About some Metamorphoses of Computer Programs

Yann Regis-Gianas

HABILITATION
À DIRIGER DES RECHERCHES
DE
L'UNIVERSITÉ PARIS DIDEROT

Spécialité Informatique
École doctorale 386
Science Mathématiques Paris-Centre

About some Metamorphoses of Computer Programs

Mémoire d'habilitation à diriger des recherches présenté et soutenu publiquement par

Yann Régis-Gianas

le 22 novembre 2019

devant le jury composé de

Président du jury	
Xavier Leroy	Collège de France
Rapporteurs	
Giuseppe Castagna	CNRS – IRIF
Peter Thiemann	Universität Freiburg
Stephanie Weirich	University of Pennsylvania
Jury	
Sandrine Blazy	Université de Rennes 1
Roberto di Cosmo	INRIA – IRIF

Contents

Remerciements

Puisque j'ai le sentiment d'avoir terminé mon manuscrit de thèse hier, permettez-moi de filer la métaphore commencée dans ses remerciements.

Quand j'ai quitté l'oasis de Rocquencourt pour rejoindre le palais doré de Chevaleret, mon code source en construction s'est senti comme arraché à son environnement d'évaluation naturel : où étaient donc passées ces variables libres (François P., Didier R., Michel M., Xavier L., Luc M., Damien D., Zaynah D. et Boris Y.) dont les valeurs guidaient si finement mon évaluation? Comment éditer des liens avec ces entités naviguant dans les intimidantes catégories aux ordres infiniment supérieurs (Pierre-Louis C., Paul-André M., Thomas E., Michele P., Daniele V., Christine T., Alexandre M.)? Comment adapter ma syntaxe à la calligraphie rigoureuse des réécrivains talentueux du royaume de PPS (Delia K., Yves G.)? Comment rattrapper sans faillir les exceptions humouristiques des doctorants et autres postdocs que j'y croiserai (Pierre B., Mehdi D., Pierre-Marie P., Gabriel K., Stéphane G., Thibaut B., Charles G., Etienne M., Victor L., Kostia C., Vincent S., Nicolas T., Guillaume M., Marie K., Nicolas A., François B., Amina D., Guilhem J.)?

Cette torpeur ne dura qu'un instant. Le lieu était en vérité rempli de valeurs chaleureuses, accueillantes et fonctionnelles (Vincent B., Roberto D.C., Jérôme V.) qui m'ont vite intégrer à l'écosystème de PPS – une sorte de village non pas gaulois mais grec où la parole est libre et les idées s'échangent plus vite que les ACKs sur l'Internet (Juliusz C.). On m'a même donné les cours d'un Master en offrande (sans me préciser toutefois que je devrai analyser syntaxiquement des scripts shell en échange [Ralf T., Claude M.]). On a aussi étendu mes fonctionnalités en m'initiant à une science stellaire (Roberto A.), en me plongeant dans un poulailler en construction perpétuelle (Hugo H., Pierre L., Matthieu S.), et en donnant à mon interface homme-machine une plus grande définition de ce qu'est l'être humain (Alexis S., Jean K., Pierre-Louis C., Odile A., Christine T.). Encore aujourd'hui, certains osent initier toute communication à mon endroit par un "jeune homme!" en faisant semblant d'ignorer mes maladroitesses de papa sans sommeil (Thomas C.) et d'autres m'appellent "prof" en me chantant mes airs préférés (Daniel D.R.). Seule ombre au tableau, je cherche toujours quelques signatures pour obtenir la radiation d'un collègue dont les traits d'esprit m'obligent à sourire beaucoup trop régulièrement (A. Guatto). L'IRIF, ce sont avant-tout des personnes mais c'est aussi l'environnement rêvé pour s'exécuter sans compter, et cela ne doit rien au hasard (Frédéric M., Eva R., Dieneba C., Etienne M., Giuseppe C., Michele P., Odile A.).

Que ferai-je donc de tous ces trésors? Que faire de mieux que de les partager? Plus encore, faisant fi des préceptes de la logique linéaire, il fallait non seulement les partager mais les dupliquer! Peu importe les ressources consommées! Des étudiants de la maternelle au doctorat (Mathias P., Lourdes G., Thibaut G., Guillaume C., Nicolas J., Théo Z. et Colin G.) ont donc subi, ou subissent encore, mes effets de bord, mes observables, et autres bogues, au profit, je l'espère, de quelques idées bien typées. Ces étudiants sont au final ce qui importe le plus dans ce métier, et fort heureusement pour eux, mes protocoles d'enseignements ont été réécrits maintes fois par mes inestimables collègues (Jean-Baptiste Y., Jean-Marie R., Inès K., Dominique P., Anne M., Vincent P., Michel H., Christophe P., Antonio B., Aldric D., Giovanni B., Ahmed B., Constantin E., Bérénice D., Nicolas S., Arnaud S., Sylvain P., Peter H., Hugues F., Carole D., Sammy A., Mihaela S., Pierre-Evariste D., Olivier C., Wieslaw Z., Yan J., François A., Eugène A., Sylvain P., François L., Valia M., Olivier S., Stefano Z.) et par ceux avec qui tentent de casser les pare-feux inter-campus à grands coups de sympathiques concours ludiques de programmation (Emmanuel C., Sylvain C., Jean-Christophe F., Pascal M.). Notez bien que ce système cafouillerait sans arrêt s'il n'avait pas ses administrateurs de l'ombre pour le faire vivre (Jean-Michel M., Nicolas L., Laurent P., Patricia T., Mickaël F., Raja T., Sylvia C., Léa F., Charlotte S., Sammy A., Christine D.S., Anne B., Jacqueline M.A., Carole D., Hugues F.).

Pendant ces interstices qui séparent mes trop nombreux changements de contexte, j'aime à observer ce que tous ces programmes dont j'ai vu s'écrire les premières lignes font d'extraordinaire aujourd'hui (Frédéric B., Gabriel S., Gabriel R., Thomas R., Paul L., Loïc R., David B.). Ils continuent à évoluer là-bas, hors du village, là où je rencontre continuellement de brillants ingénieurs qui construisent des preuves pour la France (Thomas L., Arnaud F.), des locomotives certifiées (David M.), des Coqs dans les navigateurs (Benoît R., Guillaume D.), des virtualisations autogérées de Bourg-Palette (Benjamin C., Grégoire H.) ou encore des chameaux plus solides (Louis G., Muriel S.). Grâce à Michel M., j'ai d'ailleurs la chance de contribuer à la Fondation OCaml, qui cherche à remplacer les serpents qui piquent par des chameaux qui portent, pour une programmation plus sûre.

Un jour de pluie, Ralf T. a annoncé : il faut faire une release! Je pensais qu'il parlait de la version 0.9 de Morbig mais il faisait allusion à l'auto-documentation que vous avez entre les mains. Sans lui, ni Stephanie W., ni Peter T., et encore moins Giuseppe C., n'auraient eu à endurer cette relecture. Je leur en serai redevable longtemps mais qu'ils pardonnent à Ralf T! Il pensait bien faire! Je le remercie infiniment pour le soin qu'il porte quotidiennement au petit programme que je suis ainsi qu'à la distribution Debian dans laquelle je m'exécute depuis 20 ans. Que dire aussi à Xavier L. qui après m'avoir donné un langage dont la musique rythme l'essentiel de mes heures de programmation, me fait l'honneur d'orchestrer ce moment où je vais m'essayer à quelques improbables vocalises? Je remercie aussi Sandrine B. d'être de mes béta-testeurs, alors qu'elle est si sollicitée tant par l'enseignement que par la recherche. Si Ralf est le responsable de cette release et probablement mon mainteneur en chef, Roberto D.C. en est le chef de projet : de l'étuve souterraine où nous avons filmé le MOOC OCaml à son bureau de l'IRIF où il m'a raconté ces nombreuses histoires riches en enseignement, il n'a cessé depuis 12 ans d'être une source d'inspiration et un modèle. Je ne remercierai jamais assez Delia K. d'avoir mené plusieurs batailles importantes pour l'UFR et d'être un invariable soutien de si bon conseil, P. Letouzey d'(é)prouver le monde avec moi entre les murs du bureau 3128, Paul-André M. et Hugo H. pour me rappeler que la recherche est un échange d'idées, et Pierre-Louis C. pour avoir construit tout ça.

