, we can give a specification of Pairing Heaps in terms of the *bags* structure, also known as multisets. We equip type `heap` with the following invariant:

WhyML
```
  invariant { forall x. nb_occ x bag = occ_heap x data }
  invariant { card bag = size_heap data }
  invariant { heap data }
  invariant { no_middle_empty_heap h }
```

The first two lines establish the connection between physical representation of the heap, field `data`, and the logical model `bag`: the number of occurrences of any element `x` is the same in both `data` and `bag`; the number of elements is the same in both fields. Functions `occ_heap` are straightforwardly defined, and so we do not give their definition here. The third line of the invariant states that `data` is a heap-ordered structured. The `heap` predicate is defined as follows:

WhyML
```
predicate heap (t: tree) = match t with
  | E     -> true
  | T x l -> le_roots_list x l /\ heap_list l
```

```
    end
  with heap_list (l: list tree) = match l with
    | Nil      -> true
    | Cons h r -> heap h /\ heap_list r
    end
```

where `le_roots_list x l` states that `x` is no greater than the roots of the trees in `l`. Finally, we impose that there is no empty tree `E` in the middle of the child list of a `T` node. Predicate `no_middle_empty_tree` is straightforwardly defined.

Pairing Heaps are very simple to implement, while performing very well in practice [94]. The `merge` function simply takes the `heap` with the larger root, with respect to the `X.compare` order, and makes it the first child of the heap with the smaller root. We give the following WhyML implementation:

WhyML

```
let merge_tree (t1 t2: tree) : tree
  requires { heap t1 /\ heap t2 }
  ensures  { heap result }
  ensures  { size_heap result = size_heap t1 + size_heap t2 }
  ensures  { forall x. occ_heap x result = occ_heap x t1 + occ_heap x t2 }
= match t1, t2 with
  | E, h | h, E -> h
  | T x1 l1, T x2 l2 ->
     if X.compare x1 x2 <= zero then T x1 (Cons t2 l1)
     else T x2 (Cons t1 l2)
  end

let merge (h1 h2: heap): heap
  ensures  { card result.bag = card h1.bag + card h2.bag }
  ensures  { forall x. nb_occ x result.bag = nb_occ x h1.bag + nb_occ x h2.bag }
= { data = merge_tree h1.data h2.data; bag = union h1.bag h2.bag }
```

Proving that `merge` and `merge_term` conform to their specification is rather easy, as SMT solvers are able to discharge all the generated verification conditions in no time. The delete minimum operation works in two steps: first, we remove the root of the heap and merge the children in pairs, *i.e.*, the first child with second, the third with the fourth one, and so on; the second step merges all the impaired heaps to get the final heap. We implement these two steps as the following WhyML function:

WhyML

```
let rec merge_pairs (l: list tree) : tree
  requires { heap_list l /\ no_middle_empty_heap_list l }
  variant  { length l }
  ensures  { heap result }
  ensures  { size_heap result = size_heap_list l }
  ensures  { forall x. occ_heap x result = occ_heap_list x l }
= match l with
  | Nil                 -> E
  | Cons h Nil          -> h
  | Cons h1 (Cons h2 r) -> merge_tree (merge_tree h1 h2) (merge_pairs r)
  end
```

Pairing Heaps are named so because of the `merge_pairs` operation. Function `delete_min` is easily deduced from `merge_pairs`, as follows:

```
let delete_min (t: heap) : heap                                    WhyML
  requires { 0 < size t }
  ensures  { occ (minimum t) result = occ (minimum t) t - 1 }
  ensures  { forall y. y <> minimum t -> occ y result = occ y t }
  ensures  { size result = size t - 1 }
= match t.data with
  | E     -> absurd
  | T x l -> { data = merge_pairs_heap l; bag = diff t.bag (singleton x) }
  end
```

The **absurd** point is provable thanks to the given pre-condition. All the verification conditions generated for `delete_min` are discharged automatically. The remaining heap operations are even easier to implement, and their correctness proof is easily done by SMT solvers.

The last step in the proof of Pairing Heaps amounts to specification inclusion. This is no different from the previously presented proofs of specification inclusion, except that we need to the follow the hierarchy of namespaces, in order to provide the good substitution to the **clone** command. For the case of Pairing Heaps, this is as follows:

```
clone PairingHeap.Sig with                                         WhyML
  goal     Make.X.is_pre_order,
  type     Make.X.t       = Impl.Make.X.t,
  val      Make.X.compare = Impl.Make.X.compare,
  function Make.X.compare = Impl.Make.X.compare,
  goal     Make.min_def,
  type     Make.heap      = Impl.Make.heap,
  function Make.minimum   = Impl.Make.minimum,
  ...
```

We omit here the substitution for the Pairing Heaps **val** functions. Note that Why3 generates verification conditions to prove the definition of axioms `is_pre_order` and `min_def`, against the provide implementation of functions `compare` and `minimum`. These, and the other verification conditions generated for this **clone** expression, are discharged by SMT solvers.

Finally, we can extracted an OCaml version of the Pairing Heaps data structure. By default, Why3 refuses to extract a WhyML code where there are some uninterpreted symbols left which are not defined in the driver, *e.g.*, type `elt` or function `compare` from the `X` scope. The only exception is a namespace that contains *only* uninterpreted symbols. In such a case, we turn it into a functor argument when the `--modular` option is given in the Why3 extraction command-line, as follows:

```
> why3 extract -L . -D ocaml64 --modular PairingHeap.Impl -o .     Terminal
```

With such an option, Why3 generates a different OCaml file for each WhyML module, while extracting every scope as a sub-module. The absence of `modular` would induce a *flat* extraction, *i.e.*, every WhyML symbol extracted into a single file, contained no sub-module, even if the development is spread across different WhyML modules. With our Pairing Heaps example, we get some OCaml code with the expected structure, *i.e.*,

```
module Make(X: sig type t val compare : t -> t -> t end) = struct  OCaml (extracted)
  type tree = E | T of X.t * tree list

  type heap = tree
```

module	spec	code	#VCs	
`UnionFind`	74	176	135	union-find
`PairingHeap`	41	245	52	persistent priority queues
`ZipperList`	66	180	87	zipper data structure for lists
`Arrays`	37	121	77	binary search and binary sort
`Queue`	54	185	119	mutable queues
`Vector`	149	309	142	resizable arrays
`HashSet`	21	34	12	sets using hash tables
`MergeSort`	12	401	630	in-place mergesort of lists
`BinaryHeap`	198	144	222	binary heaps implemented using Vector

Table 5.1: Verified OCaml Modules.

```
  let merge_heap (t1: tree) (t2: tree) : tree =
    begin match (t1, t2) with
    | (E, h) | (h, E) -> h
    | (T (x1, l1), T (x2, l2)) ->
        if (X.compare x1 x2) <= 0 then begin T (x1, (t2 :: l1)) end
        else begin T (x2, (t1 :: l2)) end
    end

  let merge (h1: heap) (h2: heap) : heap = merge_heap h1 h2

  ...
end
```

Note that, in the extracted code, type `heap` is simply an alias for type `tree`. Indeed, the Why3 extraction mechanism is able to recognize and optimize record types containing a single, *non-mutable* field.

5.4 Experimental Evaluation

We have used our approach to verify several other OCaml modules. These examples illustrate many features we described in this chapter, *i.e.*, preconditions verified at run-time, OCaml functors, higher-order effectful functions, absence of arithmetic overflows, and pointer-based implementations. Figure 5.1 summarizes the size of these examples, column "spec" showing the number of lines in the `.mli` files and the "code" column showing the number of lines in the WhyML implementation and proof. It is worth stressing out that all the specification and proof tasks are conducted inside our toolchain, and all the proofs are done fully automatically.

5.5 Discussion and Related Work

Verified programming libraries. Libraries are the basic building blocks of any realistic programming project. It is thus of crucial importance that libraries are bug-free software artifacts, preventing the programmer from the pain of seeing her implementation crashing due to a piece of code that she did not even write in the first place. As we have already discussed, even massively used and tested libraries can contain bugs, which makes programming libraries interesting

candidates for deductive verification. Although this may seem surprising, program verification has seldom been applied to libraries of significant size. The most remarkable exception is the verification of the EiffelBase2 containers library [128, 129], performed with the AutoProof system [142]. The verification of the EiffelBase2 library builds on the work of Nadia Polikarpova [127], namely on the use of semantic collaboration [130] and model-based contracts to extend the classic Design by Contract approach of the AutoProof tool. It is our purpose to continue using and improving our methodology to grow our verified library to a size comparable to that of EiffelBase2.

The recent work of Raphael Cauderlier and Mihaela Sighireanu [27] reports on the verified implementation of a bounded list container. The authors verify a C implementation of the mentioned container, using the VeriFast platform and its underlying separation logic [135] to prove functional correctness and safety of memory accesses. An interesting aspect pointed out in this work is the importance of reusable specification of data structures, as the specification of the bounded list container is drawn from a previously existing Ada 2012 specification [56]. This opens good perspectives about the proof of bounded lists in other verification systems. For instance, such a data structure is certainly a good candidate to our toolchain and methodology.

Following the trend of verified container libraries, the work by Joachim Breitner *et al.* [23] explores the use of an Haskell to Coq translation tool to verify significant portions of the Haskell's containers library. The authors attest that no bugs were found during this experience, but their verification effort lead to some direct optimizations in the code of the library. One key difference between our methodology and this works is the effective treatment of arithmetic, in particular the proof of arithmetic overflows absence. When it comes to prove that the size of a set container does not exceed $2^{63} - 1$ elements, the authors invoke a space argument (the amount a memory required to store such a data structure) to translate the Haskell's `Int` type to Coq's `Z` type of unbounded integers. Such an argument does not always hold, especially in a setting of a pure language like Haskell, where memory sharing is of great importance. The storage needed for the set container can thus be dramatically reduced. Within our methodology, our proofs are always done under some form of bounded arithmetic.

Using the KeY verification tool, Stijn de Gouw, Frank de Boer, and Jurriaan Rot built a proof of two classical sorting algorithms, Counting sort and Radix sort [50]. Pursuing their efforts to verify the implementation of massively used algorithms, this team of researchers embraced the challenge of verifying TimSort, the default sorting algorithm for generic collections of the JDK platform. Interestingly, the team was not able to provide a correctness proof for the implementation of TimSort simply because this was broken. This is reported in the work by Stijn de Gouw *et al.* [51, 52]. More recently, Bernhard Beckert, Jonas Schiffl, Peter Schmitt, and Mattias Ulbrich used KeY to formally verify JDK's dual pivot quicksort implementation [17], the default sorting algorithm for integer or long arrays.

Refinement. One key ingredient in our approach is the ability to refine a Why3 module containing specifications by a Why3 module containing the implementation, with suitable verification conditions generated to ensure correctness of refinement. This is reminiscent of other systems using a refinement approach. One obvious example is the B-method [1]. A fundamental difference, though, is that we only proceed in one step, where a B machine is typically refined in several steps. Nonetheless, Why3 modules are richer than B machines. These contain only a global notion of state, whereas in Why3 we can refine record types containing mutable fields, using them locally, and then refine the definition of such data types.

Closer to our work is the integration of module refinement in the Dafny program verifier by Leino and Koenig [89]. Dafny module system does not make distinction between interface and implementation: the same notion of module is used both to give abstraction and to refine it. When refining a module in Dafny, one may give definitions to the data structures and methods

left uninterpreted in the interface module, bring additional declarations, and refine previously given specifications. The main difference between module refinement in Dafny and Why3 concerns mutable data structures. In Dafny, mutable state is encapsulated within a class and dynamic frames are typically used to control side effects. In Why3, mutable data is encapsulated within record types, and it is the type system that controls side effects.

Mixins [5] is another example of a flexible module system that can mix uninterpreted symbols with defined ones. However, contrary to Why3 and Dafny module systems, mixins are designed for programming purposes only.

Proof of programs equivalence. In Sec. 5.3.2, we described our approach to specify the behavior of stateful higher-order functions. We use the `equivalent` clause of OSL to give a program which we state to be operationally equivalent to a specific higher-order function. Such a program is then appended to the result of the Why3 extraction mechanism. This is not satisfactory since we are not even proving the safety of that piece of code. In order to circumvent this pitfall in our methodology, it would be interesting to explore means to extend Why3 with the ability to prove equivalence of two programs, in the style of Relational Hoare Logic [18]. A possible direction would be to adapt the work of Gilles Barthe, Juan Manuel Crespo, and César Kunz [13] on the use of *product programs* to verify relational properties between two programs.

Memory model manipulation. Our experiments with mutable data structures, presented in Sec. 5.2 and 5.3.3, show that is feasible to conduct proofs on pointer-based mutable data structures using the WhyML language. However, we feel that a better tool support from Why3 would significantly reduce the effort and time spent around the construction of memory models, as well as reasoning about common memory manipulation operations. To this end, we have already started a prototype tool that takes as input a (possibly) recursive mutable OCaml data type, and automatically generates a suitable memory model, together with memory interaction functions and a driver for extraction. In the future, such a prototype tool could become a Why3 plugin, working in very similar way as Jessie [112] works for Frama-C. On the other hand, to relieve some of the effort of reasoning about an explicit memory model, we plan on equipping Why3 with some form of support to reason about memory contents and evolution. This could be, for instance, a separation logic library, as in KIV [134].

Finally, our use of small heaps statically guarantees the separation of two memory fragments passed as arguments to our memory model functions. This relies on the principle that memory is represented as a private data type and is only updated through abstract functions. We argue that is ensures a correct manipulation of the heap, *e.g.*, we cannot allocate the same pointer in two distinct portions of the heap. This is, however, a rather non-trivial argument which deserves a proof of soundness. In order to formally express such a statement about memory manipulation, we would certainly have to depart from the logic of Why3 and use a proof system based on a richer logic, such as those based on separation logic [29, 90].

L'éducation c'est la répétition.

French saying

6 A Modular Way to Reason About Iteration

Iteration is a central concept in programming. It can be as simple as a while loop or a recursive function, but it can also appear as a more complex artifact, such as a cursor, a higher-order iterator, a generator, or a lazy list. When it comes to verifying the correctness of a program, we need tools to reason about iteration. Typically, we provide a suitable loop invariant for a while loop and a contract for a recursive function. In this chapter, we consider the problem of verifying programs where iteration is performed by other means, such as cursors or higher-order iterators.

Our first goal is to devise an approach to *specify* an iteration process, independently of how this is implemented. We seek for a *modular* specification, which clearly establishes an *abstraction barrier* between implementation and client code. A second goal is to use such a specification in the context of deductive program verification. We validate our work using Why3, but the idea is broader and could be implemented in any other deductive verification tool. We experimentally validate our approach through the verification of several implementations of cursor and higher-order iterators, as well as client code. The results of this chapter were partially presented at JFLA 2016 [66] and at NFM 2016 [67].

6.1 Specifying Iteration

We present in this section our proposal to formally specify iteration. We use several examples to illustrate this proposition, including cases of non-deterministic and infinite iteration. In particular, we are interested in answering the following challenges:

- Iteration is not necessarily the traversal of a data structure. It can be, for instance, the result of an algorithm, such as the enumeration of all prime numbers.
- Iteration is not necessarily finite, as in the aforementioned case of prime numbers.
- Iteration is not necessarily deterministic. The simplest example is that of a symbol generator. From the client point of view, the only required property is that the next element is distinct from the previous ones. Another example is the traversal of a set where elements are presented in some unspecified order. When the iteration is deterministic, however, we want to be able to specify it.

- When iteration depends on mutable data, client code may put iteration in some inconsistent state. In Java, for instance, this problem is solved by maintaining version numbers and by raising an exception in the case of a concurrent modification. In our case, we wish instead to be able to prove, statically, that there is no concurrent modification.
- When a data structure is abstract (for example, a set for which we do not know the implementation) we still want to be able to specify an iteration over its elements and to verify a program using such an iteration. Even when we have access to the implementation of the iteration, we are still interested in performing verification in a modular way with an abstraction barrier. It means verifying the client code independently of a particular implementation for the iteration.

We propose to specify iteration in terms of the finite sequence ν of the elements enumerated so far, and only those. This can be depicted as follows:

$$\underbrace{\nu_0 \quad \nu_1 \quad \cdots \quad \nu_{n-1}}_{\textit{already visited}} \quad \underbrace{? \quad ? \quad \cdots \quad ?}_{\textit{to come}}$$

By imposing no particular property about the forthcoming elements, we are able to cope with non-deterministic and (potentially) non-terminating enumerations. To reason about the evolution of the iteration process, we introduce two predicates: the first predicate, called *permitted*, characterizes the elements of ν and we can consider such a predicate as the invariant of iteration; the second predicate, called *complete*, indicates whether the iteration is completed. In the following, $\|\nu\|$ denotes the length of ν, $\nu[i]$ denotes the i-th element of ν (assuming a 0-based indexation), and $x \in \nu$ means that x occurs in ν.

Let us illustrated the use of *permitted* and *complete* through several examples. Consider for instance the iteration over an array a, from left to right. The first predicate, *permitted*, is as follows:

$$\mathit{permitted}(\nu, a) \triangleq \|\nu\| \leq \mathit{length}(a) \wedge \forall i.\ 0 \leq i < \|\nu\| \implies \nu[i] = a[i]$$

In other words, the sequence ν is a prefix of the array a. The second predicate, *complete*, simply compares the length of ν with that of a:

$$\mathit{complete}(\nu, a) \triangleq \|\nu\| = \mathit{length}(a)$$

Specifying a right to left iteration is just a matter of changing the definition of *permitted* to state the sequence v forms a suffix of the array.