Que le temps passe et qu'il est loin celui où j'écrivais mon premier analyseur syntaxique en bonne compagnie (Pierre-Yves S.)! Peu importe, il faut à tout prix préserver les invariants qui vous font avancer un pas après l'autre (Olivier, Hélène). J'ai la chance d'avoir deux sœurs, un frère et un beau-frère toujours prêts à répondre à mes appels systèmes à l'aide d'une bonne dose de rires, de musiques ou d'adorables enfantillages. Merci à ma mère, mon père et Pierre de veiller sur ce petit écosystème si fragile.

Quant à Marine, elle est tout à la fois : l'exponentiation aux ressources infinies, la dualité parfaite, l'opposante bienveillante qui initie chacun de mes mouvements. Ses exploits sont innombrables! Mais, l'un d'entre eux les dépasse tous : celui d'avoir créer ces trois singulières différences – Arthur, Alice et Félix – qui ont révolutionné à eux-seuls les fondements premiers de toute ma sémantique.

Introduction

> Mathematics is the art of giving the same name to different things.
>
> Henri Poincaré

> There are only two hard things in Computer Science: cache invalidation and naming things.
>
> Phil Karlton

Software is an object placed on a wire in tension between logic and mechanics, between staticity and dynamicity, between conceptualism and pragmatism. In the eyes of the programmer, who must successively be at one end or the other of this thread, a piece of software is never limited to its source code. On the contrary, the programmer sees his program under various forms, and each of these forms exists for a very specific purpose. This manuscript is about a few of these "metamorphoses" that are explicitly or implicitly applied to programs to execute them, to understand them or to change them.

Some of these metamorphoses have been known for a long time as the transformation of a source program into machine code or of a text into an abstract syntax tree ; others have been formally defined more recently as the transformation of a program into its "derivative" or the transformation of a program from one version to the next one. Whatever the transformation, we apply to these metamorphoses the exact same systematic treatment: to interpret them with as much logic as possible [14] to give them meaning, to reason about their properties, and, echoing the two quotes above, to try to "name" them.

Let us be more concrete from now and consider the following OCaml [15] program, that will allow us to introduce the first four chapters of this document.

```
1 let rows = Csv.load "grades.csv" in
2 let grades = List.map (fun ℓ -> List.nth ℓ 2 |> int_of_string) rows in
3 let mean = List.fold_left ( + ) 0 grades / List.length grades in
4 Csv.save "statistics.csv" [ [ string_of_int mean ] ]
```

This program computes the average of grades stored in the file named "grades.csv". In this file, each line contains the surname, the first name and the grade of a student. On Line 2, the list of grades is built by applying a function that extracts the field numbered 2[1] of each line and converts the resulting string into a number. On Line 3, the program traverses this list of grades to compute their sum and divides it by the length of the list to return the average grade. Finally, this grade is stored in a file named "statistics.csv".

Chapter 1: Certified effectful functional programming When reading this source code, a seasoned programmer will certainly feel uncomfortable. This programmer must have immediately detected several potential problems in that program: for instance, how should it behave if the input file is empty, if it is not formatted as expected, if the input file does not exist or if the user has not enough privileges to create the output file? This question is actually double since it makes us wonder about the precise specification – the assumptions and the promises – of this program, and it forces us to consider the correctness of the program with respect to that specification.

[1] In Computer Science, we always index from 0.

The first chapter will focus on this double problematic through the presentation of our contributions to the field of *certified effectful functional programming*. This line of work develops the idea that functional programming in Type Theory [18], especially using the Coq proof assistant [7], is an effective programming environment for designing software together with their proof of correctness. In Coq, our example OCaml program is transformed into a total function, i.e., a mathematical function that associates a result with any element of its domain. Through these glasses, the calculation of the average cannot be accepted unless we remove the empty input file from the domain of the program. Hence, we are either forced to specify the prerequisite on the inputs explicitly or to have this program be preceded by a dynamic test to produce an error message when this problematic situation arises. Likewise, the extraction of grades will impose the input file to contain a third column made of integers.

Coq is especially well-suited to define and to verify purely functional computations, computations that are naturally expressible as total functions. The reasons for this adequacy are profound because they originate from a fundamental correspondence between proofs and program – the Curry-Howard correspondence [10] – which Coq is probably one of the most sophisticated realizations.

Nevertheless, the two problems related to the input/output of our example program are more delicate to deal with because the execution environment of the program enters the scene. In this environment, we find the file system with which the program interacts to load and to store its data. At first glance, this interaction may seem incompatible with the program-as-total-function perspective: indeed, two distinct evaluations of a mathematical function on the same input always produce the same output while two different interactions with an environment can have distinct effects. To be convinced, it suffices to think about a function that returns the current time or that creates a new file. The answers to these requests performed twice are not identical! Should we conclude that we cannot reason about their behavior mathematically?

To reach this conclusion, we would have to forget the expressive power of mathematics, that can model changing processes if their parameters are made explicit. From that perspective, we could therefore see our OCaml program as a total function of its environment to fall back to a form of mathematical reasoning. Yet, if this idea works in theory, it does not resist to practice because execution environments are very complex. To take their full complexity into account, the programmer would have to reason about his program and its environment globally. Caricaturing a little, the programmer would then have to check that all the programs being executed by the operating system do not modify the files on which his program is based. This is of course untractable and would prevent the method to scale.

As a consequence, we need appropriate concepts to abstract the execution environment of a program. This abstraction would allow for a local reasoning about the effects of the interactions between the program and its environment. We study that question through multiple refinements of the monadic approach introduced by Moggi [19] and popularized by Wadler [22]. The precise description of these contributions will be the topic of Chapter 1 but we can summarize them as follows:

- In collaboration with Beta Ziliani, Lourdes del Carmen Gonzalez Huesca and Guillaume Claret, we introduce simulable monads, a specific class of monads that revalidate in Type Theory unsafe executions of effectful computations. Simulable monads are especially useful to develop the decision procedures invoked by proofs by reflection.

- In collaboration with Beta Ziliani, Jan-Oliver Kaiser, Robbert Krebbers, Derek Dreyer, Kate Deplaix, Béatrice Carré et Thomas Refis, we present Mtac*2*, a free monad to use Coq as a meta-programming and tactic language for Coq. Mtac2 exploits dependent types to guarantee the robustness of proof scripts.

- In collaboration with Thomas Letan, Pierre Chifflier, Guillaume Hiet, Vincent Tourneur and Guillaume Claret, we explore several formulation of free monads to develop, to specify and to prove correct interactive applications within Coq. The Ph.D. thesis of Letan [16] gives a promising answer to this question with FreeSpec, a framework for modular development of certified interactive applicative software.

Chapter 2: Incremental programming As students are often making us notice, we sometimes assign them wrong grades. In that case, we typically have to update the input file containing the grades and recompute the average grade by running our OCaml program again. Doing so, the program will compute the sum of a list of numbers which is very close to the list encountered during its last invocation. If we had kept the previous sum, we could take the change of this list into account by a mere subtraction followed by an addition. With an additional division, we would get the new value for the average grade in only three arithmetic operations. Such a computation would be *incremental* with respect to the previous one, in the sense that it would share a large number of subcomputations with it. This optimization is perhaps not justified in practice if our file only contains a hundred grades. However, with today's big data systems processing ever-changing large volumes of data, the question of incrementality seems crucial if not economically, at least ecologically speaking.

Let us write down the update applied to our file of grades as follows:

On Line 5, the value of Column 2 changes from 19 to 14.

We have no immediate way to exploit this difference using our OCaml program. Ideally, this change should trigger the computation of another change to apply to the stored average grade. Yet, to achieve that, we would have to significantly modify the program source code.

Still, this program source code already has all the elements to connect the output with its input, and therefore to connect the output change with its input change. Why couldn't we use this piece of information to automatically transform the program into an incremental version of itself? In Chapter 2, we answer this question positively: it is indeed possible to differentiate a function f expressed as a term of the λ-calculus to obtain a function that we will name – maybe abusively – a derivative of f, that computes output changes from input changes.

Here again, practice demands extra refinements. Indeed, even though a derivative of f can be automatically generated, its efficiency is not guaranteed. There are two reasons for that. First, in its most natural formulation, the incrementalization of λ-terms introduces computational redundancies between f and its derivatives. We have to significantly modify the transformation to factorize these computations out. The idea is to have the function communicate its intermediate results to the derivative to avoid recomputations. This form of static memoization is obtained by translating the initial program and its derivative into a "cache passing style". Second, there are sometimes situations where the mathematical properties of f can be exploited to get the right algorithmic complexity of its derivative. In that case, only the programmer can define the optimal derivative. However, manual incrementalization of programs is an error prone activity, and, in our opinion, it requires a proof assistant to be conducted safely.

Our contributions on the topic of incremental programming can be sum up as follows:

- In collaboration with Lourdes del Carmen Gonzalez Huesca, we propose a deterministic differential λ-calculus which includes a dynamic differentiation operator. The definition of this operator is based on change structures, algebraic objects that give changes the status of first class citizens.
- In collaboration with Paolo Giarrusso and Philipp Schuster, we improve on the static differentiation of Giarrusso's Ph.D. [9] by extending Liu's cache passing style [17] to higher-order programs.
- We propose a framework for certified incremental functional programming based on Coq and OCaml.

Chapter 3: Certified software evolutions As Theseus' boat, a program slowly evolves along many small source code patches crafted by programmers. Most of these changes have an impact on the program's semantics and, after some time, it may become difficult to know if we are still in front of the original program. Strictly speaking, we don't. We indeed constantly change the implementation or the specification of a program along its life. This evolution is not problematic as long as all these changes are made consciously and their implications are fully understood.

Unfortunately, we have very few tools to characterize software evolutions. In practice, we are bound to review patches manually or to use well-designed test suites to convince ourselves that a supposed refactoring indeed preserves the program semantics, that a bugfix is correctly removing some misbehavior, or that the introduction of a new feature has no unforeseen interaction with the other components of a system. To formally apprehend the impact of changes is even more critical in the case of certified software: industrial software certifications is so costly that we would save a lot of time and energy by transferring as much proofs as possible from one version to the next one.

Comparing two programs' semantics amounts to establish that a given relation holds between two potentially infinite execution traces. It is a well-studied topic in two particular cases: program equivalence and program simulation. In these two cases, the set of behaviors of one program includes the other program's. Here, we are interested in a more general question because software evolution can induce a set of behaviors that has no inclusion relationship with the behaviors of the original program. This relation can be arbitrarily complex, and thus very difficult to establish.

By chance, we usually compare "close" programs: there usually exist correlating points where we expect some relation to hold between the configurations of the two programs. Benton [3]'s Relational Hoare Logic exploits this remark in a program logic for relational reasoning on pairs of programs. This logic generalizes Hoare triples to characterize two commands instead of only one. In Relational Hoare Logic, the triple $\{P\}c_1 \sim c_2\{Q\}$ is read "Assuming that the union of the two programs' states satisfies the precondition P, then the execution of the two commands c_1 and c_2 produce two states whose union satisfies the postcondition Q".

Relational Hoare Logic has drawbacks. First, it assumes that we are able to find a finitary correlation between proof trees. Second, contrary to standard Hoare Logic, it does not provide a procedure to reduce program verification to the validity of a set of proof obligations. Third, Relational Hoare Logic cannot compare programs that diverge or crash. These cases are however important to characterize bugfix for instance.

Our contributions on the certification of software evolutions are the following ones:

- In collaboration with Thibaut Girka and David Mentré, we define an algorithm that generates correlating programs. A correlating program encodes a static scheduling between two programs in such a way that the proof of a Relational Hoare triple is reduced to the proof of a standard Hoare triple. Our contribution is to provide a sound algorithm for correlating programs generation while the algorithm of the literature was unsound [20].

- In collaboration with Thibaut Girka and David Mentré, we introduce the framework of correlating oracles which generalize Relational Hoare Logic to the case of nonterminating programs and of crashing programs. Using this framework, we are able to define the very first language of verifiable semantic differences.

Chapter 4: Corecursive programming Let us come back one last time to our OCaml's program and let us imagine that we want to make it evolve to a program that computes the average grade on a window sliding over an infinite list of integers. In the Haskell programming language, such a change does not require a fundamental adjustment of the program's datatypes. Indeed, Haskell's laziness gives, by construction, a potentially infinite length to lists because they can be produced by a suspended, but diverging, computation. In OCaml, which is a strict programming language, we cannot manipulate infinite lists without entering an endless process that will never give the control back: the evaluation strategy of OCaml indeed computes values entirely before they are passed as arguments to function. For infinite lists, this is clearly not the right strategy.

Of course, an OCaml programmer would point out the standard `Lazy` module. This library would allow us to simulate Haskell's lazy evaluation through the insertion of suspensions and forcings here and there to ensure a progressive consumption of our infinite list. However, if we follow that path, the resulting program is complicated by these insertions and any reasoning about its behavior has to take the operations of the module `Lazy` into account. In the end, programming with infinite objects seems a bit painful in OCaml...

Copatterns introduced by Abel et al. [1] provide a nice solution to tackle this weakness. Indeed, this line of work generalizes the syntax for patterns that we daily use to define functions by case on the shape of values. Copatterns denote an observation of a coinductive, hence potentially infinite, object. A copattern matching realizes a potentially infinite value by providing an answer to all the future observations to be made about this value. To some extent, copattern matching generalizes the notion of function, and therefore of delayed computation through a high-level syntax which discriminates between various forms of observations.

The work of Abel et al. [1] applies to Type Theory, and more specifically to Agda [5]. We transfer this idea to OCaml and we explore the programming opportunities it opens. More precisely, our contributions are the following:

- In collaboration with Paul Laforgue, we show how to extend any "sufficiently typed" programming language with copattern matching using a mere macro. We apply this technique to OCaml.

- In collaboration with Paul Laforgue, we show how this technique offers first class observations for free (by contrast, the system of Abel et al. [1] only gives second-class observations).

Chapter 5: Cost annotating compilation We know how to mechanically prove the correctness of a compiler for a realistic programming language. The CompCert project led by Leroy [13] is a compiler for a realistic subset of the C language mechanically verified using the Coq proof assistant. This achievement has a significant impact on the domain of formal methods and software certification as it demonstrates that a mechanized proof can be conducted on medium-sized complex software, and that the resulting software is of better quality than what is obtained using the traditional software engineering approach [23]. CompCert paves the way of very ambitious research projects like DeepSpec [2], whose objective is to certify a computer system from the ground up, i.e., from the processor to the applicative layer.

By looking a bit closer on the correctness property proved about CompCert, one can realize that its theorem of semantics preservation essentially characterizes the behaviors of the source program and of the compiled code. These behaviors indicate if the program diverges, terminates normally or crashes. They also describe the trace of interactions between the program and its environment. Roughly speaking, the preservation of semantics means that the behaviors of the compiled code simulate all the behaviors of the source, plus some behaviors that are undefined in the source language semantics. This theorem allows for instance the transfer of proofs that characterize the execution of the source program to get proofs that characterize the execution of the compiled code.

We sometimes want to transfer properties the other way around: from the compiled code to the source code. If we have a cost model about the execution of the compiled code, it can be useful to transport this model on the source program to reason about the concrete complexity of its execution in terms of the size of the source values.

The CerCo project tackles this question by developing a certified compiler for the C language that annotates the source program with concrete execution costs. The contributions of this project are the following ones:

- In collaboration with Roberto Amadio et Nicolas Ayache, we propose a modular architecture to extend a compiler and its proof of correctness in such a way that the compiler can produce a cost annotated source program.