Let us now consider the iteration over the elements of a finite set s, in a non-deterministic way. Such an iteration can be specified as follows:

$$\begin{aligned} \mathit{permitted}(\nu, s) &\triangleq \mathit{distinct}(\nu) \wedge \forall x.\ x \in \nu \implies x \in s \\ \mathit{complete}(\nu, s) &\triangleq \|\nu\| = \mathit{card}(s) \end{aligned}$$

The condition $\mathit{distinct}(\nu)$ means that the elements of sequence ν are pair-wise distinct. This is needed to reflect the fact s is a set (and not a bag), so no element can be visited twice. We require, as well, the elements of ν to be elements of s. Since we do not require any additional property, we have a non-deterministic iteration. The iteration is completed whenever the length of ν is equal to the cardinal of s. Let us now specify, instead, a *deterministic* iteration over the elements of s. One way to do so is to introduce some oracle function *elements* that returns a sequence containing the elements of s in the order they will be visited. Then, *permitted* merely says that we have already visited a prefix of this sequence, that is,

$$\mathit{permitted}(\nu, s) \triangleq \mathit{prefix}(\nu, \mathit{elements}(s))$$

with a natural definition for *prefix*:

$$\mathit{prefix}(s_1, s_2) \triangleq \|s_1\| \leq \|s_2\| \wedge \forall i.\ 0 \leq i < \|s_1\| \implies s_1[i] = s_2[i]$$

With such a specification, the behavior of the enumeration is determined from the beginning. For instance, if the elements of s are totally ordered, then $\mathit{elements}(s)$ could be the sorted sequence of the elements of s.

Let us switch to examples of iteration that are not traversals of a data structure. Consider for instance an iteration obtained by the repeated application of a function f starting with some initial value x_0, that is, the infinite sequence

$$x_0,\ f(x_0),\ f(f(x_0)),\ f(f(f(x_0))),\ \ldots$$

One way to specify it is as follows:

$$\mathit{permitted}(v, x_0, f) \triangleq \forall i.\ 0 \leq i < \|v\| \implies v[i] = f^i(x_0)$$

assuming f^i is defined as the i-th functional power of f. To account for the fact that this iteration never halts, we simply define *complete* as follows:

$$\mathit{complete}(v, x_0, f) \triangleq \mathit{false}$$

Note that, in this case, both *permitted* and *complete* are parameterized by the value x_0.

The next example is the specification of a scanner for a possibly infinite channel c. The elements of v are characters and a special character `EOF` marks the end of the channel. The specification looks like:

$$\begin{aligned} \mathit{permitted}(v, c) &\triangleq \cdots \wedge \forall i.\ 0 \leq i < \|v\| - 1 \implies v[i] \neq \texttt{EOF} \\ \mathit{complete}(v, c) &\triangleq \|v\| > 0 \wedge v[\|v\| - 1] = \texttt{EOF} \end{aligned}$$

The first part of the definition of *permitted*, marked with $\ldots$, specifies that the channel, for instance, is only composed of alpha-numeric characters. The given specification covers both the case of a finite channel, with a terminal `EOF`, and the case of an infinite channel, where `EOF` never shows up.

Our last example is that of a *symbol generator*, *i.e.*, a program that generates fresh symbols on demand. Its output is an infinite iteration of distinct symbols, which we specify as follows:

$$\begin{aligned} \mathit{permitted}(v) &\triangleq \mathit{distinct}(v) \\ \mathit{complete}(v) &\triangleq \mathit{false} \end{aligned}$$

In this case, the specification does not depend on any information other than the sequence v itself.

6.2 Cursors

A cursor [41] is a data structure used to traverse collections, where the *consumer* is in control of the iteration process. The word "collection" is to be taken here broadly, as it does not necessarily means a physical data structure. Each time a new element is needed, one needs to explicitly interact with the cursor. Cursors are broadly used in C++ and Java, for instance.

We adopt a model where we interact with the cursor via two functions: `has_next` returns a Boolean indicating the existence of a next element in the iteration; and `next`, which advances to the next element and returns it. The latter operation updates the cursor via a side effect. A typical client code looks like

```
c <- create_cursor(...)                                          Syntax
while has_next(c) do
  x <- next(c)
  ...
```

In Java, the "for each" loop construct `for (E x: ...)` is nothing more than syntactic sugar for the above.

In this section we describe the use of predicates *permitted* and *complete* to formally specify what is a cursor (Sec. 6.2.1), to verify a cursor implementation (Sec. 6.2.2), and to verify client code (Sec. 6.2.3). Finally, we apply our approach to specify and formally verify several case studies (Sec. 6.2.5 and 6.2.6).

6.2.1 Cursor Specification

In this section, we detail on a generic specification for the cursor data structure. We introduce two types: type `elt` of the elements enumerated by a cursor, and type `t` for the collection whose elements we enumerate, as follows:

```
type t                                                           WhyML
type elt
```

Next, we declare predicates `permitted` and `complete`, as follows:

```
predicate permitted t (seq elt)                                  WhyML
predicate complete  t (seq elt)
```

These are undefined predicates, which we must instantiate for each new kind of cursor. A cursor is modeled by the following record type:

```
type cursor = private {                                          WhyML
  ghost mutable    visited: seq elt;
  ghost         collection: t;
} invariant { permitted collection visited }
```

This is a private type, with two fields: ghost field `visited` stands for the sequence of already enumerated elements by the cursor; field `collection` represents the collection for which the cursor is defined. Being exclusively composed of ghost fields, the `cursor` type has no physical representation. For every new instance of a cursor we are supposed to refine type `cursor` with new, possibly regular, fields. One can think of a private type as a type for which we are not yet aware of the full definition. Let us note that we equip type `cursor` with the `permitted` property as the type invariant. This follows the idea that `permitted` acts as an invariant of the iteration process. Why3 generates a verification condition of the form $\exists\, \nu,\ \mathtt{t}.\ \mathtt{permitted\ t}\ \nu$ to ensure there is an inhabitant of type `cursor` that satisfies the type invariant. To prove such a formula, we add to our formalization the following axiom:

```
axiom permitted_empty: forall t. permitted t empty               WhyML
```

It simply states that `permitted` always holds for an `empty` sequence of visited elements. When we shall provide concrete definitions for types `t` and `elt`, as well as for predicates `permitted` and `complete`, we can then turn such an axiom into a provable lemma.

Using predicates `complete` and `permitted`, we can provide suitable contracts for functions `has_next` and `next`. The first one, is declared as the following unimplemented function:

```
val has_next (c: cursor) : bool                                  WhyML
  ensures  { result <-> not (complete c.collection c.visited) }
```

Function `has_next` decides whether predicate `complete` holds, as expressed in its postcondition. The second operation, `next`, is introduced as follows:

```
val next (c: cursor) : elt                                        WhyML
  requires { not (complete c.collection c.visited) }
  writes   { c.visited }
  ensures  { c.visited = snoc (old c).visited result }
```

Following the pre-condition of this function, we can only call `next` whenever the iteration is not complete. The postcondition ensures the returned element is append to the end of the `visited` sequence. This is done via a side effect, which we state in the **`writes`** clause of the function.

To be able to use cursors, we also need an operation to create a new value of type `cursor`. We name such an operation `create`, and declare it as follows:

```
val create (t: t) : cursor                                        WhyML
  ensures  { result.visited = empty }
  ensures  { result.collection = t }

end
```

Given a collection `t`, function `create` returns a new cursor with an empty `visited` sequence, as well as with field `collection` assigned to `t`. This completes the definition of module `Cursor`.

Let us note that, at any moment, we refer to predicate `permitted` in the contract of the cursor operations. Thanks to the `cursor` type invariant, any operation manipulating a value of type `cursor` must preserve the `permitted` property. In other words, the cursor operations assume the invariant as a pre-condition and that it is re-established at the postcondition. For instance, the writing operation of function `next` must not invalidate the `cursor` invariant. Whenever we call `next`, Why3 generates verification conditions including the `permitted` condition in the premises of the proof context.

Example: cursor specification for arrays. We detail on a specification for a cursor over arrays. We encapsulate the WhyML code within a module named `CursorArraySpec`, and we begin by declaring types `elt` and `t` as follows[1]:

```
module CursorArraySpec                                            WhyML

  type elt
  type t = seq elt
```

We define the `cursor` type as follows:

```
type cursor = private {                                           WhyML
  ghost mutable    visited: seq elt;
                collection: t;
} invariant { permitted collection visited }
```

We keep `permitted` as the type invariant of `cursor`. We can provide a concrete definition for predicates `permitted` and `complete`, as follows[2]:

```
predicate permitted (t: t) (v: seq elt) =                         WhyML
  length v <= length t /\
```

[1] Whenever we introduce a new WhyML module, for the sake of readability, we do not present the external dependencies on other WhyML modules.

[2] For the sake of clarity, we authorize ourselves to surcharge operations `length` and `[]` to apply these both to sequences and arrays, even if Why3 does not (yet) support operations overloading.

```
  forall i. 0 <= i < length v -> t[i] = v[i]

predicate complete (t: t) (v: seq elt) =
  length v = length t
```

The `permitted` sequences are those that form a prefix of the collection, and the iteration is `complete` when the length of the `visited` sequence is the same as the length of the collection.

We conclude module `CursorArraySpec` with the declaration of the cursor operations:

WhyML
```
val next (c: cursor) : elt
  requires { not (complete c.collection c.visited) }
  writes   { c.visited }
  ensures  { c.visited = snoc (old c).visited result }

val has_next (c: cursor) : bool
  ensures  { result <-> not (complete c.collection c.visited) }

val create (a: array elt) : cursor
  ensures  { result.visited = empty }
  ensures  { result.collection = a }
```

6.2.2 Cursor Implementation

A specification is nothing without a corresponding implementation. In this section, we present the actual implementation of a cursor to traverse an array. We shall equip type cursor with some regular fields, and give concrete implementations to the cursor operations. At end, we shall prove that the chosen implementation actually refines the cursor specification given in the previous section.

We encapsulate the WhyML implementation in a module named `CursorArrayImpl`, which begins with the definition of the `cursor` type, as follows:

WhyML
```
module CursorArrayImpl

  type elt
  type t = array elt

  predicate permitted (collection: t) (visited: seq elt) (index: int) =
    index = length visited /\
    length visited <= length collection /\
    forall i. 0 <= i < index -> visited[i] = collection[i]

  type cursor = {
    ghost mutable    visited: seq elt;
          mutable      index: int;
                  collection: t;
  } invariant { permitted collection visited index }
```

We traverse an array by keeping in the mutable field `index` an integer value that corresponds to the index of the next element to be enumerated. Such a property is expressed in the first line of `permitted`. The next two lines express the fact that `visited` is always a prefix of `collection`. To define predicate `complete`, we keep the same definition as in the previous section:

```
predicate complete (c: cursor) =                                    WhyML
  length c.visited = length c.collection
```

For convenience, we define `complete` over the `cursor` type.

We now focus on the implementation of the cursor operations. Function `has_next` decides if enumeration is complete by comparing the value of `index` with the length of `collection`:

```
let has_next (c: cursor) : bool                                     WhyML
  ensures { result <-> not (complete c) }
= c.index <> length c.collection
```

Function `next` takes the element `x` on the `index` position, increments the value of `index` and returns `x`:

```
let next (c: cursor) : elt                                          WhyML
  requires { not (complete c) }
  ensures  { c.visited = snoc (old c).visited result }
= let x = c.collection[c.index] in
  ghost c.visited <- snoc c.visited x;
  c.index <- c.index + 1;
  x
```

In the body of function `next`, we update the `visited` sequence, via a ghost writing. Giving the regular status of fields `index` and `collection`, function `next` is type-checked by the Why3 type system as a regular function, even tough there is an assignment to a ghost field. Finally, in function `create` we initialize the cursor as follows:

```
let create (a: array elt) : cursor                                  WhyML
  ensures { result.visited = empty }
  ensures { result.collection = a }
= { visited = empty; collection = a; index = 0 }
```

The `visited` sequence is initially `empty`, the field `collection` is initialized to the array `a`, and the field `index` starts at 0.

To conclude module `CursorArrayImpl`, we show that this module is a valid implementation of module `CursorArraySpec`, via the following `clone` expression:

```
clone CursorArraySpec with                                          WhyML
  type elt, type cursor, val has_next, val next, val create
end
```

We do not need to include predicates `permitted` and `complete` in the substitution since these are already defined in module `CursorArraySpec`. Such a refinement includes a proof that the type invariant of `cursor` in module `CursorArrayImpl` implies the type invariant of `cursor` in `CursorArraySpec`, as well as specification inclusion for operations `has_next`, `next`, and `create`. All the generated verification conditions, either for the implementation of cursor operations and the module refinement, are automatically discharged in no time.

6.2.3 Cursor Client

To illustrate the use of cursors, we give the WhyML implementation of program `sum_array`, which uses a cursor to traverse an array of integer to compute the sum of its elements. We begin by importing an instance of the `CursorArraySpec` module, where occurrences of type `elt` are replaced by type `int`, as follows:

```
clone CursorArraySpec with type elt = int                    WhyML
```

This imposes an abstraction barrier between implementation and client code: the client code uses only the specification of the array cursor, instead of the actual implementation given by the `CursorArrayImpl` module.

We specify `sum_array` with the following contract:

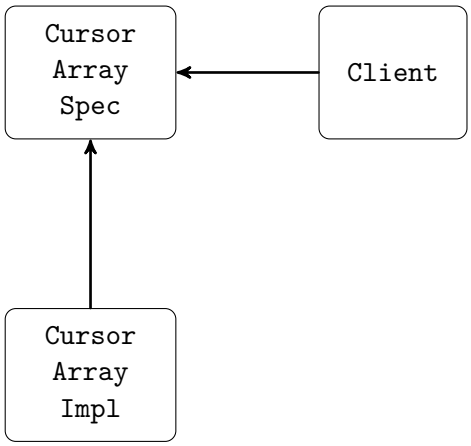
```
let sum_array (a: array int) : int                           WhyML
  ensures { result = sum (fun i -> a[i]) 0 (length a) }
```

Logical function `sum`, of type $(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$, is defined in the Why3 standard library, and stands for the summation of integers values in a given range, *i.e.*, the result of `sum f a b` is equal to $\sum_{\texttt{a} \leq \texttt{i} < \texttt{b}} \texttt{f(i)}$. The code begins by creating a reference `s` to hold the sum, and the cursor `c`. For each new element returned by the cursor, we update the value of `s`, as follows:

```
= let s = ref 0 in                                           WhyML
  let c = create a in
  while has_next c do
    variant   { length c.collection - length c.visited }
    invariant { !s = sum (get a) 0 (length c.visited) }
    let x = next c in
    s := !s + x
  done;
  !s
```

The loop invariant states that reference `s` contains the sum of already enumerated elements. The termination of function `sum_array` is easily proved by annotating the `while` loop with the termination measure `length c.collection - length c.visited`. The type invariant of `cursor` guarantees that `c` is kept in a valid state, which allows to correctly call function `next` and `has_next`. All the generated verification conditions for function `sum_array` are easily discharged.

The relation between the cursor specification, concrete implementation and the client code can be depicted as follows:



The vertical arrow represents the refinement relation between modules `CursorArrayImpl` and `CursorArraySpec`. On the other hand, the horizontal arrow stands for the use of `CursorArraySpec` in the implementation of the client code.

6.2.4 Collection Modification

If we consider mutable data structures, nothing prevents us from modifying its contents while a cursor is traversing it. We could imagine the following code:

```
let c = array_cursor a in                                    WhyML
let x = next c in
a[0] <- 42;
let y = next c in
```

that modifies array `a` after creating the cursor `c`. This is a typical example where we do not want to be able to prove that, at moment of the the second call to `next`, the cursor `c` respects its invariant.

Outside the context of program verification, checking if a data structure is coherent with an associated cursor is typically done via dynamic tests, which inspect a version number included in both the cursor and the data structure. In case the two numbers differ, an exception is raised. This is the case of the Java standard library. In our case, on the contrary, we statically ensure the coherence between a cursor and a collection through the set of generated verification conditions. There is no need for dynamic tests or exception raising which, additionally, has the benefit of producing a more efficient code.

6.2.5 Case Studies

In this section, we present several examples of cursors and associated client code, as well as their formal correctness proof. One key idea is to show that cursors are not limited to data structures traversal. All the given examples are proved correct, with respect to the given specification, in a completely automatic way.

6.2.5.1 Gensym

The first example we present is that of a fresh symbol generator, *i.e.*, a cursor that returns a sequence of values without any repetition. The `permitted` predicate is easily defined, as follows:

```
predicate permitted (v: seq elt) = distinct v                WhyML
```

where `distinct` is a predicate of the Why3 standard library which specifies that the elements of a sequence are pair-wise distinct. The predicate `complete` is even simpler, since this is the case of a non-terminating iteration:

```
predicate complete (v: seq elt) = false                      WhyML
```

We suppose here that the type `elt` is infinitely inhabited, otherwise it would not be possible to implement such a cursor.

If we use the example of a generator for natural numbers, a possible implementation of the `cursor` type is as follows:

```
type elt = int                                               WhyML
type   t = unit

type cursor = {
  ghost mutable visited    : seq elt;
  ghost         collection : t;
        mutable n          : int;
} invariant { permitted visited }
  invariant { forall i. 0 <= i < length c.visited -> n > visited[i] }
```

where field `n` contains the next element in the enumeration. The field `collection` has no meaning here; we keep it just for coherence with our presentation. The function `next` is, thus, easily implemented. It is enough to increment the value of `n`:

```
let next (c: cursor) : elt                                    WhyML
  ...
= let x = c.n in
  c.n <- c.n + 1;
  ghost c.visited <- snoc c.visited x;
  x
```

We omit here the function specification as this does not change with respect to the one we presented in Sec. 6.2.1. The implementation of function `has_next` is trivial, as it always returns `true`, and so we do not show it here.