- In collaboration with Roberto Amadio et Nicolas Ayache, we develop a prototype of a cost annotating C compiler and a plugin for the Frama-C verification framework that exploits these cost annotations to produce synthetic results about concrete complexity.

- In collaboration with Roberto Amadio, we apply the same method on a standard compilation chain for a functional programming language.

Chapter 6: Analysis of Posix shell scripts The design of a programming language does not always follow a reasoned path, but it always answers a need. Many semantics have inconsistencies that lead to programming errors, at least from non-expert programmers. The case of Posix shell [11] provides a famous example of such semantics, amongst others [21]. Despite the wobbliness of their semantics, the existence and the acceptance of these languages probably show that they fulfilled a particular need of a community in a way that was better than other languages at a specific time. A pragmatic and constructive approach consists in studying these languages [4], in helping in the definitions of safe subsets [12], and in developing useful static analysis to find bugs [6]. Finally, through this study, we might understand the profound reasons why they have been successful, giving insight to better design the next programming languages.

Beyond these general considerations, the CoLiS project aims at verifying a precise corpus of Posix shell scripts: the so-called maintainer scripts of the Debian operating system [8]. These programs are critical since they are executed with the administrator privileges during the installation, the removal, and the update of software packages. A programming error in one of these scripts can obviously have dramatic consequences on the computer system that executes it! Besides, we would like to check that certain properties hold about these scripts: for instance, we would like to check that the installation followed by the removal of a package is equivalent to the identity.

To achieve that, the CoLiS project transforms the scripts into symbolic evaluations in domains where decision procedures exist. The first domain currently handled by CoLiS is a language of constraints about file systems. The satisfiability of each of these constraints ensures the existence of file systems for which the program takes a specific execution path. When this execution path provokes an error, it means that there exists a file system on which the script probably crashes. The CoLiS project is still a work in progress but we can already report the following contributions:

- In collaboration with Nicolas Jeannerod et Ralf Treinen, we develop Morbig a static parser for Posix shell scripts. The specificity of this work is the usage of purely functional programming techniques to maintain a high-level of abstraction in the parser implementation in spite of the complexity and the irregularities of Posix shell syntax.

- In collaboration with Nicolas Jeannerod, Benedikt Becker, Claude Marché, Ralf Treinen et Mihaela Sighireanu, we propose a translation from a realistic subset of Posix shell to a language of constraints through symbolic execution.

Research positioning

Now that we have summarized the research projects we worked on during the last ten years, let us explain how they assemble in a more global picture. These research projects indeed live in the fields of Theory, Design and Implementation of Programming Languages, Program Verification, and Software Engineering. Figure 1 positions the Ph.D. theses that we have supervised conjointly with colleagues and the different projects to which we contributed in these three topics to give a visual account of our research spectrum.

We see a computer program as an object shaped by a formal language, the programming language, and by a human activity, software development. Hence, besides the traditional questions about the design of programming languages[2], one general underlying question that directs our research is *how a formal language can capture the dynamic of programming, that is, the interaction between the developer and the source code.*

In our opinion, one of the most important aspects of this interaction is the process through which a developer refines source code to reach a point of equilibrium where the specification is an adequate answer to a problem or a task, and the implementation is correct with respect to this specification. We believe that this refinement should take place in Type Theory, and in particular using the Coq proof assistant because it unifies specification, implementation, and verification in a single tool. By regrouping these three activities in a single place, the programmer indeed gains more control and flexibility on the full development process comparing to the traditional engineering situation where the separation between specification, implementation and verification is enforced.

We are also convinced that this refinement process itself needs formal languages for developers to better understand the metamorphoses of their computer programs through their evolutions or through the different development tools. For example, such a formal language will help in structuring collaborative development to better understand how concurrent patches can be merged safely. It will equally help in minimizing the amount of proofs needed to validate an evolution of a certified piece of software. It could also be used to relate a source program with its compiled code through a representation that helps the programmer to reason about this relation, like the cost annotated source programs produced by CerCo.

If we want to make these new formal languages practical, we will need a new generation of tools to support them. These tools should probably be based on expressive logics like Coq's because they will exploit mechanized semantics but they will also be based on new static analyses to automatically infer simple classes of relational properties between close programs. In any case, these tools will probably not only process source code but also *changes of source code.* This is one of the reasons why we are interested in incremental programming.

To conclude, our global objective is to promote Coq as a general purpose programming language where collaborative software development can be conducted with a high level of confidence to tackle

[2] What is a good syntax and semantics for a programming language to be expressive, safe, usable, etc?

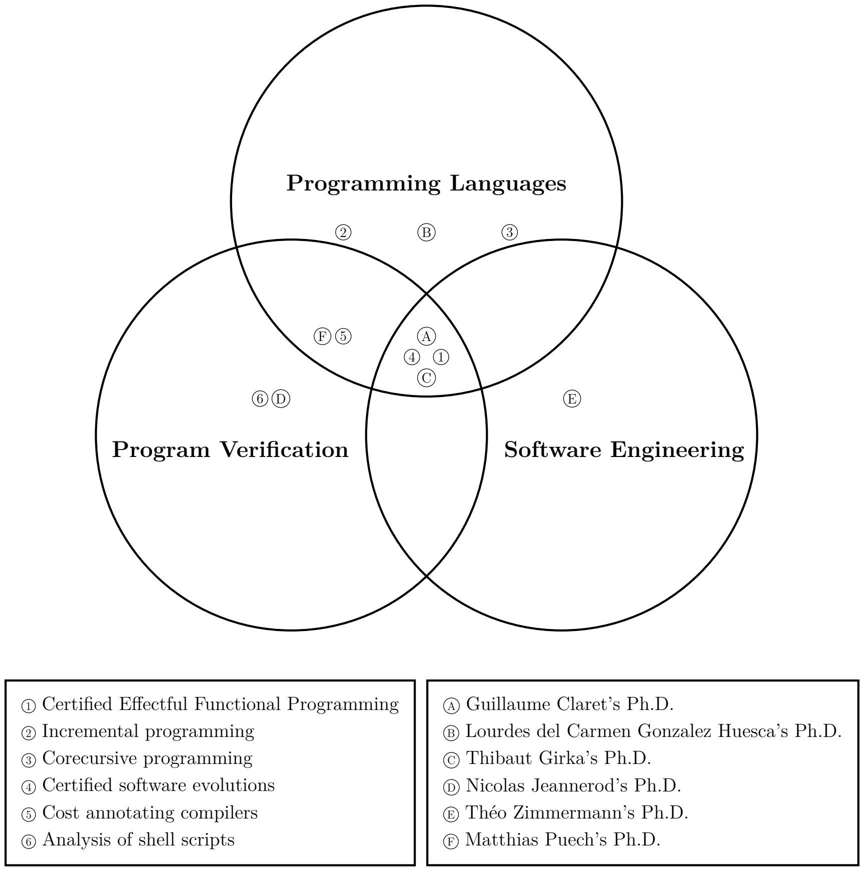

Programming Languages
Program Verification
Software Engineering
② Ⓑ ③
Ⓕ ⑤
Ⓐ
④ ①
Ⓒ
⑥ Ⓓ
Ⓔ
① Certified Effectful Functional Programming
② Incremental programming
③ Corecursive programming
④ Certified software evolutions
⑤ Cost annotating compilers
⑥ Analysis of shell scripts
Ⓐ Guillaume Claret's Ph.D.
Ⓑ Lourdes del Carmen Gonzalez Huesca's Ph.D.
Ⓒ Thibaut Girka's Ph.D.
Ⓓ Nicolas Jeannerod's Ph.D.
Ⓔ Théo Zimmermann's Ph.D.
Ⓕ Matthias Puech's Ph.D.

Figure 1: Research projects and Ph.D. students, visually.

challenging programming tasks, typically software certification or incremental programming. Each chapter of this manuscript will enumerate some intermediate future projects to reach that end.

Research organization

In this section, I briefly summarize how this research work was organized during the past ten years.

Research projects

- Between 2009 and 2013, I participated to the European project CerCo about the certification of compilers.
- Between 2011 and 2014, I participated to the ANR[3] project Paral-ITP about the implementation of proof assistants.
- Between 2013 and 2016, I participated to the ANR project Rapido about infinitary programs and proofs.
- Since 2015, I am a member of the ANR project CoLiS about the verification of Posix shell scripts.

Ph.D. students

- Between 2010 and 2013, I informally supervised the last three years of Matthias Puech's Ph.D. thesis funded by the University of Bologna. This thesis contributed to the topic of incremental programming.
- Between 2011 and 2015, I supervised the Ph.D. thesis of Lourdes del Carmen Gonzalez Huesca about simulable monads and incremental functional programming. This thesis was fund by the Paral-ITP project.
- Between 2012 and 2018, I supervised the Ph.D. thesis of Guillaume Claret about effectful programming in Coq. This thesis was fund by the French ministry of higher education and research.
- Between 2014 and 2018, I co-supervised with David Mentré the Ph.D thesis of Thibaut Girka about semantics differences. This Ph.D. thesis was a CIFRE grant[4] funded by Mitsubishi Electrics.
- Since 2018, I am co-supervising with Ralf Treinen the Ph.D. thesis of Nicolas Jeannerod about the verification of shell scripts. This thesis is funded by the CoLiS project.
- Since 2019, I am co-supervising with Hugo Herbelin the Ph.D. thesis of Théo Zimmermann about "Challenges in the collaborative development of a complex mathematical software and its ecosystem". This thesis is funded by the French ministry of higher education and research.

Philosophy of this document

To conclude this introduction, I want to warn the reader of the somewhat informal nature of the forthcoming chapters. The philosophy of this document is to follow a form which is more argumentative, or explanatory, than technical. As a consequence, some formal definitions are missing, and some statements are a bit simplified for the sake of the presentation. As a matter of fact, this document is more of a reading guide for my papers of the last ten years than a technical reference. I encourage the curious reader with an appetite for technicalities to dive into the papers, or maybe even better, into the Coq proofs when they are available.

There is also some redundancy between this introduction and the contents of each chapter. Indeed, I found it useful to make each chapter as self-contained as possible to make it readable independently from the rest of the document.

[3] ANR is the French agency that funds research grants.

[4] CIFRE grant is the French name for industrial funding.

References

[1] Andreas Abel et al. “Copatterns: programming infinite structures by observations”. In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 27–38.

[2] Andrew Appel et al. *The DeepSpec project*. https://deepspec.org/main.

[3] 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 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. ACM, 2004, pp. 14–25.

[4] Martin Bodin et al. “A trusted mechanised JavaScript specification”. In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 87–100.

[5] Ana Bove, Peter Dybjer, and Ulf Norell. “A Brief Overview of Agda - A Functional Language with Dependent Types”. In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Ed. by Stefan Berghofer et al. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 73–78.

[6] Cristiano Calcagno and Dino Distefano. “Infer: An Automatic Program Verifier for Memory Safety of C Programs”. In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by Mihaela Gheorghiu Bobaru et al. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 459–465.

[7] Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA-Rocquencourt, 2012.

[8] *Debian, the universal operating system*. https://www.debian.org.

[9] Paolo G. Giarrusso. “Optimizing and incrementalizing higher-order collection queries by AST transformation”. PhD thesis. University of Tubingen, 2018.

[10] W. A. Howard. “The Formulæ-as-Types Notion of Construction”. In: *The Curry-Howard Isomorphism*. 1969.

[11] IEEE and The Open Group. *The Open Group Base Specifications Issue 7*. http://pubs.opengroup.org/onlinepubs/9699919799/. 2018.

[12] Ralf Jung et al. “RustBelt: securing the foundations of the rust programming language”. In: *PACMPL* 2.POPL (2018), 66:1–66:34.

[13] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Commun. ACM* 52.7 (2009), pp. 107–115.

[14] Xavier Leroy. *Software: between Mind and Matter*. https://www.college-de-france.fr/site/en-xavier-leroy/inaugural-lecture-2018-11-15-18h00.htm.

[15] Xavier Leroy et al. *The OCaml system release 4.07: Documentation and user’s manual*. Intern report. Inria, 07/2018, pp. 1–752.

[16] Thomas Letan. “Specifying and Verifying Hardware-based Security Enforcement Mechanisms. (Spécifier et vérifier des stratégies d’application de politiques de sécurité s’appuyant sur des mécanismes matériels)”. PhD thesis. CentraleSupélec, Châtenay-Malabry, France, 2018.

[17] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. “Static Caching for Incremental Computation”. In: *ACM Trans. Program. Lang. Syst.* 20.3 (1998), pp. 546–585.

[18] Per Martin-Löf. “An intuitionistic theory of types”. In: *Twenty-five years of constructive type theory (Venice, 1995)*. Ed. by Giovanni Sambin and Jan M. Smith. Vol. 36. Oxford Logic Guides. Oxford University Press, 1998, pp. 127–172.

[19] Eugenio Moggi. “Notions of Computation and Monads”. In: *Inf. Comput.* 93.1 (1991), pp. 55–92.

[20] Nimrod Partush and Eran Yahav. “Abstract Semantic Differencing for Numerical Programs”. English. In: *Static Analysis Symposium*. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 238–258.

[21] Michael Pradel and Koushik Sen. “The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript”. In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Ed. by John Tang Boyland. Vol. 37. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 519–541.

[22] Philip Wadler. “Comprehending Monads”. In: *Mathematical Structures in Computer Science* 2.4 (1992), pp. 461–493.

[23] Xuejun Yang et al. “Finding and understanding bugs in C compilers”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 283–294.

CHAPTER 1

Certified Effectful Functional Programming

> Purity is the power to contemplate defilement.
>
> Simone Weil

Contents

Related publications:

- *Modular verification of programs with effects and effect handlers in Coq.*
  FM (2018) with Thomas Letan, Pierre Chifflier, and Guillaume Hiet.
- *Mtac2: typed tactics for backward reasoning in Coq.*
  ICFP (2018) with Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, and Derek Dreyer.
- *Program in Coq*
  Ph.D thesis of Guillaume Claret (2018)
- *Mechanical verification of interactive programs specified by use cases.*
  FormaliSE (2015) with Guillaume Claret.
- *Incrementality and effect simulation in the simply-typed λ-calculus*
  Ph.D thesis of Lourdes del Carmen González Huesca (Chapters 2–3) (2015)
- *Lightweight proof by reflection using a posteriori simulation of effectful computation.*
  ITP (2013) with Guillaume Claret, Lourdes Del Carmen González-Huesca, and Beta Ziliani.

1 Introduction

Context Gallina, the programming language of Coq, is pure: every program written in Gallina acts as a total function. Unfortunately, the world is impure: you cannot eat the same cake twice, you cannot undo missile launches. From a software development perspective, this means that a program sometimes interacts with an environment that has a state, "remembers" the previous actions, and can respond differently to two subsequent and seemingly similar requests. Can we cope with this impurity within the program-as-total-function paradigm?

As shown by Moggi [38] and Wadler [46], there is a trick to write effectful programs in a purely functional setting : simply represent impure programs using monads and give them their "destructive" semantics later, by external means, typically when the compiler produces the final low-level code.

This technique has been fabulously exploited by Haskell programmers for almost three decades. However, it has some pitfalls: (a) developers regularly blame monads because they do not compose well ; (b) it is usually impossible to escape from the monadic world once you enter it; (c) reasoning about monadic computations is almost always conducted by equational means as there is usually no deductive reasoning principles associated with monadic computations.

The functional programming community has invented several mechanisms to work around problem (a). Monad transformers [31], monad morphisms [34] or monad parameterization [5] techniques give more or less the ability to inject a computation expressed in a monad M into another monad M'. Algebraic effects [7] also tackle this issue by interpreting a possibly variable set of effects thanks to dynamically installed effect handlers, which are, roughly speaking, a generalization of exception handlers with significantly more power over the program control flow. The research results about problems (b) and (c) is less luxuriant.

In Type Theory, Problem (b) amounts to finding a total function of type:

$$\forall A, M\,A \rightarrow A$$

for each monad M, and that is impossible in general! Indeed, a monad can perfectly represent potentially diverging computations and we cannot expected an outcome of type A from a infinite loop.