6.2.5.2 Depth-first Search

The next example is that of a cursor to perform a depth-first search (DFS) traversal of a graph. We represent here a graph through a type `vertex` for the nodes, and the `succ` function that associates to each node the set of its successors:

```
type vertex                                                   WhyML
val function succ vertex : set vertex
```

Note that we declare `succ` as both a logical and programming function, as we shall use it in the implementation and to reason about the cursor's behavior.

The notion of path plays an important role in the specification of such a cursor. We define it as follows:

```
predicate edge (v1 v2: vertex) = mem v2 (succ v1)             WhyML

predicate path (v1: vertex) (s: seq vertex) (v2: vertex) =
  if length s = 0 then v1 = v2
  else edge v1 s[0] /\ s[length s - 1] = v2 /\
    forall i. 0 <= i < length s-1 -> edge s[i] s[i+1]
```

The statement `path v1 s v2` means that there is a path from node `v1` to `v2`, following the nodes contained in sequence `s`. We chose to exclude `v1` from the sequence. Let us note that the definition of `path` is given entirely by means of universal quantification and arithmetic, instead of declaring an inductive or recursive predicate. This is intentionally done, as it leads to a much more automated proof, similarly to what we discussed in Sec. 5.2.2. From the predicate `path` we can deduce the following reachability predicate:

```
predicate reachable (v1 v2: vertex) =                         WhyML
  exists s. path v1 s v2
```

Expressing that we reach vertex `v2` from `v1` is as simple as stating that there is a path between the two, *i.e.*, there exists a sequence `s` for which the statement `path v1 s v2` holds.

As we did for the previous cursors, we start the definition of the DFS cursor by introducing a type `elt` for the elements being enumerated, and a type `t` for the collection:

```
type elt                                                      WhyML
val eq (x1 x2: elt) : bool
  ensures { result <-> x1 = x2 }
```

```
type t = unit

clone Graph with type vertex = elt
```

The type of the collection is of no interest here, since we represent the graph through the `succ` function. The `clone` clause is used here to import an instance of the `Graph` module (which assembles the graph-related definitions, *i.e.*, type `vertex`, function `succ`, predicate `path`, etc.) where every occurrence of `vertex` is replace by `elt`.

To perform the DFS traversal, we follow a traditional approach: we keep a stack of nodes, together with a set of marked nodes that have already been reached by the traversal. We provide the following `cursor` type:

```
type cursor = {                                                    WhyML
  ghost mutable    visited: seq elt;
  ghost         collection: t;
  ghost             source: elt;
        mutable      stack: list elt;
        mutable     marked: set elt;
} invariant { permitted collection visited stack source marked }
```

The `source` field stores the starting node of the traversal, for specification purposes only. The `permitted` property of this `cursor` type is a 5-place predicate, as it is parameterized by the `stack`, the `source` of the traversal, and the set of `marked` nodes, besides the `collection` and the sequence of `visited` elements. The definition of the `permitted` predicate is actually the conjunction of two other predicates, as follows:

```
predicate permitted (collection: t) (visited: seq elt) (stack: list elt)   WhyML
                    (source: elt) (marked: set elt) =
  permitted1 visited stack source marked /\ permitted2 visited marked
```

The reason to split such a definition between predicates `permitted1` and `permitted2` shall become evident when we present the proof of function `next`. For now, let us give the definition of each of these predicates, step-by-step. First, the `permitted1` predicate states that the elements of `stack` are pair-wise distinct,

```
predicate permitted1 (collection: t) (visited: seq elt) (stack: list elt)  WhyML
                     (source: elt) (marked: set elt) =
  distinct stack /\
```

that the `source` is always marked,

```
  mem source marked /\                                             WhyML
```

that the `source` is the head of the `stack` when the traversal starts,

```
  (mem source stack -> stack = Cons source Nil) /\                 WhyML
```

that the elements in the `stack` are always disjoint from the already `visited` elements,

```
  inter (elements stack) (to_set visited) = empty /\               WhyML
```

that the set `marked` elements is the union of the already `visited` elements and those of the `stack`,

```
  marked = union (elements stack) (to_set visited) /\              WhyML
```

and finally that all `marked` nodes are reachable from the `source`,

```
  forall v. mem v marked -> reachable source v                     WhyML
```

As to the `permitted2` predicate, this is much simpler defined, as follows:

WhyML
```
predicate permitted2 (collection: t) (visited: seq elt) (marked: set elt) =
  forall v. mem v visited -> forall w. edge v w -> mem w marked
```

It states that, if a node is already `visited`, then all of its successors are already marked.

The other element of our specification is the `complete` predicate. From a logical point of view, the traversal terminates when all reachable nodes from the `source` have been `visited`. We materialize such an idea in the definition of the `complete` predicate, as follows:

WhyML
```
predicate complete (c: cursor) =
  forall v. reachable c.source v -> mem v c.visited
```

From a programming point of view, we stop the traversal as soon as the `stack` is empty. The implementation of the `has_next` function follows exactly this explanation:

WhyML
```
let has_next (c: cursor) : bool
  ensures  { result <-> not (complete c) }
= match c.stack with Nil -> False | _ -> True end
```

The proof that function `has_next` respects the given specification is harder than it might suggest. To better understand why, let us do a case analysis on the proof of the postcondition:

1. if the `stack` is not empty, then there are yet reachable nodes from the `source` to be enumerated;

2. if the stack is empty, then all of the reachable elements from the `source` have been `visited` during the traversal.

The first property stands for the *correction* of the traversal termination, while the second one represents *completeness*. The correction part is easily established, as follows:

- the `stack` is not empty, and the value `c` respects the `permitted` predicate which gives us, in particular, that all the elements in the `stack` belong to the `marked` set (using property that `marked` is the union of the `stack` and the `visited` sequence);

- also from the definition of `permitted`, we get that for every node `v` in the `marked` set, `v` is reachable from the `source`;

- so, by the fact that the `stack` and the `visited` sequence share no common elements (given, again, by the `permitted` property), we get that there reachable elements from the `source` that are not yet in the `visited` sequence, which corresponds exactly to the statement `not (complete c)`.

SMT solvers are able to follow a similar reasoning to what we have just described, and so this part of the proof is completed fully automatically. The second part, completeness, is much harder. In fact, we need to introduce some extra information in our proof context in order to help provers to succeed. We introduce the following auxiliary lemma:

WhyML
```
lemma path_stack : forall c v s.
  path c.source s v -> not (mem v c.visited) ->
  exists w. mem w c.stack /\ reachable w v
```

Such a lemma ensures that every path from the `source` to a not yet `visited` vertex must contain a vertex from the `stack`. To see how the `path_stack` lemma can be used to complete the proof of completeness of function `has_next`, we reason by contradiction:

- let us suppose that `not (complete c)` holds;
- we get that there exists a node `vv` and a sequence `ss`, such that `path c.source ss vv` and `not (mem vv c.visited)` hold;
- when we instantiate lemma `path_stack` with `c`, `v`, and `s`, we get in our proof context

```
exists w. mem w c.stack /\ reachable w v
```
WhyML

- we know that `stack` is equal to `Nil`;
- so, we get that

```
exists w. mem w Nil /\ reachable w v
```
WhyML

- this is a contradiction (predicate `mem` always returns `false` for an empty list), resulting from assuming `not (complete c)`.

With auxiliary lemma `path_stack`, SMT solvers are able to discharge the verification conditions stating the completeness of the `has_next` function. Now, we turn our attention to the proof the `path_stack` lemma itself. To obtain a fully automatic proof, we write the following lemma function, for which the contract is automatically translated to the above statement:

```
let lemma path_stack (c: cursor) (v: elt) (s: seq elt) : (w: elt)
  requires { path c.source s v }
  requires { not (SM.mem v c.visited) }
  ensures  { LM.mem w c.stack /\ reachable w v }
= if LM.mem v c.stack then v
  else if LM.mem c.source c.stack then c.source
  else begin
    let _, w, _, _ = intermediate_value (is_in_visited c) c.source v s in w
  end
```
WhyML

The proof of `path_stack` is done by case analysis:

1. if `v` is in the `stack`, then it is the vertex we are looking for;
2. if `v` is not in the `stack` but the `source` is the `stack`, then the `source` is the vertex we are looking for;
3. if neither `v` or the `source` are in the `stack`, we then know that the `source` is in `visited` and so there is a path from the inside of `visited` to the outside of `visited` (vertex `v`). Such a path takes an edge $\mathtt{v}' \rightarrow \mathtt{w}$ such that `v'` is in `visited` but not `w`. Using the `permitted` property, we know that `w` is necessarily in the `stack`, and so this is the vertex we are looking for.

We use function `intermediate_value` to find vertex `w` in the third case. This is a ghost function used to retrieve the first vertex of a path between the vertices `u` and `v` that does not verify a given property $\mathcal{P}$, knowing that $\mathcal{P}(\mathtt{u})$ and $\neg\mathcal{P}(\mathtt{v})$ hold. In the case of `path_stack`, we call `intermediate_value` with the property *to be in the visited sequence*, defined as follows:

```
function p (c: cursor) : elt -> bool =
  fun x -> SM.mem x c.visited

val ghost is_in_visited (c: cursor) : elt -> bool
  ensures { result = p c }
```
WhyML

Function `intermediate_value` is implemented as the following recursive function:

```
let rec ghost intermediate_value                                    WhyML
  (p: vertex -> bool) (u v: vertex) (s: seq vertex) :
  (u': vertex, v': vertex, s1: seq vertex, s2: seq vertex)
requires { p u }
requires { not (p v) }
requires { path u s v }
ensures  { p u' /\ not (p v') /\ path u s1 u' /\ path v' s2 v /\ edge u' v' }
variant  { length s }
= if length s = 0 then absurd
  else if not (p s[0]) then (u, s[0], empty, s[1 ..])
  else let (u', v', s1, s2) = intermediate_value p s[0] v s[1 ..] in
    (u', v', cons s[0] s1, s2)
```

It is worth point out that `intermediate_value` takes as the last argument a sequence `s` to establish property `path u s v`, instead of using the `reachable` predicate. Using such a sequence we are able to implement the `intermediate_value` function by recursion over the structure of `s`. All the verification conditions generated for `path_stack` and `intermediate_value` are automatically discharged, which concludes the proof of function `has_next`.

We now focus on the proof of function `next`. This function follows a much traditional approach: we take the head element `v` of the `stack` (the pre-condition assures that `stack` is not empty), then we iterate over the successors of `v`, adding them to the `stack` and the set of `marked` vertices. We give the following WhyML implementation:

```
let next (c: cursor) : elt                                          WhyML
  requires { not (complete c c.collection) }
  writes   { c }
  ensures  { c.visited = snoc (old c.visited) result }
= match c.stack with
  | Nil      -> absurd
  | Cons v r ->
      c.stack   <- r;
      c.visited <- snoc c.visited v;
      let s = ref (succ v) in
      while not (is_empty !s) do
        variant { cardinal !s }
        let x = choose !s in
        s := remove x !s;
        if not (mem x c.marked) then begin
          c.stack  <- Cons x c.stack;
          c.marked <- add x c.marked end
      done;
      v
  end
```

It is worth pointing out that the `absurd` point is provable thanks to the `path_stack` lemma. The most interesting part of the proof of `next` is to provide the invariant to the `while` loop. Let us go through in detail over such an invariant: the `source` vertex is not in the `stack`,

```
invariant { not (mem c.source c.stack) }                            WhyML
```

during the iteration, the cursor `c` respects only the `permitted1` property, since we cannot state that all successors of elements in `visited` are `marked` as, in fact, we are using the `while` loop to insert all successors of `v` in the `stack` and the `marked` set,

```
invariant { permitted1 c.visited c.stack c.source c.marked }
```
WhyML

nonetheless, all the successors of `visited` vertices different from `v` are `marked`,

```
invariant { forall u. mem u c.visited -> u <> v ->
            forall w. mem w (succ v) -> mem w c.marked }
```
WhyML

and, as well, all vertices already taken from the initial set `!s` of successors of `v` is `marked`,

```
invariant { forall w. mem w (diff (succ v) !s) -> mem w c.marked }
```
WhyML

set `!s` is a subset of the successors of `v`,

```
invariant { subset !s (succ v) }
```
WhyML

and finally all vertices in `!s` are reachable from the `source`,

```
invariant { forall x. mem x !s -> reachable c.source x }
```
WhyML

All the generated verification conditions are proved in no time by a combination of SMT solvers. It is interesting to note how we are able to frame the set of successor vertices that are incrementally included in the `marked` set. After the loop, as we know that `!s` is empty, the provided invariant is sufficient to prove that all the successors of `v` are included in the `marked` set. This proves the definition of the `permitted` predicate, which re-establishes the invariant of type `cursor`.

As an example of a client code to the DFS cursor, we give the following WhyML program that decides whether there is a path between two given vertices:

```
let is_path (v w: elt) : bool
  ensures { result <-> Graph.reachable v w }
  diverges
= let it = create v in
  while has_next it do
    invariant { not (mem w it.visited) }
    let x = next it in
    if eq x w then return True
  done;
  False
```
WhyML

We keep iterating as long as function `has_next` returns `True`, *i.e.*, while the cursor can provide new elements. In the case of an infinite graph, we can indeed iterate forever, which prevents us from proving termination of the `is_path` function. The **`diverges`** keyword accounts for the possibility of divergence. The loop invariant assures that, as long as we iterate, vertex `w` is not `visited`. If the loop terminates, then we can use the invariant to prove that vertex `w` is not reachable from `v`. On the other hand, if we reach vertex `w` during the traversal, then we use the **`return`** instruction to interrupt iteration and return `True`.

6.2.5.3 In-order Traversal of Binary Trees

The last example we give in this section is a cursor to perform an in-order traversal of a binary tree. This example comes with a bonus: we show how to use the WhyML modules system not only to prove refinements between cursors specification and implementation, but also to get, at the end, an executable code linking a client and a specific implementation of the cursor, even tough the WhyML client code knows only about the cursor specification.

Cursor specification. The polymorphic type of binary trees is defined in Why3 standard library, as follows:

```
type tree 'a = Empty | Node (tree 'a) 'a (tree 'a)
```
WhyML

The `tree 'a` type is defined together with the following logical function `elements`

```
function elements (t: tree 'a) : seq 'a = match t with
  | Empty      -> empty
  | Node l x r -> elements l ++ cons x (elements r)
end
```
WhyML

which specifies the sequence of the tree elements when traversed in-order.

We begin with the logical specification and declaration of the in-order cursor operations. We encapsulate the WhyML code in the `InorderCursorSpec`, which starts with the declaration of the `elt` and `t` types:

```
module InorderCursorSpec

  type elt

  val eq (x1 x2: elt) : bool
    ensures { result <-> x1 = x2 }

  type t = tree elt
```
WhyML

Note that we equip the elements of type `elt` with an equality operation, defined via the abstract function `eq`. We need to introduce such an operation since polymorphic equality is, in WhyML, exclusively used for specification purposes. Next, we add to the `InorderCursorSpec` module the definitions of predicates `permitted` and `complete`, as follows:

```
  predicate permitted (t: t) (v: seq elt) =
    length v <= length t /\
    forall i. 0 <= i < length v -> v[i] = t[i]

  predicate complete (t: t) (v: seq elt) =
    length t = length v
```
WhyML

Function `elements` is declared in the Why3 standard library as a coercion, which allows us to write very clean and compact definitions for the above predicates. Every application of the `t` symbol to a logical function is automatically translated to `(elements t)`, the infix sequence of the tree elements. The `permitted` relation simply states the `visited` sequence is a prefix of the in-order sequence of the collection; the traversal is `complete` once the length of the elements sequence of `t` is the same as the length of the `visited` sequence.

The type of the in-order cursor keeps the two ghost fields `visited` and `collection`, where we know that `collection` is of type `tree elt`, as follows:

```
  type cursor = private {
    ghost mutable    visited: seq elt;
    ghost         collection: t;
  } invariant { permitted collection visited }
    by { visited = empty; collection = Empty }
```
WhyML

We use predicate `permitted` as the type invariant of `cursor` and the invariant witness is simply the record where fields `visited` and `collection` are assigned, respectively, to the `empty` sequence

and to the `Empty` tree. We are now in position to declare the cursor operations `next` and `has_next`, whose contracts follow exactly those of the general specification of a cursor:

```
val next (c: cursor) : elt                                        WhyML
  requires { not (complete c.collection c.visited) }
  writes   { c.visited }
  ensures  { c.visited = snoc (old c).visited result }

val has_next (c: cursor) : bool
  ensures { result <-> not (complete c.collection c.visited) }
```

We introduce, as well, the following operation to create a value of type `cursor`:

```
val create (t: t) : cursor                                        WhyML
  ensures { result.collection = t }
  ensures { result.visited = empty }
```

This creates a cursor with an empty `visited` sequence, making the cursor ready to use and return the leftmost element of the tree the first time we call function `next`[3]. The collection of the cursor is the `elements` sequence of the `t`.

Cursor Implementation. We now turn our attention to the actual implementation of an in-order cursor. We encapsulate the implementation of the in-order cursor in a module named `InorderCursorImpl`, which begins as follows:

```
module InorderCursorImpl                                          WhyML

  type elt

  val eq (x1 x2: elt) : bool
    ensures { result <-> x1 = x2 }

  type t = tree elt
```

To implement the cursor operations performing an in-order traversal, we use the *zipper* [78] structure. Since we only perform descents towards the left sub-tree, we specialize the zipper data type as follows:

```
type zipper = Done | Next elt (tree elt) zipper                   WhyML
```

An element of type `zipper` forms a list where each element is a pair composed of an element and the corresponding right-hand sub-tree. We build such a list bottom-up, which represents the left part of the tree that is still to be traversed. Together with the `zipper` data type, we introduce the following logical function to convert from a `zipper` to a sequence:

```
function zipper_elements (e: zipper) : seq elt = match e with    WhyML
  | Done -> empty
  | Next x r e -> cons x (elements r ++ zipper_elements e)
  end

meta coercion function zipper_elements
```

[3]We could, very well, imagine a creation function which would place the cursor in a specific position of tree, instead of always *placing* it on the leftmost element.

We use the `zipper_elements` function to define predicate `permitted`, as follows (the coercion system avoids its explicit application):

```
predicate permitted (t: t) (visited: seq elt) (zipper: zipper) =
  t = visited ++ zipper
```
WhyML

This definition of `permitted` acts as a *linking invariant* in the following definition of type `cursor`:

```
type cursor = {
  ghost mutable    visited: seq elt;
  ghost         collection: t;
        mutable     zipper: zipper;
} invariant { permitted collection visited zipper }
  by { visited = empty; zipper = Done; collection = Empty }
```
WhyML

This type `cursor` contains the zipper in a *regular* mutable field, since we want to preserve the zipper after extraction. Predicate `complete` is straightforwardly defined as follows:

```
predicate complete (t: t) (v: seq elt) =
  length t = length v
```
WhyML

We can now define the cursor operations. Starting with the creation of a new element of type `cursor`, we first define the following operation to create the zipper for a given tree:

```
let rec function zipper_build (t: tree elt) (z: zipper) : zipper
= match t with
  | Empty      -> e
  | Node l x r -> zipper_build l (Next x r e)
end
```
WhyML

Function `zipper_build` simply traverses the tree all the way down to the left, collecting each element and its right-hand sub-tree, using argument `z` as an accumulator. The use of the `function` keyword instructs Why3 to automatically generate the logical counter-part of `zipper_build`. The advantage of such automatic encoding is two-fold: first, there is no need for us to write a formal contract for function `zipper_build`, as Why3 infers the strongest postcondition of such a function; second, it allows to use and reason about the definition of `zipper_build` in the logic of WhyML. For instance, the following lemma relates the result of `zipper_build t z` with the sequence formed by the arguments `t` and `z`:

```
let rec lemma zipper_spec (t: tree elt) (z: zipper)
  ensures { zipper_build t z == t ++ z }
  variant { t }
= match t with
  | Empty      -> ()
  | Node l x r -> zipper_spec l (Next x r z)
  end
```
WhyML

This lemma states that if we convert the zipper `zipper_build t z` to a sequence (function `zipper_elements` is applied as a coercion in the postcondition of the lemma) we get exactly the sequence `t ++ z` (functions `elements` and `zipper_elements` are here used as coercions). The proof is done by straightforward induction on the tree structure. Lemma `zipper_spec` is crucial to prove that the type invariant of a `cursor` containing a newly created `zipper` holds indeed. This is exactly the case for the function that creates a new in-order cursor:

```
let create (t: tree elt) : cursor
  ensures { result.collection = t }
```
WhyML

```
    ensures { result.visited = empty }
  = { visited = empty; zipper = zipper_build t Done; collection = t }
```

All the generated verification conditions for function `iterator` are automatically discharged. This includes a proof that the new value of type `cursor` respects the type invariant, which is possible thanks to the `zipper_spec` lemma. For the `next` function, each time we return a new element we need to rebuild the zipper from the left-hand sub-tree of the returned element, together with the remaining of the zipper:

WhyML
```
  let next (c: cursor) : elt
    requires { not (complete c) }
    ensures  { c.visited == snoc (old c).visited result }
  = match c.zipper with
    | Done -> absurd
    | Next x r e -> c.visited <- snoc c.visited x;
                    c.zipper <- zipper_build r e;
                    x
    end
```

The proof that the updated cursor respects the type invariant is automatically done by SMT solvers using, once again, the `zipper_spec` lemma. The remaining verification conditions for the `next` function are easily discharged by SMT solvers. Finally, we give the definition of function `has_next`. This is implemented as follows:

WhyML
```
  let has_next (c: cursor) : bool
    ensures { result <-> not (complete c) }
  = match c.zipper with
    | Done -> false
    | _    -> true
    end
```

The enumeration of the tree elements is complete once the zipper is equal to `Done`. Module `InorderCursorImpl` terminates with the proof that this particular implementation of a in-order cursor is a refinement of the module `InorderCursorSpec`. This is achieved via the following `clone` clause:

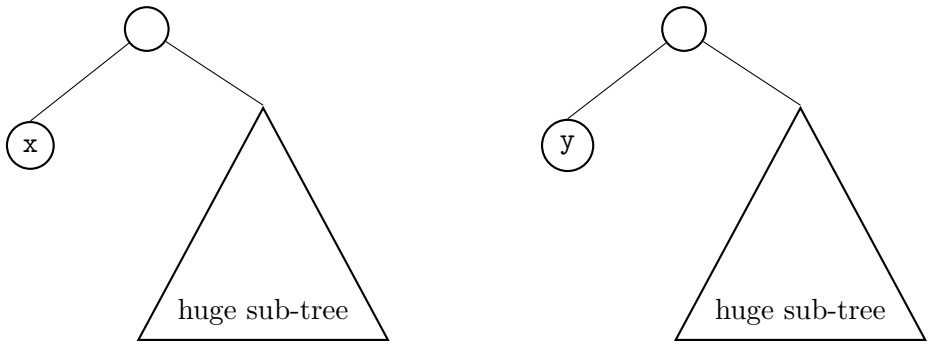
WhyML
```
  clone InorderCursorSpec with
    type elt, type cursor,
    val next, val has_next, val create

end
```

All the generated verification conditions are automatically discharged, which include the refinement of type `cursor`, *i.e.*, the type invariant of `InorderCursorImpl.cursor` implies the type invariant of `InorderCursorSpec.cursor`, as well the proof of specification inclusion for operations `next`, `has_next`, and `createa`.

Cursor client. We present now a client program using the in order cursor. We chose the classical algorithmic problem known as *same fringe*, which consists in deciding whether two binary trees present the same sequence of elements, when traversed in-order. In other words, we seek here to write a function with the following contract:

WhyML
```
let same_fringe (t1 t2: tree elt) : bool
  ensures { result <-> elements t1 = elements t2 }
```

A possible implementation for the `same_fringe` function would be to explicitly build the in-order sequence of each tree and then compare them. This is, however, an inefficient solution. Let us consider, for instance, the two following binary trees:



The two trees differ in the leftmost element, as we have `x` for the left-hand side tree, and `y` for the right-hand side tree. Following the solution proposed above, we would build two (huge) sequences, which is a shame. Even tough this is a linear solution on the number of elements of the trees, this has the pitfall of building in memory the two sequences, even if the two trees differ directly on the first element issued from the in-order traversal. Another solution would be to build just the sequence of one of the trees, then traverse the second one and comparing each new element issued from the traversal with the head of the sequence. One would still have to explicitly build in memory a sequence of elements, other than the two trees, which can be prohibitive under some circumstances. We propose to explore here a different approach: we build a cursor for the two trees, and we enumerate the elements of each tree while both are equal, with respect to the `eq` operation. Therefore, we stop the traversal as soon as two elements are different. This is both an elegant and an efficient solution. The following `eq_cursors` function implements such a comparison:

WhyML

```
module SameFringe

  clone InorderCursorSpec as CI

  predicate permitted (c: CI.cursor) =
    CI.permitted c.CI.collection c.CI.visited

  let rec eq_cursors (c1 c2: CI.cursor) : bool
    requires { c1.CI.visited == c2.CI.visited }
    ensures  { result <-> c1.CI.collection == c2.CI.collection }
    variant  { length c1.CI.collection - length c1.CI.visited }
  = match CI.has_next c1, CI.has_next c2 with
    | False, False -> True
    | True, True   -> CI.eq (CI.next c1) (CI.next c2) && eq_cursors c1 c2
    | _            -> False
    end
```

In the `eq_cursors` function, we ask each cursor for a new element and compare them. If the two elements differ from each other, we immediately return false and stop the traversal (`&&` is a lazy operand in WhyML). Otherwise, we keep comparing the two cursors just until one of two situations happens: either the `has_next` signals the end of the traversal for both cursors, in which case we return `True`; or one of the cursors stops before the other, in which case we return `False`. We encapsulate our solution to the same-fringe problem in the WhyML module `SameFringe`. Note that we import the definitions of the `InorderCursorSpec` module into the `SameFringe` module.

This means that we establish here an abstraction barrier: we implement and formally prove a client code for our in-order cursor knowing only about the specification of such a cursor. This follows our approach of modular proofs that we discuss in Chap. 5. Using function `eq_cursors`, we can easily deduce the implementation of `same_fringe`, as follows:

WhyML
```
  let same_fringe (t1 t2: tree CI.elt) : bool
    ensures { result <-> elements t1 = elements t2 }
  = eq_cursors (CI.iterator t1) (CI.iterator t2)

end
```

All the generated verification conditions for functions `eq_cursors` and `same_fringe` are easily discharged by SMT solvers.

Putting all together. We complete our presentation of the specification, implementation, and use of a cursor to traverse a binary tree in-order by showing how we can link the client code with a specific implementation of such a cursor. We follow the approach we presented in Sec. 6.2.3: we use the `clone` instruction to create an instance of the `SameFringe` module, where each operation and declaration of the `InorderCursorSpec` module is replaced by the counter-part of the `InorderCursorImpl`. We write the following WhyML module:

WhyML
```
module Linking

  clone SameFringe with
    type CI.elt      = CImpl.elt,
    type CI.cursor   = CImpl.cursor,
    val  CI.iterator = CImpl.iterator,
    val  CI.has_next = CImpl.has_next,
    val  CI.next     = CImpl.next,
    val  CI.eq       = CImpl.eq

end
```

This use of WhyML `clone` clause feels much like OCaml functors application. Indeed, we can view module `InorderCursorSpec` as the argument (named `CI`) of the `SameFringe` functor, *i.e.*,

OCaml
```
module SameFringe (CI: sig type elt type cursor val iterator ... end) = struct
  let eq_cursors (c1 c2: CI.cursor) : bool = ...
  let same_fringe (t1 t2: tree CI.elt) : bool =
    eq_cursors (CI.iterator t1) (CI.iterator t2)
end
```

and the `clone SameFringe with` line would correspond to

OCaml
```
module Linking = SameFringe (CImpl)
```

Thanks to this linking operation, we are able to extract an executable OCaml implementation of the same-fringe function, using the in order cursor. Supposing the WhyML code is contained in a Why3 file named `inorder_cursor`, the following command can be used to extract the code of the `Linking` module:

Terminal
```
> why3 extract -L . -D ocaml64 -D sf.drv --recursive inorder_cursor.Linking
```

cursor	loc	los	time (sec)
gensym	12	30	0.03
array	12	23	0.05
list	15	28	0.40
set	12	22	13.74
binary tree	36	72	0.21
merge	36	75	2.83
dfs	48	85	11.02
total	171	335	

(a) Cursor Implementations

program	loc	los	time (sec)
array sum	8	12	0.70
list length	8	4	0.03
search	8	10	0.10
same fringe	36	72	0.21
check path	11	4	1.25
merge cursors	36	75	2.83
mjrty	32	22	1.67
total	139	199	

(b) Cursor Clients

Table 6.1: Experimental Results.

The result of running such a command is given in Fig. 6.1. It is worth pointing out that there is currently no mechanism in Why3 to verify that a `clone` instruction replaces all the undefined symbols of a WhyML module. This prevents us from extracting OCaml code containing functor applications. Indeed, the code in Fig. 6.1 represents the *defunctorized* code of an hypothetical application of functor `SameFringe` to its argument. Let us note the use of the custom extraction driver `sf.drv` in the `extract` command line. Such a driver simply defines that every occurrence of the `eq` operation is to be replaced by the OCaml polymorphic equality, as follows:

Driver

```
module inorder_cursor.InorderCursorImpl
  syntax val eq "%1 = %2"
end
```

We note, as well, that similarly to the example of the array cursor (Sec. 6.2.3, page 145), we pass the `--recursive` option to the `extract` command, in order to extract all the dependencies of the `Linking` module. Without such an option, we would end up with only the extracted code of the `eq_cursors` and `same_fringe` functions.

6.2.6 Other Case Studies

Table 6.1a shows the lines of code, the lines of specification (functions contracts, invariants, and auxiliary lemmas), and the total verification time (in seconds) for each cursor.

We have also implemented and verified a number of client programs that use cursors. We always keep the proof of client code modular in respect to the cursor implementation, *i.e.*, the client programs are only using the cursor interface and have no access to the underlying implementation. Our programs include summing the elements of an array, computing the length of a list, searching for a particular element in some abstract collection, checking for the existence of a path in a graph using a DFS cursor, merging two ordered sequences, and implementing Boyer & Moore's "mjrty" algorithm [22] using array cursors. Table 6.1b shows the lines of code, the lines of specification, and the total verification time (in seconds) for each program.

For all the listed examples, both implementations of cursors and client codes, all generated verification conditions are discharged automatically, using a combination of the SMT solvers.

6.3 Higher-Order Iteration

In programming languages featuring first-class functions, iteration is frequently implemented as a higher-order function that takes as argument a function to be applied to each element of the

OCaml (extracted)

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree

type elt

type t = elt tree

type zipper =
  | Done
  | Next of elt * elt tree * zipper

type cursor = {
  mutable zipper1: zipper;
  }

let has_next (c: cursor) : bool =
  begin match c.zipper1 with
  | Done -> false
  | _ -> true
  end

let rec zipper_build (t1: elt tree) (e: zipper) : zipper =
  begin match t1 with
  | Empty -> e
  | Node (l, x, r) -> zipper_build l (Next (x, r, e))
  end

let next (c: cursor) : elt =
  begin match c.zipper1 with
  | Done -> assert false (* absurd *)
  | Next (x, r, e) -> begin let o = zipper_build r e in c.zipper1 <- o; x end
  end

let rec eq_cursors (c1: cursor) (c2: cursor) : bool =
  begin match (has_next c1, has_next c2) with
  | (false, false) -> true
  | (true, true) ->
    (let o = next c2 in let o1 = next c1 in o1 = o) && (eq_cursors c1 c2)
  | _ -> false
  end

let iterator (t1: elt tree) : cursor = { zipper1 = (zipper_build t1 Done) }

let same_fringe (t1: elt tree) (t2: elt tree) : bool =
  eq_cursors (iterator t1) (iterator t2)
```

Figure 6.1: In-Order Cursor and Same-fringe Extracted Code.

enumerated sequence. We refer to such function as the *consumer function*. A common example is the `fold` function [79]. We consider here `fold` to be a function of three arguments, with the following type:

$$\texttt{fold} : (acc \to elt \to acc) \to collection \to acc \to acc$$

The second argument is the collection to be iterated over, of type *collection*. The first argument is the function to be applied in turn to each element of the collection. This function takes two arguments, an accumulator of type *acc* and an element of type *elt*, and returns an updated value for the accumulator. The accumulator stands for the value being computed by `fold` and its initial value is the third argument of `fold`. Thus, if the elements of a collection `c` are traversed in order $x_1, \ldots, x_n$, a call to `fold f c a` amounts to evaluating $\texttt{f}(\ldots(\texttt{f}\ (\texttt{f}\ \texttt{a}\ x_1)\ x_2)\ldots)\ x_n$. For instance, assuming the elements of `c` are integers, we can sum them as simply as

$$\texttt{fold}\ (\lambda\, \texttt{s}\, x.\ \texttt{s} + x)\ \texttt{c}\ 0$$

where λ introduces an anonymous function.

The concept of `fold` is ubiquitous in functional programming languages. The OCaml standard library, for instance, offers many different instances of `fold` iterators for data structures such as lists, arrays, or maps. The recent introduction of closures in languages such as C++ and Java eases this style of programming. In these languages, higher-order iterators coexist with cursors, allowing the user to choose the paradigm that suits best. The main difference between the two is that control is given to the producer in the case of a higher-order iterator, while it is given to the consumer in the case of a cursor.

In this section, we use predicates *permitted* and *complete* in order to specify and verify higher-order iterators. As we did with cursors, we intend to verify both implementations of `fold` functions (Sec. 6.3.1) and client code (Sec. 6.3.2). One way to tackle the verification of higher-order functions is to use a higher-order (program) logic, in such a way that one can quantify over the specification of function arguments. There exist already several systems in which we can do so; we will discuss those in Sec. 5.5. We consider here a different approach, which only requires first-order logic. This is possible thanks to the *abstraction barrier* provided by the `permitted`/`complete` predicates. On both sides of this interface, we are making distinct first-order program proofs, one for the implementation of `fold` and one for each call to `fold`. We present several case studies for our methodology (Sec. 6.3.3 and 6.3.4).

6.3.1 Fold Implementation

We use the `fold_left` function over lists, from the OCaml standard library, as the guiding line to the presentation of our proposition for the specification and proof of fold-like iterators. This is implemented as follows:

OCaml
```
let rec fold_left f acc l = match l with
  | []     -> acc
  | e :: r -> fold_left f (f acc e) r
```

We scan the list `l` from left to right, applying the consumer function `f` to `acc`, the current value of the accumulator, and the next element in the iteration.

Similarly to what we did for cursors, we want to specify iteration performed by `fold` iterators by means of the `permitted` and `complete` predicates. For the case of the `fold_left` function, it is easy to derive the definition of such predicates: it is valid to iterate while the visited sequence forms a prefix of the whole list, and we are done as soon as the length of the visited sequence is equal to the length of the list being iterated, *i.e.*,