We could weaken a bit our ambition and look for a total function of type:

$$\forall A, M\,A \rightarrow T_M\,A$$

for some well chosen type constructor T_M. For the monad of potentially diverging computations, $T_M\,A = A + 1$ could seem to solve the issue. Yet, this answer obviously breaks immediately since there is no satisfying function in this type capable of deciding when to give up and return 1.

Contributions Our take on Problem (b) is to *complete* monadic computations with some contextual information C to have them produce their results (if they converge). In other words, we are looking for total functions of the form:

$$\forall A, C \rightarrow M\,A \rightarrow A$$

and we found out that for many monads for which the previous types were not adapted, this idea of completing monadic computations worked. We explore several instances of this idea in work we describe in Sections 2, 4 and 5.

Problem (c) benefits from answers to Problem (b): if we can reinterpret a monadic computation in Type Theory, we can recover proof-by-computation and complete the equational theory associated with the monad under focus. Another ingredient we introduce to solve (c) is the extensive usage of free monads. A free monad is a purely syntactic representation of a monadic computation whose semantics will be defined externally. In other words, an inhabitant of a free monad acts as an abstract syntax tree. By exposing a syntax for monadic computations, we allow the definition of program logics, that is deductive rules of reasoning which follow the syntax of the program.

The Ph.D of Guillaume Claret explores this idea to specify and to reason about a large variety of effects as we will see in Section 4. This line of work is generalized in many directions by the FreeSpec project [29], a collaboration with Thomas Letan [30], that I will present in Section 5. With FreeSpec, we make a big step towards using Coq to develop general purpose software with strong guarantees.

2 Simulable Monads

In this section, we present our work about simulable monads [14] which enables a form of lightweight proof-by-reflection in Coq using *a posteriori* simulation of effectful computations. Let us start with a brief explanation of what proof-by-reflection is.

2.1 Proof-by-reflection in a hurry

Imagine that you are confronted to a Coq goal $\mathcal{G}$ of the following form:

$$\begin{array}{rccc} \mathcal{H}_1 : & A & \sim & A_2 \\ \mathcal{H}_2 : & A_3 & \sim & A_4 \\ & & \vdots & \\ \mathcal{H}_n : & A_n & \sim & A' \\ \hline & A & \sim & A' \end{array}$$

where $\sim$ is an equivalence relation and the A_i are some objects related by $\sim$.

How should we address this goal? Performing this proof by hand requires n applications of the transitivity lemma about $\sim$. This is exhausting and not robust with respect to the future evolutions of this proof. If we invoke instead a decision procedure, we make the computer do the job which is a more sensible strategy. A possible decision procedure for that form of goals could build upon a union-find data structure representing the equivalence classes over the A_i induced by the hypotheses and then check whether A and A' are members of the same equivalence class. In OCaml, this procedure looks like this:

```
let decide_equivalence (hs, (i, j)) : bool =
  iter union hs;
  find i = find j
```

where `union (a, b)` merges the equivalence class of `a` and `b` and `find a` returns the equivalence class of `a`. Here, `hs` and `(i, j)` are the reification of $\mathcal{G}$, a representation that can interpreted back to $\mathcal{G}$ with some function `I`.

Now, there may be an error in that procedure implementation, therefore, to exploit it in Coq, we must first prove its soundness:

$$\text{sound} : \forall r, \texttt{decide\_equivalence}\, r = \texttt{true} \rightarrow \mathrm{I}\, r$$

Then, we can prove a specific goal $\mathcal{G}$ reifed as `g` by checking that the term `sound g (refl_equal true)` is well-typed. To that end, Coq will have to check that the type of `refl_equal true`, that is `true = true`, is convertible to `decide_equivalence g = true` by evaluation of the decision procedure. That form of proof – called proof by reflection – is thus purely computational, it replaces a potentially large proof-term by a computation.

A certified decision procedure written in a total language is a robust tool for the proof developer. However, certifying such procedure is usually expensive in terms of brain power. Besides, certification sometimes leads to over-simplifications that can jeopardize performances hence the usability of the decision procedure.

Another choice consists in implementing the decision procedure in another programming language, typically OCaml in the case of Coq, and to have the resulting decision procedure produce a certificate. This decision procedure cannot be trusted since it has not been certified in Coq. Yet, a certificate verifier can be implemented and proved sound in Coq. On the one hand, developing such a verifier in Coq is usually a feasible task. On the other hand, proof certificate can become very large even for medium-sized problem. Sometimes, they are so large that they cannot fit in memory.

2.2 Monad-based proof-by-reflection

In collaboration with Beta Ziliani, Guillaume Claret, and Lourdes Gonzalez, we introduced simulable monads as a new alternative for decision procedures development.

The first idea of this work is to reduce the cost of defining a decision procedure in Coq by exploiting a partiality monad. This monad provides facilities like fixpoints for general recursion, exception

```
Definition is_equivalent (known_eqs : equalities) (i j : T) : M (i ~ j).
  refine (
    let! r := tmp_ref s 0 nil in
    do! List.iter (unify r) known_eqs in
    let! '(i', Hii') := find r i in
    let! '(j', Hjj') := find r j in
    if eq_dec i' j' then ret _ else error "The terms are not equivalent."
  ).
  apply (equiv_trans _ _ Hequiv _ i'); trivial.
  apply (equiv_trans _ _ Hequiv _ j').
  replace i' with j'.
  now apply (equiv_refl _ _ Hequiv).
  now apply (equiv_sym _ _ Hequiv).
Defined.
```

Figure 1.1: A decision procedure implemented with a simulable monad.

mechanisms or mutable references. All these mechanisms help in reducing development time and implementing efficient imperative algorithms when efficiency matters. For instance, the decision procedure of our running example defined in Coq is presented in Figure 1.1.

The return type of this function says that the caller will obtain a proof of the equivalence between `i` and `j` *if the computation terminates normally (i.e. without uncaught exceptions)*. This condition allows the developer of this procedure to skip the proof of termination that is usually mandatory for any standard recursive Coq function. The programmer can also replace a complicated proof that some property holds by a mere dynamic check[1]

Obviously, the more dynamic checks the function has, the higher the probability of a bug is. However, the developer can choose to experiment first with its decision procedure and once she is satisfied with the implementation, she can remove the dynamic checks one by one to smoothly reach a full certification of the decision procedure. That possibility is a clear improvement with respect to writing decision procedures in OCaml which is not equipped with certification tools, or in comparison to writing certified decision procedures in non monadic Coq which is a really difficult task.

Guillaume Claret developed in Coq plugin named Cybele [2] which includes a monad with mutable references, an arbitrary (dependently-typed) fixpoint operator and an exception mechanism [12]. Programming using this monad has the same taste as programming in an hypothetical dependently-typed OCaml.

2.3 Effect simulation

There is still a catch: how do we extract a value of type A from a computation of type MA? As explained earlier, there is in general no total function of type $M\,A \rightarrow A$ in Type Theory, and this is especially true in the case of Cybele's monad since it models partial computations.

The second idea of this work comes from the observation that the user of a decision procedure stays in front of his screen because he needs the answer to his problem to continue his proof. In other words, we invoke decision procedures *interactively*. If a decision procedure takes too long to answer, we stop it. If a decision procedure does not find an answer to our problem, we are not surprised: maybe our goal is not provable after all or maybe it does not fall in the domain of the decision procedure. As a consequence of this observation, we can assume that the evaluation of the monadic decision procedure will always terminate, sometimes normally, sometimes *manu militari*. We can therefore do something a bit unexpected: executing it twice!

The first execution is done in an unsafe impure environment providing all the effectful operations of the monad as builtins. Thanks to the extraction mechanism of Coq, an OCaml program can realize this first execution. This execution has two purposes: (i) it implements the decision procedure in an efficient environment to try to find a proof for the goal with as much computational power as possible ;

[1] If the property is decidable, of course.