```
function elements (l: list 'a) : seq 'a = match l with                    WhyML
  | Nil -> empty
  | Cons x r -> cons x (elements r)
  end
meta coercion function elements

predicate permitted (v: seq 'a) (l: list 'a) =
  length v <= length l /\
  forall i. 0 <= i < length v -> v[i] = l[i]

predicate complete (v: seq 'a) (l: list 'a) =
  length v = length l
```

We can use the predicates `permitted` and `complete` to specify the `fold_left` implementation, as follows:

```
type elt                                                                   WhyML
type acc

let rec fold_left (ghost v: seq elt) (ghost l0: list elt) (f: acc -> elt -> acc)
  (acc: acc) (l: list elt) : (ghost vres: seq elt, accu: acc)
  variant  { l }
  requires { permitted v l0 }
  requires { l0 = v ++ l }
  ensures  { permitted vres l0 }
  ensures  { vres = v ++ l }
= match l with
  | Nil      -> (v, acc)
  | Cons x r -> fold_left (snoc v x) l0 f (f acc x) r
  end
```

Let us note that in the above implementation we assume `f` to be a pure function, since Why3 does not feature support for higher-order stateful computations. We add to the regular arguments of `fold_left` two extra ghost arguments: sequence `v` represents the sequence of visited elements; list `l0` is the whole list being traversed, as we need it as an argument of the `permitted` property. From the first pre-condition and postcondition we deduce that `permitted` acts as an *invariant of the traversal*. On the other hand, the second pre-condition states that at the entry of every recursive call to the `fold_left` function, sequence `v` forms a prefix of `l0`. Finally, the last postcondition ensures that this is a deterministic traversal: we enumerate the elements of `l`, and only those.

When we feed Why3 with the above implementation and specification of `fold_left`, all the generated verification conditions are automatically discharged by SMT solvers. However, the given specification is of no use when we write a client program of `fold_left`. We must characterize the returned value of the accumulator, which in turn leads us to the specification of the consumer function `f`. This is where we need to introduce the notion of *client invariant*: as a `fold`-like function encapsulates an iteration, we must provide a loop invariant to `fold_left`, which represents the evolution of the iteration from the point of view of a client code. In fact, we manipulate here two different sorts of invariants: the `permitted` property, which we name the *iterator invariant*, and the client invariant. We add to `fold_left` the extra argument `inv`, a function of type `acc -> seq elt -> bool`, as well as the following specification over `f`, `acc`, and `inv`:

WhyML

```
let rec fold_left (ghost v: seq elt) (ghost inv: acc -> seq elt -> bool)
  (ghost l0: list elt) (f: acc -> elt -> acc) (acc: acc) (l: list elt) :
  (ghost vres: seq elt, accu: acc)
  ...
  requires { inv acc v }
  requires { forall v acc x.
               inv acc v -> permitted (snoc v x) l0 -> not (complete v l0) ->
               inv (f acc x) (snoc v x) }
  ...
  ensures  { inv accu vres }
```

The first new pre-condition and the new postcondition ensure that `inv` is, indeed, and invariant property of the `fold_left` function. The other added pre-condition specifies the behavior of the consumer function `f` over the accumulator value. For every sequence `v`, accumulator value `acc`, and an element `x` of the list, if `inv` holds for `acc` and `v`, the element `x` is the next in the iteration (`permitted` holds for `snoc v x`), and the traversal is not yet complete, then a call to `f` preserves the `inv` property for the sequence `snoc v x`. We have now a complete specification of the `fold_left` iterator. To take into account the extra argument `inv`, we update the implementation as follows:

WhyML

```
= match l with
  | Nil      -> (v, acc)
  | Cons x r -> fold_left (snoc v x) inv l0 f (f acc x) r
  end
```

All the generated verification conditions for this program are automatically discharged, which completes the proof of `fold_left` in terms of the `permitted`, `complete`, and `inv` predicates.

A first-order approach. Even if the described implementation gives us a complete specification of the behavior of a `fold`-like iterator, we argue that this introduces a cumbersome WhyML development. The `fold_implementation` gets difficult to handle, since we have introduced several ghost arguments (which we have to instantiate in every client code), and the function contract might be hard to follow. We believe that, from a WhyML programmer perspective, it would be more natural and amenable to abstract out some of the `fold_left` elements and create an interface to be cloned by clients of the iterator, just as we did for cursors. Let us precise our statement: the `fold_left` implementations always acts on the same values of the `inv` predicate, the list `l0`, and the consumer function `f`. These symbols are defined once and for all, which makes them good candidates to be *lifted* to top-level symbols of a WhyML module. These can be later instantiated to match the needs of a particular client code of `fold_left`. Following such an approach, we devise module `FoldLeft` and introduce the `l0` constant, as follows:

WhyML

```
module FoldLeft

  type elt
  type acc

  val constant l0 : list elt
```

Next, we declare `inv` as the following undefined predicate:

WhyML

```
  predicate inv (seq elt) acc
```

and the consumer function, using the same definitions for `permitted` and `complete`, as follows:

```
predicate permitted (v: seq 'a) (l: list 'a) =                    WhyML
  length v <= length l /\
  forall i. 0 <= i < length v -> v[i] = l[i]

predicate complete (v: seq 'a) (l: list 'a) =
  length v = length l

val f (v: seq elt) (acc: acc) (x: elt) : acc
  requires { inv v acc }
  requires { permitted (snoc v x) l0 }
  requires { not (complete v l0) }
  ensures  { inv (snoc v x) result }
```

We note that it is possible to turn `f` into a global function of the program, since every recursive call to `fold_left` is done on the same argument `f`. We note, as well, that this leads us to an entirely first-order proof. The contract of function `f` is, in our opinion, easier to understand with respect to when it was embedded in the contract of the `fold_left` function itself. From the point of view of `fold_left`, function `f` and predicate `inv` remain undefined, as it is up to the client to provide particular instances according to each use case. Finally, we can derive the following, much simpler, WhyML implementation of `fold_left`:

```
let rec fold_left (ghost v: seq elt) (acc: acc) (l: list elt) :   WhyML
  (ghost vres: seq elt, accu: acc)
  variant  { l }
  requires { permitted v    l0 /\ l0   = v ++ l /\ inv v    acc  }
  ensures  { permitted vres l0 /\ vres = v ++ l /\ inv vres accu }
= match l with
  | Nil      -> (v, acc)
  | Cons x r -> fold (snoc v x) (f v acc x) r
  end
```

Such an implementation features only three arguments (with only the sequence `v` as a ghost argument), which is much closer to its OCaml counterpart. The contract is simplified, as well, since there is no need to reason about invariant `inv` and with the postcondition almost mimicking the pre-condition. When fed to Why3, this program generates a set of verification conditions that are easily proved by SMT solvers.

In order to test our verified implementation of `fold_left`, we can use the following `fold_correct` function:

```
let fold_correct (acc: acc) : (ghost vres: seq elt, accu: acc)    WhyML
  requires { permitted empty l0 /\ inv empty acc }
  ensures  { permitted vres  l0 /\ inv vres  accu /\ complete vres l0 }
= fold_left empty acc l0

end
```

This simply calls `fold_left` to perform an iteration over list `l0`. When the iteration halts, we can prove that the `complete` property holds. This completes the definition of module `FoldLeft`.

6.3.2 Fold Client

In this section, we detail on a client program to our implementation of the `fold_left` iterator. Our use case is the `list_length` function, an implementation that calls `fold_left` to compute the number of elements in the list.

We begin a `FoldLeftClient` module by defining a function `f`, to be used as the consumer function, as follows:

```
module FoldLeftClient                                                    WhyML

  type elt
  type acc = int

  let f (ghost v: seq elt) (accu: acc) (x: elt) : acc
  = accu + 1
```

There is no need to specify function `f`: Why3 can automatically infer a specification that corresponds exactly to the body of the function. Next, we give the following definition to the `inv` predicate:

```
  predicate inv (acc: acc) (v: seq elt) =                                WhyML
    acc = length v
```

This states that any moment, during the iteration, the value of `acc`, the accumulator, is equal to the number of elements in `v`, *i.e.*, we count each element of the list one and only once.

We can now instantiate the `FoldLeft` module with the elements we have just defined. This is done via the following **`clone`** expression:

```
  clone FoldLeft with                                                    WhyML
    type elt, type acc, predicate inv, val f
```

Let us note that there is no need to introduce and then instantiate constant `l0`. For every symbol missing in the **`clone`** substitution, Why3 automatically uses the one defined in the `FoldLeft` module in the remaining of the WhyML development. To instantiate function `f`, Why3 generates verification conditions that stand for the specification inclusion of such a function. These are easily discharged, since the inferred postcondition for the implementation of `f` straightforwardly implies the postcondition given to `f` in the `FoldLeft` module.

We can, finally, implement and specify function `list_length`, as follows:

```
  let list_length () : (ghost vres: seq elt, accu: int)                 WhyML
    ensures { accu = length l0 }
  = fold_left empty 0 l0

end
```

This function takes no argument, as every needed element is a top-level symbol of the same WhyML module. It is worth pointing out that, even if we know the definition of function `f`, after module cloning Why3 only uses the specification of `f` from the `FoldLeft` module to generate verification conditions for `list_length`. Thanks to the specification declared for `f`, the definitions of `permitted`, `complete`, and the `inv` predicates, SMT solvers are able to discharge in no time the verification conditions generated for `list_length`.

6.3.3 Case Studies

We present here some more case studies, employing different data structures and `fold` iterators, in order to experimentally validate our proposal based on the `permitted`/`complete` pair, as well as on the Why3 modules system to build a clearer proof.

6.3.3.1 Binary trees

Fold implementation. The first case study we present is the use of a `fold`-like function to traverse and compute the number of elements of a binary tree. This example is inspired by the work of Yann Régis-Gianas and François Pottier [133].

We introduce module `FoldTree` to encapsulate the WhyML implementation of the `fold` iterator over binary trees. The structure of this module is very similar to that of `FoldLeft`, presented in the previous section. We introduce a constant `t0`, the whole tree being iterated over, predicate `inv`, and the consumer function `f`, which presents the very same contract as the consumer function of `fold_left`:

WhyML
```
module FoldTree

  type elt
  type acc

  val constant t0 : tree elt

  predicate inv (seq elt) acc

  val f (ghost v: seq elt) (acc: acc) (x: elt) : acc
    requires { inv acc v }
    requires { permitted t0 (snoc v x) }
    requires { not (complete t0 v) }
    ensures  { inv (snoc v x) result }
```

Predicates `permitted` and `complete` are defined exactly as in Sec. 6.2.5.3. The signature of the `fold` function is exactly the same as the one of `fold_left`:

WhyML
```
  let rec fold (ghost v: seq elt) (acc: acc) (t: tree elt) :
    (ghost vres: seq elt, accu: acc)
```

When it comes to specification, there are some subtle differences between the two iterators. In the case of `fold_left`, at any moment we can rebuild the entire list from the elements of sequence `v` and the portion of the list that remains in the iteration. This is not the case when we iterate over a binary tree: the sequence of visited elements and the tree `t` are only a prefix of the entire sequence of the elements in `t0`. The following first two pre-conditions account for this behavior:

WhyML
```
    variant  { t }
    requires { length t0 >= length (v ++ t) }
    requires { v ++ t = t0[.. length vres] }
    requires { permitted t0 v    /\ inv acc  v }
    ensures  { permitted t0 vres /\ inv accu vres /\ vres = v ++ t }
  = match t with
    | Empty      -> (v, acc)
    | Node l x r -> let v1, accl = fold v acc l in
        fold (snoc v1 x) (f v1 accl x) r
```

```
    end

end
```

Using these extra preconditions, SMT solvers are able to prove every verification condition related to the `permitted` property. The remaining of the verification conditions generated for the `fold` function are also automatically discharged. This completes the definition of module `FoldTree`.

Fold client. The client program of `fold` uses this iterator to compute the number of elements of a given binary tree. This is very similar to function `list_length` from the previous section. We devise the following WhyML module:

WhyML

```
module TreeLength

  use seq.Seq, bintree.Tree, int.Int, Elements

  type elt
  type acc = int

  predicate inv (ghost v: seq elt) (acc: acc) =
    acc = length v

  let f (ghost v: seq elt) (acc: acc) (x: elt)
  = acc + 1

  clone FoldTree with
    type elt, type acc, predicate inv, val f

  let tree_length () : (ghost v: seq elt, accu: acc)
    ensures { accu = length t0 }
  = fold empty 0 t0

end
```

The regular use (as a coercion) of logical function `elements` to reason about the elements of a binary tree allows us to specify `tree_length` exactly as we did for the `list_length` function. This is, indeed, a very interesting aspect about our modular WhyML development of interfaces and clients of the `fold` iterators. As expected, all the generated verification conditions for the above module are discharged in no time.

6.3.3.2 Horner Method

The last example we present is that of a program computing the value of a polynomial. We represent the polynomial $c_0 + c_1x + \ldots + c_nx^n$ by the list of $[c_0, c_1, \ldots, c_n]$ of its coefficients, which we assume to be integer coefficients.

Fold implementation. The `fold_right` function is the iterator used to traverse a list from right to left. This is implemented in WhyML as follows:

WhyML

```
let rec fold_right (f: acc -> elt -> acc) (l: list elt) (a: acc) : acc
= match l with
  | Nil      -> acc
```

```
  | Cons x r -> f (fold_right f r acc) x
  end
```

It works by calling itself recursively over the tail of a list, and only then by applying the consumer function to the head of the list and the value computed recursively.

In order to formally specify `fold_right`, we need to give appropriate definitions to `permitted` and `complete` properties. We first introduce the following logical function to convert a list into a sequence:

```
function elements_left (l: list 'a) : seq 'a = match l with                WhyML
  | Nil -> empty
  | Cons x r -> cons x (elements_left r)
end
```

We then derive a function `elements` to reverse the sequence computed by `elements_left`, as follows:

```
function elements (l: list 'a) : seq 'a =                                  WhyML
  reverse (elements_left l)
```

Function `reverse` comes from the Why3 standard library. We can finally give the definitions of `permitted` and `complete` using the reversed sequence of a list, as follows:

```
predicate permitted (v: seq elt) (l: list elt) =                           WhyML
  length v <= length (elements l) /\
  forall i. 0 <= i < length v -> v[i] = (elements l)[i]

predicate complete (v: seq elt) (l: list elt) =
  length v = length (elements l)
```

Predicate `permitted` states that the `visited` sequence forms a reversed suffix of list `l`.

The implementation of `fold_right` is given in Fig. 6.2. The consumer function `f`, declared in lines 5-9, presents exactly the same contract as in the previous `fold` examples. Function `fold_right` is given three arguments: the sequence `v` of visited elements, the value `acc` which corresponds to the current value of the accumulator, and finally list `l`, the list being iterated over. It returns the ghost sequence `vr`, the final sequence of visited elements, and the final computed value for the accumulator, which we name `accu`. The contract given in lines 15-20 expresses the total correctness of the `fold_right` implementation. The first pre-condition states that, at each call of `fold_right`, the sequence `v` remains empty. Contrarily to the `fold_left` implementation, the sequence of visited elements remains empty until the end of all recursive calls. It is only when the `fold_right` returns that we feed function `f` with elements from the list, and consequently appending those to the sequence `v`. The pre-condition in line 16 and the postcondition in line 19 express that the sequence of recursive calls preserve the `inv` and `permitted` predicates. The pre-condition in line 17 and the postcondition in line 20 stand for some auxiliary specification, specific to `fold_right` traversal. This pre-condition states that at entry of each recursive call, `l` forms a prefix of the whole `l0` list; the postcondition ensures that a recursive call enumerates the elements of `l`, and only those. All the verification conditions generated from the given implementation and specification are automatically proved.

Fold client. We can evaluate a polynomial using the Horner form $c_0 + x(c_1 + x(\ldots + xc_n)\ldots)$, which is easily implemented as a `fold_right` client program:

```
let val_of_pol (p: list int) (x: int) : int                                OCaml
= fold_right (fun acc c -> c + acc * x) 0 p
```

```
type elt                                                                    WhyML
type acc
predicate inv (seq elt) acc

val constant l0 : list elt

val f (ghost v: seq elt) (acc: acc) (x: elt) : acc
  requires { inv v acc }
  requires { permitted (snoc v x) l0 }
  requires { not (complete v l0) }
  ensures  { inv (snoc v x) result }

let rec fold_right
  (ghost v: seq elt) (acc: acc) (l: list elt) : (ghost vr: seq elt, accu: acc)
  requires { v = empty }
  requires { inv v acc /\ permitted v l0 }
  requires { prefix (elements l) (elements l0) }
  variant  { l }
  ensures  { inv vr accu /\ permitted vr l0 }
  ensures  { vr = elements l }
= match l with
  | Nil -> (empty, acc)
  | Cons x r ->
     let (v_r, acc_r) = fold acc r in
     (snoc v_r x, f v_r acc_r x)
  end
```

Figure 6.2: `Fold_right` Implementation and Specification.

This program runs in $O(n)$ time, performing exactly n multiplications and n additions. A suitable contract for the `val_of_pol` function is the postcondition

```
ensures { result = eval p x }                                               WhyML
```

where `eval p x` returns the same result as

$$\sum_{0 \le i < n} p_i x^i \tag{6.1}$$

where p_i represents the i-th element of polynomial p.

In order to devise a proper definition to `eval`, let us introduce some auxiliary functions. We begin by defining the result of the `i`-th value of a sequence `v` to the `i`-th power of a value `x`, as the following logical function:

```
function mult_power (v: seq int) (x: int) (i: int) : int =                  WhyML
  v[i] * power x i
```

Next, we get closer to the definition given in equation (6.1), as we use the `sum` function from the Why3 standard library to define the valuation of a sequence `v` of integers, given the value of `x`:

```
function eval_sequence (v: seq int) (x: int) : int =                        WhyML
  sum (mult_power v x) 0 (length v)
```

We can now easily deduce the definition of `eval`:

```
function eval (p: list int) (x: int) : int =                    WhyML
  eval_sequence (elements_left p) x
```

This is as simple as turning the polynomial `p` into a sequence, using the previously defined function `elements_left`, and then traverse it using the `eval_sequence` function.

We implement and specify the consumer function to be applied to the `fold_right` iterator as follows:

```
val constant x : int                                            WhyML

let fun_f (ghost v: seq elt) (acc: acc) (c: elt) : int
  ensures  { result = c + acc * x }
= c + acc * x
```

Following the Horner method, the consumer function takes the next element in the enumeration, `c`, and adds it to the product of the accumulator value, `acc`, by `x`, the value assigned to the polynomial variable. Next, we provide the following suitable client invariant to the traversal:

```
predicate inv (v: seq elt) (acc: acc) =                         WhyML
  acc = eval_sequence (reverse v) x
```

We can finally instantiate the non-defined elements of the `FoldRight` module:

```
clone FoldRight as FR with                                      WhyML
  type elt, type acc, val l0, predicate inv, val fun_f
```

The above **`clone`** expression generates verification conditions that state the specification inclusion of the implemented `fun_f` function with respect to the `fun_f` function declared in the `FoldRight` module. The proof of specification inclusion amounts to proof that the postcondition of the implementation of `fun_f` implies the postcondition given to the `fun_f` val in the `FoldRight` module, *i.e.*, the following property holds:

```
acc = eval_sequence (reverse (snoc v c)) (c + acc * x)          WhyML
```

To prove such a statement, we need two auxiliary lemmas. The first one is a simple property about the composition of `reverse`, `cons`, and `snoc`:

```
lemma snoc_reverse_cons : forall v: seq 'a, c: 'a.              WhyML
  reverse (snoc v c) = cons c (reverse v)
```

The second states exactly the implication we wish to prove:

```
let lemma eval_sequence_cons (v: seq int) (c x: int)            WhyML
  ensures { eval_sequence (cons c v) x = c + eval_sequence v x * x }
```

The proof of lemma `eval_sequence_cons` proceeds as follows:

```
= assert { eval_sequence (cons c v) x                           WhyML
         = (* definition of eval_sequence *)
           sum 0 (length (cons c v)) (mult_power (cons c v) x)
         = (* lemma sum_left from Why3 standard library *)
           c + sum 1 (length (cons c v)) (mult_power (cons c v) x) };
  (* instantiating lemma sum_shift *)
  sum_shift (mult_power (cons c v) x) (multf (mult_power v x) x)
            1 (length (cons c v)) 0 (length v);
  assert { c + sum 1 (length (cons c v)) (mult_power (cons c v) x)
         = (* lemma sum_shift from Why3 standard library *)
```

iterator	loc	los	time (sec)
list fold_right	11	38	0.10
list fold_left	9	32	0.17
array fold_left	23	45	0.02
set fold	9	26	1.67
tree inorder_fold	12	30	0.56
interval fold	14	39	0.27
total	78	210	

(a) Fold Implementation

program	loc	los	time (sec)
list length	12	4	0.01
set of list	29	17	0.18
value of polynomial	15	38	0.49
tree size	4	12	0.02
set cardinal	4	12	0.02
set copy	4	12	0.02
sigma	4	9	1.11
total	109	147	

(b) Fold Clients

Table 6.2: Experimental Results (fold iterators).

```
  c + sum 0 (length v) (multf (mult_power v x) x)
= (* lemma sum_mult_constant from Why3 standard library *)
  c + (sum 0 (length v) (mult_power v x)) * x
= (* definition of eval_sequence *)
  c + eval_sequence v x * x }
```

where `multf` is the following logical function:

```
function multf (f: int -> int) (x: int) : int -> int =                    WhyML
  fun i -> f i * x
```

All the generated verification conditions for `eval_sequence_cons` are automatically discharged. This completes the proof of refinement of the `FoldRight` module.

We complete the presentation of this case study with the proof of functional correctness of the `val_of_pol` function. After having refined the `FoldRight` module, we devise the following WhyML implementation:

```
let val_of_pol () : (vres: seq elt, accu: acc)                            WhyML
  ensures { accu = eval l0 x }
= FR.fold_right empty 0 l0
```

This uses the verified implementation of the `fold_right` iterator, showed previously in this section. The proof of `val_of_pol` uses only the definitions of `permitted`, `complete`, and the `inv` predicate. The implementation of the consumer function is *hidden* by the refinement and is never used in the proof of `val_of_pol`. All the generated verification conditions for this program are easily proved by SMT solvers.

6.3.4 Other Case Studies

Using our proof methodology, we have implemented and verified several other instances of the `fold` function. These include a `fold_left` over lists, a `fold_left` over arrays, a `fold` over sets, an `inorder_fold` over binary tree, and finally a `fold` that enumerates the integers in a given range. Table 6.2a shows the lines of code, the lines of specification, and the verification time (in seconds) for each `fold` implementation. All of these examples are proved automatically.

From a client code perspective, we have used our approach to verify several examples. These include computing the length of a list (both using `fold_left` and `fold_right`), converting a list into a set, computing the cardinal of a set, making a copy of a set, and computing the sum of all integers within a given interval. Table 6.2b shows the lines of code, the lines of specification, and

the total verification time (in seconds) for each program. The whole set of verification conditions generated for these examples are proved fully automatically.

6.4 Discussion and Related Work

Tool-independent approach. As we have mentioned throughout this chapter, our approach to specify iteration based on predicates `permitted` and `complete` is not tied to any particular verification technology. We conducted all of our experiments within the Why3 framework, but we argue that it would not be a major task to translate these to other verification tools, *e.g.*, Dafny or Viper [114].

It is even more interesting to observe that our approach also combines well with the use of interactive proof assistants and very rich specification languages. The proof, by François Pottier of the OCaml hash table implementation [131] supports this claim. In this work, the OCaml code is translated to Coq by the CFML tool and is verified using higher-order separation logic. The iteration operations (`fold` and cascades) over hash tables are specified in terms of the sequence of elements the consumer may observe. Such a sequence is characterized using predicates `permitted` and `complete`. In fact, the name used to refer to the two predicates are inspired by this work (originally, we were using `enumerated` and `completed`). Contrarily to our experiments in Why3, the setting of a CFML and Coq propose a very rich specification logic and type system, which enables reasoning about imperative higher-order functions.

Catherine Dubois and Alain Giorgetti use our specification in terms of `permitted`/`complete` predicates to provide WhyML implementations of generators. Then, a correct-by-construction implementation of these generators is extracted to OCaml code, which can later be used to test Coq developments, via the QuickChick plugin.

Specifying and proving cursors. The idea of formally specifying and proving cursors is not new. Weide presents a formal specification for the cursors' behavior [144] using the *RESOLVE* language [91]. A collection is modeled as a finite set (in the mathematical sense) and a cursor is specified using a `past` sequence corresponding to our `visited` and another `future` sequence corresponding to remaining elements. A third sequence, `original`, contains the set of elements of the collection. Under such formalization, a cursor can only be used with finite collections and the traversal is necessarily deterministic. The author also presents a mechanism to ensure coherence, by means of extra operations over cursors, `Start_Iterator` and `Finish_Iterator`, that should limit all the cursor uses. In this way, and contrary to our approach, the validity of a cursor can only be verified once the traversal is finished.

Catherine Dubois and Renaud Rioboo provide a FoCaLiZe implementation of functional iterators, addressing the Specification and Verification of Component Based Systems 2006 challenge [57].

Jacobs *et al.* [80] describe a methodology to specify and verify both implementations and clients of the Iterator pattern, in the context of the Spec# programming system. In their formalism, methods are separated into regular methods and *enumerator methods*, so the latter can present an `invariant` clause in their contract. In the contract of such methods it is possible to refer to the keyword `values`, which represents the set of elements enumerated so far. This resembles our use of the sequence of visited elements. The client programs are *for-each* loops whose loop invariants can also refer to `values`. These programs are then translated into regular `for` loops to be verified. This methodology is only applied to first-order programs and does not support enumeration with side-effects, or the creation of objects performing iteration.

Cursors in future Why3. In the short term, it would be interesting to explore an extension of Why3 with a `for` loop *à la* Java, based on cursors. For instance, we could imagine a client code of the form

```
let s = ref 0 in                                                    WhyML
for x in a with c do
  variant   { length c.collection - length c.visited }
  invariant { !s = sum (fun i -> a[i]) 0 (length c.visited) }
  s += x
done;
!s
```

As it happens in Java, this would be nothing more than syntactic sugar for a `while` loop invoking `next` at each time a new element is needed. We would have the following WhyML program:

```
let s = ref 0 in                                                    WhyML
let c = create s in
while has_next c do
  variant   { length c.collection - length c.visited }
  invariant { !s = sum (fun i -> a[i]) 0 (length c.visited) }
  let x = next c in
  s += x
done;
!s
```

which is exactly the client program we can find in Sec. 6.2.3. An important difference between directly writing the client code with a `while` loop and using the (imaginary) `for` loop is that we can optimize the set of verification conditions that are generated. In fact, since the loop is guarded by the result of `has_next`, its postcondition `result <-> not (complete c.collection c.visited)` trivially implies the pre-condition of function `next`.

Persistent Cursors. We adopt a model where cursors are treated as mutable data structures, with function `next` updating the cursor via a side effect. This is not mandatory. A cursor can be implemented as a persistent structure [58]. In such a case, function `next` no longer updates the internal state of the cursor but, instead, *returns* the next element in the enumeration together with a *new* cursor. From a point of view of WhyML verified implementations, reasoning with persistent cursors would not imply a drastic reformulation of our development. We would completely remove the mutable character out of our `cursor` type

```
type cursor = private {                                             WhyML
  ghost visited    : seq elt;
  ghost collection : t;
} invariant { permitted collection visited }
```

and would change the return type and specification of the `next` function, accordingly:

```
val next (c: cursor) : option (elt, cursor)                         WhyML
  ensures  { match result with
             | None         -> complete c.collection c.visited
             | Some (e, cn) -> cn.visited    = snoc c.visited e /\
                               cn.collection = c.collection end }
```

We no longer need to use a `has_next` function, since we know that the iteration is `complete` as soon as `next` returns `None`. This would suit particularly well client code that is implemented as a

recursive function, *e.g.*,

```
let rec traversal (c: cursor) (acc: int) =                    WhyML
  match next c with
  | None         -> acc
  | Some (e, cn) -> traversal cn (acc + e)
  end
```

Specification and proof of higher-order iterators. Many tools exist that tackle the verification of higher-order effectful programs, in particular of higher-order iterators. These are normally based on rich specification logics and type systems. Liquid Types [137] is a type system with refinement types extracted from a decidable logic. This type system is used to infer simple "loop invariants" from a given code. In our case, the user supplies the loop invariant and, contrary to the Liquid Types approach, we apply and prove an iterator client without access to the iterator implementation, in a modular way.

Yann Régis-Gianas and François Pottier [133] propose a Hoare Logic for call-by-value pure programs. One cannot use their framework to reason about side effects, but the user can, nonetheless, write effectful programs, for instance divergent ones. The authors detail on a verified implementation of sets as binary trees. This includes a `fold`-like iterator, as well as persistent iterators to traverse the elements of the data structure. They use both paradigms to implement and verify client programs that compute the number of elements in a set. The given specification can be easily adapted to the framework of `permitted`/`complete`, as we show in Sec. 6.3.3.1.

Dependent types and monad structures are used in the F* tool [139] as the theoretical basis to tackle the proof of higher-order programs with effects. F* can be used both as a programming language and as a proof assistant, featuring a higher-order specification and programming language. This tool has been used to verify many complex effectful programs including cryptographic protocols and the mechanization of lambda calculi metatheory. Even though F* is able to use SMT solvers during the proving process, it seems that the verification of nontrivial (effectful) higher-order programs is out of the realm of automatic provers. In particular, the specification of a higher-order iterator is very similar to what one would write in a general-purpose proof assistant like Coq.

The CFML tool [29] uses characteristic formulas to verify OCaml code within the Coq proof assistant. A characteristic formula is a higher-order formula that is generated from a source code and that describes its semantics. Using a proof assistant based on higher-order logic, the characteristic formula can be exploited to prove complex properties about that program. Up to now, CFML has been used to verify several nontrivial higher-order imperative programs, including higher-order iterators over mutable data structures. However, the specification used to describe a higher-order iterator is always tied to a specific collection data type.

Higher-order iterators in Why3. We propose an approach to the verification of higher-order iterators in Why3 based exclusively on first-order specification and programming, and on the use of the Why3 refinement mechanism to link client programs with iterator providers. Even if this approach works well in practice, this might be a bit puzzling for a common OCaml programmer, who is used to higher-order iterators. Additionally, our approach demands some knowledge on WhyML modules and associated **`clone`** operations. Considering the client program of Sec. 6.3.3.2, it would be much more natural to write a WhyML program as follows:

```
let val_of_pol (p: list int) (x: int) : int                   WhyML
  ensures { result = eval p x }
= fold_right (fun (ghost v) acc c ->
```

```
                invariant { acc = eval_sequence (reverse v) x }
                c + acc * x) p 0
```

This is close to what an OCaml programmer would expect, with only a little overhead of specification elements. The user-provided invariant characterizes the whole iteration performed by the `fold_right` function. We have began the development of a prototype tool that reads both implementations and clients (as the above one) of higher-order iterators, together with a specification, and turns them into first-order Why3 programs to be verified. For the case of `fold`-like implementations, our prototype tool accepts a specification of the form

```
let rec fold_right (ghost inv: seq elt -> acc -> bool) (f: acc -> elt -> acc)
                   (l: list elt) (acc: acc) : acc
  foldspec { (permitted, complete) }
  ...
```
WhyML

which is then translated into the WhyML program of Fig. 6.2.

When it comes to effectful higher-order iterators, we can only present a partial answer to this problem. We are able to specify the sequence of elements produced by such an iterator, via the `permitted` and `complete` predicates, which we can use to interface the client with the iterator provider. However, we cannot establish strong guarantees about the implementation of the iterator itself. For instance, nothing prevents us from applying the consumer function twice to the third element in the iteration, thrice to the 42nd element and not at all to the last one. This is problematic in the context of a consumer function with side effects. Currently, our prototype tools translates an effectful iterator into a first-order program that uses a cursor to perform the same traversal. Using this approach, we have verified a program that removes duplicate elements from a list. The consumer function updates a hash table with the elements encountered during iteration, which is done via a side effect.

Integration with the VOCaL project. The fact that we do not dispose of a satisfying treatment of stateful higher-order iterators is a major pitfall of our approach and makes it hardly applicable, at least in the immediate, to the VOCaL project. As explained in Sec. 5.3.2, we use the `equivalent` clause of OSL to introduce an operationally equivalent program to a higher-order function. We could imagine, for instance, to add to OSL a `foldspec` clause, like we propose for our prototype tool, and to replace all `equivalent` statements. We would give a precise semantic meaning to the `foldspec` clause: the consumer function is applied once and only once to each element in the iteration, while the sequence of visited elements should respect the `permitted`/`complete` pair of predicates. This approach could be used in the proof of either iterators and client programs. Nonetheless, it remains unclear if the use of `foldspec` could be smoothly integrated in our toolchain for verified OCaml programs.

Un peu de programmation éloigne de la logique mathématique ; beaucoup de programmation y ramène.

Xavier Leroy

7 Conclusion

As we arrive to the *terminus* of this work, it is now time to draw some conclusions. If we get back to the title of this document, it is now easy to understand the careful choice of words and why we present them in this order. In this thesis, we extended the Why3 ecosystem with a variety of tools, *e.g.*, the new extraction mechanism and the OSL translation plug-in, and developed techniques, *e.g.*, a general specification of iteration and the mini-heaps approach, that allow us to conduct the proof of modular stateful programs within Why3. Examples range from priority queues implemented as Pairing Heaps, where modularity scales up to the level of OCaml functors, to arbitrary pointer-based data structures, such as our verified union-find implementation. On a more fundamental side, we formalized a representative part of the Why3 extraction mechanism on top of the KidML language, a sub-set of WhyML.

7.1 Contributions

Why3 extraction mechanism. The new Why3 extraction mechanism is the core of our work. We have re-implemented from scratch this part of the framework into a robust, modular (scaling up to OCaml functors) extraction machinery. We have been able to successfully use extraction on several verified WhyML programs, in order to produce correct-by-construction executable code. We have mainly used extraction to produce verified OCaml programs. This new extraction mechanism has also been used with success within a verified version of the GMP library [136], and also to produce correct-by-construction CakeML programs. Another interesting use of extraction is described by Gillaume Melquiond and Raphaël Rieu-Helft [109]. In this work, the idea of proof by reflection inside Why3 is pushed forward and is applied to the proof of challenging arithmetic algorithms. In order use the whole expressive power of the WhyML language, the authors add an interpreter to Why3 to recover the result of a decision procedure written as an arbitrary WhyML program. This interpreters operates on the intermediate representation resulting from our extraction.

Our formalization of the extraction mechanism is done in a *pen-and-paper* style. Since the rise of the POPLmark challenge [7], it has become customary to develop such forms of meta-theory within a proof assistant. We know that, in principle, it is possible to develop the formalization of the KidML language and the extraction algorithm, for instance using the Coq proof assistant. In this thesis, we focused on the careful construction of a working extraction mechanism, that is now fully available to the Why3 users. Nonetheless, it is definitely an exciting line of future work

to seek a machine version of our meta-theory, as demonstrated by the CakeML [92] and Œuf [113] projects.

Mini-heaps. OCaml It is our purpose to keep extending Why3 with means to prove a larger set of OCaml programs. Given the strong constraints of the Why3 type system, some pointer-based data structures cannot be directly encoded in the WhyML language. The solution is to introduce an explicit memory model, *i.e.*, to declare a type for pointers and memory, together with a set of functions to read and write memory. This resembles very much to the approach followed by the Jessie plug-in [106] of the Frama-C platform.

Our approach to deal with memory models is slightly different from the one that is normally employed by tools that use Why3 as an intermediate verification language. We build memory models targeting only the part of the code that cannot be expressed in WhyML, instead of building a huge memory model to deal with an existing programming language in its entirety. Additionally, we do not declare memory as a global mutable value. Instead, we pass it as a ghost argument to each function in our memory model, which, by the properties of the WhyML type system, directly guarantees some degree of separation between different fragments of the memory. We refer to this approach as the *mini-heaps*.

Using the described approach, we have been able to verify in Why3 non-trivial pointer-based implementation of data structures. The union-find data structure and the mergesort algorithm over singly-linked lists, both presented in Chap. 5, are two examples of successful application of the mini-heaps approach in the verification of programs manipulating memory in an arbitrary fashion. When it comes to extraction, it only requires a custom driver of very few lines, mapping the memory model operations to OCaml counterparts. Such a mapping is normally straightforward, which makes it very easy to convince ourselves that the driver does not introduce any bug in the extracted code. This approach has reveled to work very well in practice, making it rather pleasant to conduct the proof of complex data-structures within Why3.

Modular specification of iteration. Since iteration is a pervasive concept in programming, we addressed the challenge of specifying and verifying iteration providers, independently of the underlying implementation. This led us to a modular and generic approach to the specification of iteration processes, presented in Chap. 6. The idea is to use a pair of predicates `permitted` and `complete` to characterize the finite sequence of elements already enumerated at a given moment of the iteration. Using such an approach, we are able to deal with both finite and infinite iterations, deterministic and non-deterministic, and also with the case when mutable data is changed during iteration. Our approach is not limited to the traversal of data structures; it can be used to specify the result of an iterative algorithm, for instance.

An important aspect about our approach is that it can be employed by different verification tools. We have already mentioned the work of François Pottier in the proof of the OCaml hash table implementation in the CFML tool [131], where the (higher-order) iterators on such structure are specified using the `permitted`/`complete` approach. Given the successful use of our approach within CFML, a tool with a very different verification philosophy when compared with Why3, we are confident that our specification proposal can be easily integrated in verification platforms closely related to Why3, *e.g.*, Dafny or Viper.

We applied our specification to the paradigms of cursors and higher-order fold-like functions. We proved several providers of iteration, as well as client applications. An extremely important aspect about these proofs is that we always impose an abstraction barrier with the purpose of hiding implementation details to the client code. This makes echo to the *modular* word in the title of the this thesis.

Verification of an OCaml library. The ultimate contribution of this work is the successful use of Why3 in the development of VOCaL, the *Verified OCaml Library*. Using the methodology we propose in Chap. 5, we were able to build verified implementations of several data structures and algorithms in the WhyML language, which are then automatically translated into compilable, correct-by-construction OCaml modules. Other than its functional correctness, we also prove that the obtained OCaml code refines to the specification we provide in the associated interface file. All of these examples are ready to be distributed and used by OCaml programmers.

From a personal point of view this is, perhaps, the most rewarding outcome of this thesis. The excitement about the possibility of seeing our work being using by a potentially large base of OCaml programmers is a most delightful way to conclude these years of research. As a passionate programmer, it is my deepest honor to be able to contribute to the OCaml community and ecosystem, as I have enjoyed some many good moments using this language and the associated tools.

7.2 Discussion and Perspectives

With the conclusion of this work, we anticipate some future work that could be of great interest. We describe those here, undertaking a much more *research agenda* approach, rather than a direct continuation of this thesis.

Modules refinement. Within our toolchain for verified OCaml programs, we make extensive use of the WhyML module system and of the Why3 refinement mechanism to prove that the obtained OCaml code conforms to the specified interface. Nonetheless, the use of the `clone` command every time we wish to refine the components of a module, together with an explicit substitution for each component in the module, makes proofs of refinement tedious routines. It would be of utmost practical interest to extend Why3 with a simpler module refinement mechanism. For instance, instead of providing a `clone` expression the user would write a program of the form

WhyML
```
module Spec
  ...
end