[2] Cybele is an antic divinity. In “Astérix chez Rahazade”, the bard sings “ Ah je ris de revoir Cybèle en ce miroir… Ah ! je ris…“ (a pun whose understanding is left as an exercise to the reader).

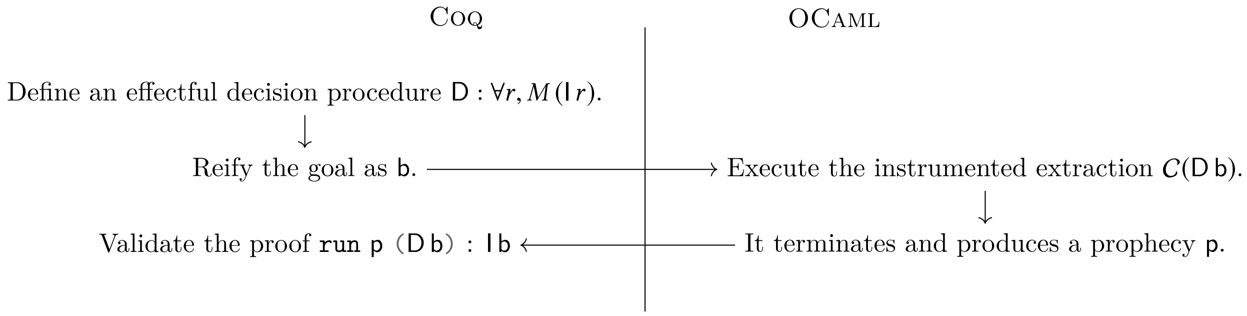

Coq
OCaml
Define an effectful decision procedure $\mathsf{D} : \forall r, M\,(\mathsf{I}\,r)$.
Reify the goal as b.
Execute the instrumented extraction $\mathcal{C}(\mathsf{D}\,\mathsf{b})$.
It terminates and produces a prophecy p.
Validate the proof run p (D b) : I b

Figure 1.2: A posteriori simulation of an effecful computation.

(ii) it serves as an oracle for the second execution of the decision procedure which will take place in the safe environment of Coq.

To share information between these two executions, we instrument the OCaml program to have it produce a so-called *prophecy*. A prophecy is a value that *completes a monadic computation of type $M\,A$ to turn it into a result value of type A knowing that this value exists.* To achieve that, the Coq monad implementation simulates the effects that have been performed by the unsafe execution using a function `run` of type:

$$\mathsf{P}_M \to M\,A \to A$$

where P_M is the type of prophecies for the monad M.

This process that we name "a posteriori simulation of an effectful computation" is picturized in Figure 1.2.

To make this presentation more concrete, let us take the simple example of the "potential-divergence" effect, that is the ability to use general recursion within Coq. What would be a good prophecy for that effect? Cybele's monad indeed provides a general recursion combinator. When we extract this combinator into an OCaml general fixpoint combinator, we instrument this combinator to increment a global loop counter each time a recursive call is done. This way, when the unsafe execution terminates, the counter is an upper bound on the number of recursive calls for all the recursive functions of the program. Hence, by transmitting the value of the counter as a prophecy, we know that all fixpoints of the monadic computation will also converge in Coq.

Let us consider now the "non deterministic choice" effect, that is the ability to choose between two computations non deterministically. As a decision procedure always performs some form of proof search, the programmer often has to express a non deterministic choice between two possible paths to look for the proof. Even though the first execution will generally have to explore both paths in the worst-case scenario (and this unfortunately leads to combinatorial explosion), the second execution can immediately take the right path by extracting it from the prophecy, reducing the complexity of second execution by an order of magnitude.

This last example illustrates the fact that effect simulation is not a mere replay of the OCaml execution in Coq: we should look at the simulation of a monadic computation $M\,A$ as a re-validation of the production of the value of type A found by the untrusted OCaml execution, ignoring the computations that did not contribute to the construction of this value. In other words, the same program acts as a decision procedure in OCaml and as a validator in Coq. One advantage of this approach is to avoid to design a language of proof certificates and to also avoid to program certificate verifier in Coq: this verification is taken care of by the simulable monad.

During his Ph.D, Guillaume Claret explored several optimization techniques to minimize the size of the prophecies. In particular, Cybele includes a notion of monotonic state which allows for cross-stage memoization between the two executions.

Lourdes Gonzalez formalized the theory of simulable monads, resulting in the following formal definition for simulable monads:

Definition 1 (Simulable monad)
A monad M is simulable if there exist

- a type P for prophecies ;
- a total order $\preceq$ over prophecies which measures the amount of “convergence information” they contain ;
- a prophecy $\bot : \mathsf{P}$ which is minimal for $\preceq$ and represents the absence of information ;
- a simplification function $\Downarrow : \forall A, P \to M\,A \to M\,A$ exploiting prophecies to simplify monadic computations ;
- a test to unit function $\mathbb{1}^? : \forall A, M\,A \to \mathbb{B}$ and $\uparrow$ of type $\forall A\,(m : M\,A), \mathbb{1}^? m = \textbf{true} \to A$.

such that:

- $\forall t\,p, \Downarrow_p (\text{unit } t) = \text{unit } t$;
- $\forall m_1\,m_2\,p, \Downarrow_p (\text{bind } m_1\,m_2) = \Downarrow_p (\text{bind } (\Downarrow_p m_1)\,m_2)$;
- $\forall m\,t\,p_1\,p_2, p_1 \preceq p_2, \Downarrow_{p_1} m = \text{unit } t \to \Downarrow_{p_2} m = \text{unit } t$.

Then, we define the instrumented compilation $\mathcal{C}$ of a purely functional language with simulable monads to an impure functional language. We introduce an instrumented big-step operational semantics judgment $E \vdash_{p \to p'} t \Downarrow v$ to account for the fact that a term t in that impure language not only produces a value but also accumulates convergence information in a prophecy initially set to $\bot$. The key requirement to make the simulation work is to demand the effectful constants to be instrumented so that they increase the accumulated prophecy with enough information for their Coq simulation to produce the same result as them. The technical details of this formal development are quite intricate. For this reason, we simply state the soundness theorem and refer the reader to the Ph.D manuscript of González-Huesca [19] for the precise formal development.

Theorem 1 (Soundness of *a posteriori* simulation of effects [19])
Let $\emptyset \vdash m : M\,A$.
If $\emptyset \vdash_{\bot \to p} \mathcal{C}(m) \Downarrow v$ then there exists $a : A$, such that $\uparrow (\text{refl\_eq}\,(\mathbb{1}^?(\Downarrow_p m)))$ is well-typed and equal to a.

3 Monads for typed meta-programming

Mtac1 The Ph.D of Beta Ziliani [49] introduces Mtac1, an extension of Coq to meta-program Coq within Coq itself. Mtac1 solves almost the same problem as simulable monads: it simplifies the development of decision procedures in Coq by extending it with effectful operations and metaprogramming facilities through a monad. Contrary to simulable monads though, the decision procedures written in Mtac1 are not meant to be used in proof-by-reflection but directly as tactics in proof scripts.

Mtac1 improves over Ltac, the traditional language for tactic programming shipped with Coq. Ltac indeed has an informal operational semantics, is untyped, lacks basic support for data structure manipulation, and more generally suffers from design choices that make programming in Ltac an error-prone activity. On the contrary, Mtac1 has a formalized operational semantics and inherits from the rich types of Coq. Developing a decision procedure with Mtac1 is therefore safer than using Ltac.