module Impl: Spec
  ...
end
```

where module `Impl` refines module `Spec`. The advantages of such an approach are two-fold: first, the WhyML development would result in a much more compact code; second, this would prevent situations in which the user forgets about including some symbol of `Spec` in the substitution given to `clone`, since all the symbols of the refined module would be automatically mapped to symbols of `Impl`, as long as these have the same name.

Support for higher-order stateful programs. When programming in the OCaml language, we are very frequently exposed to the combination of higher-order functions with side effects. On the other hand, the use of higher-order in WhyML is restricted to stateless functions, which fatally limits the set of OCaml programs that we might expect to prove within Why3. We believe it is worth to strike for a better support of higher-order computations with effects within the realm of automated verification tools. A major challenge in extending such tools with higher-order traits is to keep the specification logic and generated verification conditions simple enough, as to keep using SMT solvers at the backstage. In this context, we started an exploratory study on

the use of the defunctionalization technique as a first step in the proof of higher-order programs with effects [124]. The idea is to translate annotated higher-order programs into a first-order counterpart, which can then be fed to off-the-shelf verification tools. Using such a technique, we were able to verify some stateful higher-order programs in Why3. It remains, however, very difficult to apply such a technique in a modular setting, since defunctionalization is known to be a whole-program transformation, and it is not clear if we could smoothly integrate this approach within Why3 and scale up to the proof of significant size higher-order programs.

Growing strong and healthy. As mentioned throughout this thesis, we have applied our methodology to the proof of several OCaml modules, which are integrated into the VOCaL library. This shows that Why3 is a now a mature tool, and that it is possible to use our code extraction *modus operandi* in a larger scale programming project. To contribute to a verified OCaml library is certainly a first good sign that the proof of realistic software is possible within Why3.

However, the answer to the question "can Why3 scale up to the proof industrial-sized programming projects" remains unclear. In particular, is the approach of extracting correct-by-construction code practical enough to be included in a large software development? In the literature, we can find some successful cases of very large verified programs, that are later extracted into executable code. The most shining example in this category is CompCert [99], the verified C compiler. Such a compiler is specified and proved correct using the Coq proof assistant. The executable OCaml sources of this development are obtained via the Coq extraction mechanism. As tempting as it might be, it remains to demonstrate if, in practice, Why3 could be used to verify and generate a correct-by-construction implementation of such a large piece of software.

A discussion that often arises is the duality of code extraction *versus* verification of existing code. Let us take the example of Mirage OS [104], an operating system written in the OCaml language. If we decide tomorrow to engage in a project to prove the code of Mirage, what would the best approach be? Certainly, we do not want to duplicate the existing code base, just to provide a verified WhyML implementation of the OS modules and then extract an OCaml implementation that would mostly resemble to the already existing code. Additionally, WhyML lacks some important features, such as treatment of Input/Output operations, required for the proof of an OS source code. A much realistic approach would perhaps be to use a dedicated tool to the verification of existing OCaml code, while using Why3 to verify and extract correct-by-construction implementations of new modules, which would then be included inside Mirage. Exploring how extracted code can be smoothly linked with hand-written code is certainly an interesting research line. CompCert is a successful example of this approach, since some modules of the compiler are still hand-written.

Another interesting question that is worth to be investigated is the collaborative use of deductive verification with other forms of formal methods. Let us get back to the example of Mirage OS. We could imagine using Why3 to verify and extract critical parts of the system, while other less critical modules would be analyzed by more light-weighted formal methods. These could include, for instance, model checking [9], information flow inference [132], or some form of abstract interpretation [26].

Finally, there has been a recent growing trend around the integration of formal methods in continuous reasoning of industrial software code repositories. Peter O'Hearn [117] describes how formal methods are being used inside big software companies, such as Facebook and Amazon Web Services. The idea is to use verification tools to (almost) immediately inform development teams that some commit breaks the expected behavior of the system. One key to the success of applying formal methods in such industrial contexts is to decide when verification tools must be called to action. After each commit? Before a major release? To be integrated in a pipeline of such industrial size would certainly be a *tour de force* for Why3. One could very well imagine that, in a first moment, someone proves a property using Why3 and a combination of SMT solvers. Then,

after each commit to the code repository, we would invoke the `why3replay` command to replay the proof in the same conditions, in order to check if the introduce updates in the code do not break the established properties. In case of failure, a report could be immediately sent to the author of the commit, whom would either repair the proof or alter the commited files.

A

Permutations Library

WhyML

```
module Permut

  use import int.Int
  use import seq.Seq
  use import seq.Distinct
  use import seq.Exchange
  use import mach.int.Int63
  use import mach.array.ArrayInt63
  use import mach.int.Refint63

  type t = { a: array63; ghost inv: array63; ghost size: int }
    invariant { size = length a = length inv }
    invariant { forall i. 0 <= i < size -> 0 <= a[i]   < size }
    invariant { forall i. 0 <= i < size -> 0 <= inv[i] < size }
    invariant { forall i. 0 <= i < size -> a[inv[i]] = i }
    invariant { forall i. 0 <= i < size -> inv[a[i]] = i }
    by { size = 1; a = make 1 0; inv = make 1 0 }

  let id (n: int63) : t
    requires { 0 <= n }
    ensures  { result.size = n }
    ensures  { forall i. 0 <= i < n -> result.a[i] = i }
  = let a = make n 0 in
    let ghost inv = make n 0 in
    for i = 0 to n - 1 do
      invariant { forall j. 0 <= j < i -> a[j] = inv[j] = j }
      a[i] <- i;
      inv[i] <- i
    done;
    { size = n; a = a; inv = inv }
```

```
let transposition (i j n: int63) : t
  requires { 0 <= i < n /\ 0 <= j < n }
  ensures  { result.size = n }
  ensures  { result.a[i] = j /\ result.a[j] = i }
  ensures  { forall k. 0 <= k < n -> k <> i -> k <> j -> result.a[k] = k }
= let t = id n in
  swap t.a i j;
  swap t.inv i j;
  t

let compose (p q: t) : t
  requires { p.size = q.size }
  ensures  { result.size = p.size }
  ensures  { forall i. 0 <= i < p.size -> result.a[i] = p.a[q.a[i]] }
= let n = p.size in
  let res = make n 0 in
  let ghost ires = make n 0 in
  for i = 0 to n - 1 do
     invariant { forall j. 0 <= j < i -> res[j]  = p.a  [q.a  [j]] }
     invariant { forall j. 0 <= j < i -> ires[j] = q.inv[p.inv[j]] }
     res[i]  <- p.a  [q.a  [i]];
     ires[i] <- q.inv[p.inv[i]]
  done;
  { size = n; a = res; inv = ires }

let inverse (t: t) : t
  ensures { result.size = t.size }
  ensures { forall i. 0 <= i < t.size -> result.a[i] = t.inv[i] }
= let n = t.size in
  let res = make n 0 in
  for i = 0 to n - 1 do
    invariant { forall j. 0 <= j < i -> res[t.a[j]] = j }
    res[t.a[i]] <- i
  done;
  { size = n; a = res; inv = copy t.a }

let rec lemma pigeonhole (n m: int) (f: int -> int)
  requires { 0 <= m < n }
  requires { forall i. 0 <= i < n -> 0 <= f i < m }
  ensures  { exists i1, i2. 0 <= i1 < i2 < n /\ f i1 = f i2 }
  variant  { m }
= for i = 0 to n - 1 do
    invariant { forall k. 0 <= k < i -> f k < m - 1 }
    if f i = m - 1 then begin
      for j = i + 1 to n - 1 do
        invariant { forall k. i < k < j -> f k < m - 1 }
        if f j = m - 1 then return
      done;
      let function g k = if k < i then f k else f (k + 1) in
```

```
        pigeonhole (n - 1) (m - 1) g;
        return end
    done;
    pigeonhole n (m - 1) f

  let cycle_length (t: t) (i: int63) : (n: int63, ghost s: seq int)
    requires { 0 <= i < t.size }
    ensures  { 0 < length s = n }
    ensures  { forall j. 0 < j < n -> s[j] = t.a[s[j - 1]] }
    ensures  { s[0] = i /\ t.a[s[n - 1]] = i }
  = let n = ref 1 in
    let ghost s = ref (singleton (to_int i)) in
    let x = ref t.a[i] in
    while !x <> i do
      variant   { t.size - length !s }
      invariant { 0 <= !x < t.size }
      invariant { 0 < length !s = !n }
      invariant { !s[0] = i }
      invariant { forall j. 0 < j < !n -> !s[j] = t.a[!s[j - 1]] }
      invariant { !x = t.a[!s[!n - 1]] }
      invariant { forall j. 0 <= j < !n -> 0 <= !s[j] < t.size }
      invariant { distinct !s }
      s := snoc !s (to_int !x);
      x := t.a[!x];
      if to_int !n + 1 > t.size then pigeonhole (Seq.length !s) t.size (get !s);
      incr n;
    done;
    !n, !s

end
```

Bibliography

[1] Jean-Raymond Abrial. *The B-Book, assigning programs to meaning.* Cambridge University Press, 1996.

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. A fast and verified software stack for secure function evaluation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1989–2006, New York, NY, USA, 2017. ACM.

[4] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. *Rigorous Software Development: An Introduction to Program Verification.* Springer Publishing Company, Incorporated, 1st edition, 2011.

[5] Davide Ancona and Elena Zucca. An algebraic approach to mixins and modularity. In M. Hanus and M. Rodrìguez Artalejo, editors, *5th Intl. Conf. on Algebraic and Logic Programming*, number 1139 in Lecture Notes in Computer Science, pages 179–193, Berlin, 1996. Springer.

[6] Ali Assaf, Guillaume Burel, Raphal Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Expressing theories in the λΠ-calculus modulo theory and in the Dedukti system. In *22nd International Conference on Types for Proofs and Programs, TYPES 2016*, Novi SAd, Serbia, May 2016.

[7] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, number 3603 in Lecture Notes in Computer Science, pages 50–65. Springer, 2005.

[8] Ralph-Johan Back and Joachim Von Wright. *Refinement Calculus: A Systematic Introduction.* Springer, 1998. Graduate Texts in Computer Science.

[9] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[10] Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2005.

[11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

[13] Gilles Barthe, Juan Manuel Crespo, and César Kunz. *Relational Verification Using Product Programs*, pages 200–214. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[14] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[15] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.cea.fr/acsl.html`.

[16] Bernhard Beckert and Michał Moskal. Deductive verification of system software in the Verisoft XT project. *Künstliche Intelligenz*, 24(1):57–61, 2010.

[17] Bernhard Beckert, Jonas Schiffl, Peter H. Schmitt, and Mattias Ulbrich. Proving jdk's dual pivot quicksort correct. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2017)*, 2017. To appear.

[18] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 14–25, New York, NY, USA, 2004. ACM.

[19] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[20] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.

[21] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie*, pages 53–64, August 2011. `https://hal.inria.fr/hal-00790310`.

[22] Robert S. Boyer and J. Strother Moore. Mjrty: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.

[23] Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. Ready, set, verify! applying hs-to-coq to real-world haskell code (experience report). *Proc. ACM Program. Lang.*, 2(ICFP):89:1–89:16, July 2018.

[24] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[25] Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[26] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 3–11, Cham, 2015. Springer International Publishing.

[27] Raphaël Cauderlier and Mihaela Sighireanu. A verified implementation of the bounded list container. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 172–189, Cham, 2018. Springer International Publishing.

[28] Arthur Charguéraud. *Characteristic Formulae for Mechanized Program Verification*. PhD thesis, Université Paris 7, 2010. `http://www.chargueraud.org/arthur/research/2010/thesis/`.

[29] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP)*, pages 418–430, Tokyo, Japan, September 2011. ACM.

[30] Arthur Charguéraud. Pretty-big-step semantics. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 41–60. Springer, March 2013.

[31] Arthur Charguéraud, Jean-Christophe Filliâtre, Mário Pereira, and François Pottier. VOCAL – A Verified OCaml Library. ML Family Workshop, September 2017.

[32] Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning*, September 2017.

[33] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

[34] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, 1941.

[35] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

[36] Martin Clochard, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Formalizing semantics with an automatic program verifier. In Dimitra Giannakopoulou and Daniel Kroening, editors, *6th Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE)*, volume 8471 of *Lecture Notes in Computer Science*, pages 37–51, Vienna, Austria, July 2014. Springer.

[37] Martin Clochard, Jean-Christophe Filliâtre, and Andrei Paskevich. How to avoid proving the absence of integer overflows. In Arie Gurfinkel and Sanjit A. Seshia, editors, *7th Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE)*, volume 9593

of *Lecture Notes in Computer Science*, pages 94–109, San Francisco, California, USA, July 2015. Springer.

[38] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009.

[39] Cyrille Comar, Johannes Kanig, and Yannick Moy. Integrating formal program verification with testing. In *Proceedings of the Embedded Real Time Software and Systems conference*, ERTS^2 2012, February 2012.

[40] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A parallel SMT-based model checker for parameterized systems. In Madhusudan Parthasarathy and Sanjit A. Seshia, editors, *CAV 2012: Proceedings of the 24th International Conference on Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, Berkeley, California, USA, July 2012. Springer.

[41] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[42] Catarina Coquand. Agda, 2000. `http://www.cs.chalmers.se/~catarina/agda/`.

[43] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[44] Patrick Cousot. Methods and logics for proving programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 841–993. North-Holland, 1990.

[45] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[46] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM.

[47] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *ESOP*, number 3444 in Lecture Notes in Computer Science, pages 21–30, 2005.

[48] Sylvain Dailler, Claude Marché, and Yannick Moy. Lightweight interactive proving inside an automatic program verifier. In *Proceedings of the Fourth Workshop on Formal Integrated Development Environment, F-IDE, Oxford, UK, July 14, 2018*, 2018.

[49] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[50] Stijn de Gouw, Frank de Boer, and Jurriaan Rot. Proof pearl: The key to correct and stable sorting. *Journal of Automated Reasoning*, 53(2):129–139, Aug 2014.

[51] Stijn de Gouw, Frank S de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. Verifying openjdk's sort method for generic collections. *Journal of Automated Reasoning*, 2017.

[52] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. *OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case*, pages 273–289. Springer International Publishing, Cham, 2015.

[53] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[54] Edsger W. Dijkstra. Notes on structured programming. In O. Dahl, E. Dijkstra, and C. Hoare, editors, *Structured programming*. Academic Press, 1971.

[55] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975.

[56] Claire Dross, Jean-Christophe Filliâtre, and Yannick Moy. Correct Code Containing Containers. In *5th International Conference on Tests and Proofs (TAP'11)*, volume 6706 of *Lecture Notes in Computer Science*, pages 102–118, Zurich, June 2011. Springer.

[57] Catherine Dubois and Renaud Rioboo. Verified functional iterators using the focalize environment. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods*, pages 317–331, Cham, 2014. Springer International Publishing.

[58] Jean-Christophe Filliâtre. Backtracking iterators. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, September 2006.

[59] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(5):397–403, August 2011.

[60] Jean-Christophe Filliâtre. One logic to use them all. In *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Artificial Intelligence*, pages 1–20, Lake Placid, USA, June 2013. Springer.

[61] Jean-Christophe Filliâtre. Le petit guide du bouturage, ou comment réaliser des arbres mutables en OCaml. In *Vingt-cinquièmes Journées Francophones des Langages Applicatifs*, Fréjus, France, January 2014. `https://www.lri.fr/~filliatr/publis/bouturage.pdf`.

[62] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. In Armin Biere and Roderick Bloem, editors, *26th International Conference on Computer Aided Verification*, volume 8859 of *Lecture Notes in Computer Science*, pages 1–16, Vienna, Austria, July 2014. Springer.

[63] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. A pragmatic type system for deductive verification. Research report, Université Paris Sud, 2016. `https://hal.archives-ouvertes.fr/hal-01256434v3`.

[64] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, 2016.

[65] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[66] Jean-Christophe Filliâtre and Mário Pereira. Itérer avec confiance. In *Vingt-septièmes Journées Francophones des Langages Applicatifs*, Saint-Malo, France, January 2016.

[67] Jean-Christophe Filliâtre and Mário Pereira. A modular way to reason about iteration. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *8th NASA Formal Methods Symposium*, volume 9690 of *Lecture Notes in Computer Science*, Minneapolis, MN, USA, June 2016. Springer.

[68] Jean-Christophe Filliâtre, Mário Pereira, and Simão Melo de Sousa. Vérification de programmes fortement impératifs avec Why3. In Sylvie Boldo and Nicolas Magaud, editors, *Vingt-neuvièmes Journées Francophones des Langages Applicatifs*, Banyuls-sur-mer, France, January 2018.

[69] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *SIGPLAN Not.*, 28(6):237–247, June 1993.

[70] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[71] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, Nov 1986.

[72] Léon Gondelman. *A Pragmatic Type System for Deductive Software Verification.* Thèse de doctorat, Université Paris-Saclay, 2016.

[73] Thérèse Hardin, François Pessaux, Pierre Weis, and Damien Doligez. *FoCaLiZe – Reference Manual*, 0.9.2 edition, June 2018.

[74] J. R. Hindley. The principal type scheme of an object in combinatory logic. Transactions of the American Mathematical Society, 146:29–60, 1969.

[75] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[76] C. A. R. Hoare. Proof of a program: Find. *Commun. ACM*, 14:39–45, January 1971.

[77] William A. Howard. The formulae-as-types notion of construction. In J. Roger Hindley Jonathan P. Seldin, editor, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[78] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.

[79] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

[80] Bart Jacobs, Frank Piessens, and Wolfram Schulte. VC generation for functional behavior and non-interference of iterators. In *Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems*, SAVCBS '06, pages 67–70, New York, NY, USA, 2006. ACM.

[81] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[82] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–259, Mumbai, India, January 2015. ACM.

[83] Johannes Kanig. *Spécification et preuve de programmes d'ordre supérieur*. Thèse de doctorat, Université Paris-Sud, 2010.

[84] I. T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–288, May 2011.

[85] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, pages 268–283, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[86] Ioannis T. Kassios. *A Theory of Object Oriented Refinement*. PhD thesis, Toronto, Ont., Canada, Canada, 2006. AAINR21796.

[87] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.

[88] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[89] Jason Koenig and K. Rustan M. Leino. Programming language features for refinement. In John Derrick, Eerke A. Boiten, and Steve Reeves, editors, *Proceedings 17th International Workshop on Refinement, Refine@FM 2015, Oslo, Norway, 22nd June 2015.*, volume 209 of *EPTCS*, pages 87–106, 2015.

[90] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*, pages 696–723, New York, NY, USA, 2017. Springer-Verlag New York, Inc.

[91] Gregory Kulczycki, Murali Sitaraman, Joan Krone, Joseph E. Hollingsworth, William F. Ogden, Bruce W. Weide, Paolo Bucci, Charles T. Cook, Svetlana Drachova-Strang, Blair Durkee, Heather K. Harton, Wayne D. Heym, Dustin Hoffman, Hampton Smith, Yu-Shan Sun, Aditi Tagore, Nighat Yasmin, and Diego Zaccai. A language for building verified software components. In John M. Favaro and Maurizio Morisio, editors, *Safe and Secure Software Reuse - 13th International Conference on Software Reuse, ICSR 2013, Pisa, Italy, June 18-20. Proceedings*, volume 7925 of *Lecture Notes in Computer Science*, pages 308–314. Springer, 2013.

[92] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. Software verification with itps should use binary code extraction to reduce the TCB - (short paper). In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 362–369. Springer, 2018.

[93] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM.

[94] Daniel H. Larkin, Siddhartha Sen, and Robert E. Tarjan. *A Back-to-Basics Empirical Study of Priority Queues*, pages 61–72.

[95] Fabrice Le Fessant and Luc Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*. ACM Press, 2001.

[96] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106, 2000.

[97] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[98] Xavier Leroy. Coinductive big-step operational semantics. In *ESOP 2006: European Symposium on Programming*, number 3924 in LNCS, pages 54–68. Springer, 2006.

[99] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[100] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207:284–304, February 2009.

[101] Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer, 2003.

[102] Pierre Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. Thèse de doctorat, Université Paris-Sud, July 2004.

[103] P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report 25.085, IBM Laboratory, Vienna, June 1968.

[104] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[105] Luc Maranget. Compiling pattern matching to good decision trees. In *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46, New York, NY, USA, 2008. ACM.

[106] Claude Marché. Jessie: an intermediate language for Java and C verification. In *Programming Languages meets Program Verification (PLPV)*, pages 1–2, Freiburg, Germany, 2007. ACM Press.

[107] Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

[108] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.

[109] Guillaume Melquiond and Raphaël Rieu-Helft. A Why3 framework for reflection proofs and its application to GMP's algorithms. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *9th International Joint Conference on Automated Reasoning*, Lecture Notes in Computer Science, Oxford, United Kingdom, July 2018.

[110] R. Milner. A theory of type polymorphismn in programming. *Journal of Computer and System Sciences*, 17, 1978.

[111] J-F. Monin. *Understanding Formal Methods*. Springer Verlag, 2002.

[112] Yannick Moy and Claude Marché. *The Jessie plugin for Deduction Verification in Frama-C — Tutorial and Reference Manual*. INRIA & LRI, 2011. `http://krakatoa.lri.fr/`.

[113] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Œuf: Minimizing the coq extraction tcb. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 172–185, New York, NY, USA, 2018. ACM.

[114] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[115] Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. *The Art of Software Testing, Second Edition*. Wiley, June 2004.

[116] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[117] Peter W. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 13–25, New York, NY, USA, 2018. ACM.

[118] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[119] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga Springs, NY, June 1992. Springer.

[120] Karen Hunger Parshall. The genesis of the abstract group concept: A contribution to the history of the origin of abstract group theory. *The American Mathematical Monthly*, 93(10):823–826, 1986.

[121] Andrei Paskevich. Algebraic types and pattern matching in the logical language of the Why verification platform (version 2). Technical Report 7128, INRIA, 2010. `http://hal.inria.fr/inria-00439232/en/`.

[122] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM Press.

[123] Christine Paulin-Mohring. Inductive definitions in the system COQ. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

[124] Mário Pereira. Défonctionnaliser pour prouver. In Sylvie Boldo and Julien Signoles, editors, *Vingt-huitièmes Journées Francophones des Langages Applicatifs*, Gourette, France, January 2017.

[125] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[126] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[127] Nadia Polikarpova. *Specified and Verified Reusable Components*. PhD thesis, ETH Zurich, April 2014.

[128] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. A fully verified container library. In Nikolaj Bjørner and Frank D. de Boer, editors, *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, volume 9109 of *Lecture Notes in Computer Science*, pages 414–434. Springer, 2015.

[129] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. A fully verified container library. *Formal Asp. Comput.*, 30(5):495–523, 2018.

[130] Nadia Polikarpova, Julian Tschannen, Carlo A. Furia, and Bertrand Meyer. Flexible invariants through semantic collaboration. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 514–530. Springer, 2014.

[131] François Pottier. Verifying a hash table and its iterators in higher-order separation logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*, January 2017.

[132] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003. ©ACM.

[133] Yann Régis-Gianas and François Pottier. A Hoare logic for call-by-value functional programs. In *Proceedings of the Ninth International Conference on Mathematics of Program Construction (MPC'08)*, pages 305–335, 2008.

[134] W. Reif, G. Schnellhorn, and K. Stenzel. Proving system correctness with KIV 3.0. In William McCune, editor, *14th International Conference on Automated Deduction*, Lecture Notes in Computer Science, pages 69–72, Townsville, North Queensland, Australia, july 1997. Springer.

[135] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[136] Raphaël Rieu-Helft, Claude Marché, and Guillaume Melquiond. How to get an efficient yet verified arbitrary-precision integer library. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments*, volume 10712 of *Lecture Notes in Computer Science*, pages 84–101, Heidelberg, Germany, July 2017.

[137] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169. ACM, 2008.

[138] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[139] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.

[140] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.6*, 2016. `http://coq.inria.fr`.

[141] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.

[142] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. Autoproof: Auto-active functional verification of object-oriented programs. In *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science. Springer, 2015.

[143] Alan Mathison Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculing Machines*, pages 67–69, Cambridge, 1949. Mathematical Laboratory.

[144] Bruce W. Weide. SAVCBS 2006 challenge: Specification of iterators. In *Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems*, SAVCBS '06, pages 75–77, New York, NY, USA, 2006. ACM.

[145] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1992.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z

Synthèse

Cette thèse se place dans le cadre des méthodes formelles et plus précisément dans celui de la vérification déductive. La vérification déductive propose un ensemble d'approches pour transformer la correction d'un code d'un logiciel en un énoncé mathématique et après prouver sa validité. À fin de générer un tel énoncé, le programme doit être munis d'une description mathématique de son comportement pendant l'exécution, ce qui est traditionnellement nommée comme la spécification fonctionnel. En étant capable de montrer la correction de l'énoncé généré, on peut affirmer que le programme est correct vis-à-vis de sa spécification. Plusieurs outils existants mettent en place l'approche de la vérification déductive pour vérifier la correction des logiciels. On peut citer les assistants de preuve manuels tels que Coq, PVS, Isabelle ou KeY, mais aussi les démonstrateurs automatiques tels que les solveurs SMT comme Alt-Ergo, CVC4, Z3 et aussi les démonstrateurs de la famille TPT, comme SPASS ou Vampire. Par ailleurs, ils existent des outils qui proposent une interface entre l'utilisateur et les démonstrateurs de théorèmes. Ces outils proposent ce qui est traditionnellement appelé un langage dédié à la preuve, aussi bien qu'un mécanisme automatique pour la génération de l'énoncé de correction d'un programme. Dans cette classe d'outils on peut citer Dafny, Viper ou encore Why3.

Tout au long de cette thèse, le système Why3 a été utilisé, d'une part, comme l'environnement pour expérimenter les idées développées, et d'autre part comme la cible de certaines contributions de cette thèse. Why3 fournit un ensemble d'outils pour la spécification, l'implémentation et la vérification à l'aide de démonstrateurs externes. Why3 propose en particulier un langage de programmation adapté à la preuve, appelé WhyML. Un aspect important de ce langage est le code fantôme, à savoir des éléments de programme introduits exclusivement pour les besoins de la spécification et de la preuve. Pour obtenir un code exécutable, le code fantôme est éliminé par un processus automatique appelé extraction. L'une des contributions principales de cette thèse est la formalisation et l'implémentation du mécanisme d'extraction de Why3. La formalisation consiste à montrer que le programme extrait préserve la sémantique du programme de départ, en s'appuyant notamment sur un système de types avec effets.

Le nouveau mécanisme d'extraction de Why3 a été utilisé avec succès pour obtenir plusieurs modules OCaml corrects par construction, dans le cadre d'un projet de recherche plus large, le projet VOCaL. Le but ce projet est de construire une bibliothèque formellement vérifiée de structures de données et d'algorithmes, écrits dans le langage OCaml. Dans le cadre du projet VOCaL, les travaux de cette thèse ont contribué au développement d'un langage de spécification pour OCaml. Un tel langage à été utilisé pour spécifier formellement les éléments introduits dans des fichiers .mli, les fichiers interfaces du langage OCaml. L'utilisation de ce langage permet, notamment, de montrer qu'une structure de données ou algorithme implémenté et vérifié en WhyML est bien un raffinement de la spécification donnée dans l'interface.

Cet effort de preuve de programmes OCaml a conduit à trois autres contributions de cette thèse. La première est une technique systématique pour la vérification de structures avec pointeurs, à l'aide de modèles du tas délimités. Une preuve entièrement automatique d'une structure union-find a pu être obtenue grâce à cette technique. La seconde est un mécanisme systématique pour extraire du code fonctoriel à partir d'un programme vérifié avec Why3. Cela a permis notamment d'extraire des foncteurs OCaml, correctes par construction, pour deux implantations différentes de

filles de priorité, à savoir des pairing heaps et une variante impérative, où la fille est codée dans un tableau redimensionnable. Finalement, la dernière contribution est un moyen de spécifier un algorithme d'itération indépendamment de son implémentation. Plusieurs curseurs et itérateurs d'ordre supérieur ont été spécifiés et vérifiés en utilisant cette approche. Le point en commun, et les important, sur ces contributions est le fait qu'aucun des ces trois aspects idiomatiques de la programmation en OCaml est supporté directement par le langage WhyML et la preuve de programmes avec Why3.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Outils et techniques pour la vérification de programmes impératives modulaires

Mots clés : Vérification déductive, Why3, Effets, Bibliothèque OCaml, Modulaire

Résumé : Cette thèse se place dans le cadre des méthodes formelles et plus précisément dans celui de la vérification déductive et du système Why3. Ce dernier fournit un ensemble d'outils pour la spécification, l'implémentation et la vérification à l'aide de démonstrateurs externes. Why3 propose en particulier un langage de programmation adapté à la preuve, appelé WhyML. Un aspect important de ce langage est le code fantôme, à savoir des éléments de programme introduits exclusivement pour les besoins de la spécification et de la preuve. Pour obtenir un code exécutable, le code fantôme est éliminé par un processus automatique appelé extraction. L'une des contributions principales de cette thèse est la formalisation et l'implémentation du mécanisme d'extraction de Why3. La formalisation consiste à montrer que le programme extrait préserve la sémantique du programme de départ, en s'appuyant notamment sur un système de types avec effets. Ce mécanisme d'extraction a été utilisé avec succès pour obtenir plusieurs modules OCaml corrects par construction, dans le cadre d'une bibliothèque vérifiée de structures de données et d'algorithmes. Cet effort de preuve a conduit à deux autres contributions de cette thèse. La première est une technique systématique pour la vérification de structures avec pointeurs, à l'aide de modèles du tas délimités. Une preuve entièrement automatique d'une structure union-find a pu être obtenue grâce à cette technique. La seconde contribution est un moyen de spécifier un algorithme d'itération indépendamment de son implémentation. Plusieurs curseurs et itérateurs d'ordre supérieur ont été spécifiés et vérifiés en utilisant cette approche.

Title : Tools and Techniques for the Verification of Modular Stateful Code

Keywords : Deductive verification, Why3, Effects, OCaml library, Modular

Abstract : This thesis is set in the field of formal methods, more precisely in the domain of deductive program verification. Our working context is the Why3 framework, a set of tools to implement, formally specify, and prove programs using off-the-shelf theorem provers. Why3 features a programming language, called WhyML, designed with verification in mind. An important feature of WhyML is ghost code: portions of the program that are introduced for the sole purpose of specification and verification. When it comes to get an executable implementation, ghost code is removed by an automatic process called extraction. One of the main contributions of this thesis is the formalization and implementation of Why3's extraction. The formalization consists in showing that the extracted program preserves the same operational behavior as the original source code, based on a type and effect system. The new extraction mechanism has been successfully used to get correct-by-construction OCaml modules, which are part of a verified OCaml library of data structures and algorithms. This verification effort led to two other contributions of this thesis. The first is a systematic approach to the verification of pointer-based data structures using ghost models of fragments of the heap. A fully automatic verification of a union-find data structure was achieved using this technique. The second contribution is a modular way to reason about iteration, independently of the underlying implementation. Several cursors and higher-order iterators have been specified and verified with this approach.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France