To finish this succinct presentation of Mtac1, let us consider an example program.

```
Fixpoint inlist A (x : A) (l : list A) : M (In x l) :=
  match l with
  | [] => fail "Not found!"
  | y :: xs =>
    eq <- unify x y;
    match eq with
    | None =>
      r <- inlist A x xs;
      in_cons y x xs r
    | Some eq =>
      ret (in_eq' y x eq xs)
    end
  end.
```

The return type of this function says that if `inlist` terminates normally, it will produce a term of type `In x l`, i.e. a proof that `l` contains `x`. This function is defined by induction on `l`. If the list is empty, the computation stops abnormally with an exception. Otherwise, the program tries to unify `x` with the first element `y` of `l`. If it fails, it continues on the rest `xs` of the list and puts the potential proof that `x` is in `xs` as an hypothesis of the rule `in_cons` of type:

```
forall (A : Type) (a b : A) (l : list A), In b l -> In b (a :: l)
```

If the unification succeeds, it exploits the equality between `x` and `y` to asserts that the searched element is at the beginning of the list. This assertion is done by application of `in_eq` of type:

```
forall (A : Type) (a b : A) (l : list A), a = b -> In b (a :: l)
```

Something is unusual in Mtac in comparison with other metaprogramming languages : there is no syntactic distinction between an object-level term and a meta-level term. In this example, the variable `l` is used as a meta-level value on which we iterate and as an object-level term that is inspected by unification.

This absence of distinction is a smart way to remove the need to reify Coq terms in Coq, saving a lot of implementation energy because most of the Coq machinery (syntax, types, semantics) is reused with absolutely no effort [3]. In addition to that, the user does not have to worry about inserting modalities as in MetaOCaml or LISP to express that he wants to produce a program instead of a value. In some sense, this meta-programming model provides the two representations simultaneously. Finally, Mtac1 offers a high-level pattern-matching on Coq's term that respects scoping and is based on λ-term matching restricted to higher-order patterns [16]. By contrast, the OCaml low-level API provided by Coq represents Coq terms as first-order abstract syntax trees where bindings are represented using de Bruijn indices.

Execution model Mtac programs are written in a free monad defined in Coq. Roughly speaking, a free monad represents a monadic computation by an abstract syntax tree, deferring the actual interpretation of the monadic combinators.

Let us come back to the discussion of this chapter's introduction. Mtac's strategy to complete a monadic computation in order to evaluate it is extreme: in Mtac, the missing contextual information is the actual result of the whole computation! OCaml has therefore the whole responsibility of evaluating the program even if OCaml is considered as an untrusted execution platform. By contrast, the role of Coq is simply to typecheck that the result computed in OCaml has the right type.

This technique might appear dangerous since the result is only validated by a typechecking. Yet, thanks to the dependent types of Coq, the result type of a meta-program can be exactly the proposition we want to prove. Thus, typechecking is as powerful as proof certificate validation. After all, that is exactly what tactics implemented in OCaml do : they produce a candidate proof-term that Coq's kernel validates or rejects.

As we shall see, the types of Mtac operators are very precise contrary to the simple types used by tactic programming in OCaml. By assigning precise types to the operators of the monad, we increase our confidence that the evaluation will terminate normally. This philosophy leads to a form of meta-programming which enjoys a good balance between the guarantees provided thanks to rich types, the flexibility offered by effecful programming and, as discussed earlier, the high level of abstraction of the meta-programming mechanisms.

The Mtac monad contains many primitives, so we simplify its definition for the sake of presentation:

```
Inductive Mtac : Type -> Prop :=
  | ret   : forall {A}, A -> Mtac A
  | bind  : forall {A B}, Mtac A -> (A -> Mtac B) -> Mtac B
  | unify : forall {A} (x y : A) , Mtac (option (x = y))
  | evar  : forall {A}, Mtac A
  | nu    : forall {A B}, (A -> Mtac B) -> Mtac B
  | abs   : forall {A P} (x : A), P x -> Mtac (forall x, P x)
  | fail  : forall {A}, string -> Mtac A.
```

[3] This trick has nonetheless a drawback because it imposes a very specific, and exotic, evaluation strategy on monadic terms. We will come back on that issue in Section 7.

In addition to the standard `ret` and `bind` constructors, this monad provides four operations that capture the essence of meta-programming. `unify` takes two terms and possibly returns a proof that these two terms are equal. `evar A` introduces a unification variable of type `A`. `nu` introduces a fresh name that can be bound to a quantifier using `abs`. `fail` stops the program with an error message. Notice that this definition is already enough to encode the previous example.

Contrary to simulable monads, MTAC programs are not completed with extra information to evaluate the monadic primitives within COQ's standard semantics. Their execution is done in an extended interpreter written in OCAML. Beta Ziliani's interpreter proceeds in two steps to execute a term `t` of type `Mtac A`:

(i) it calls COQ's interpreter to compute the weak-head normal form u ;

(ii) the interpreter performs a case analysis on the constructor of u:

a. If the constructor is `ret`, the interpreter returns the argument of `ret` of type `A`.

b. If the constructor is `bind`, the interpreter is called recursively on both arguments, and one more time on the application of their results.

c. If the constructor is one of the monadic primitives, the interpreter realizes its effectful semantics using dedicated OCAML code, this evaluation results in a new environment and a new term, on which the interpreter is recursively called.

MTAC interpreter has naturally access to the actual abstract syntax trees of source term. As a consequence, it can freely choose to look at them as object-level or meta-level objects when it interprets a monadic primitive. As a matter of fact, this choice is directed by the type of term: roughly speaking, if it is monadic, it is a computational meta-level term ; otherwise, it is an object-level term.

Mtac2 MTAC1 is only meant to program decision procedures called from LTAC proof scripts. Therefore, MTAC1 is not independent from LTAC. With Kate Deplaix, Béatrice Carré and Thomas Refis, we plugged the MTAC1 interpreter in a new dedicated proof mode for COQ. A proof mode is an interpretation environment for tactics that interacts with the proof state of COQ, i.e. a representation of a proof-term with holes which must be completed into a complete proof-term before reaching `Qed`. From the user perspective, the holes are the goals that remain to be proven to complete the proof. From the point of view of COQ's typechecker, the holes are (typed) meta-variables that must be unified to refine the current partial term into a complete proof term.

This promotion of MTAC1 from a meta-programming language to a full tactic language leads to MTAC2, a next-generation tactic language inspired by MTAC1. Proof script programming imposes the introduction of new monadic primitives (especially to interact with the proof state) and thus, new types for these primitives. The design of these type signatures is directed by the following type for tactics inherited from Gordon and Milner's LCF [21, 37, 20]:

```
goal -> goal list * (thm list -> thm)
```

At first glance, this type seems quite reasonable: it denotes the reduction of some proof obligation to a (potentially empty) list of new goals as well as a procedure to later build the actual proof of the goal theorem from the proofs of the generated goals' theorems.

However, as noticed by Asperti et al. [4], this type does not witness an important mechanism at stake when we prove: the effects of a goal resolution on other open goals. Indeed, because unification variables are shared between goals, unification changes the proof state globally. In particular, some goals can be solved by some unification occurring in the resolution of a seemingly unrelated goal. For this reason, it is difficult for the proof script writer to anticipate the goals for which it will actually have to provide a tactic. To tackle this issue and write robust proof scripts, a proof script writer usually separates the goals into two categories: dynamic goals and static goals.

On the one hand, dynamic goals are the goals whose types and numbers are hard to predict : these goals should be automatically discharged. On the other hand, static goals are typically the goals which come from the application of lemmas and for which a significant effort of manual proof is required (because they are morally the difficult arguments of the proof). For this second kind of goals, the proof writer can benefit from a static type system capable of checking that the tactics that solve the static goals are indeed compatible with the types and the number of these goals. That is exactly the purpose of the *typed backward reasoning* mechanism offered by Mtac2.

The type for tactics plays of course a central role in typed backward reasoning. We will now explain this type to communicate the most important ideas of that work. The details about the actual usage of these types are omitted by lack of space. The interested reader will find them in the Mtac2 paper[25]. As said earlier, tactics are meta-programs that interact with the proof state, which is a partial proof-term containing the goals. The goals therefore live in different typing context depending on their scopes. At the point of the proof script where a tactic is applied, there is a goal under focus that we call the *active* goal. The tactic will introduce new goals that may live in different extensions of the typing context of the goal under focus. To make sure that the user properly enters these typing contexts before solving these new goals, the goal constructors wrap meta-variables with information to change the initial typing context of the focused goal into the typing context of this meta-variable.

```
Inductive goal_state := gs_active | gs_any.

Inductive goal : goal_state -> Type :=
  | Metavar: ∀ (gs : goal_state) (A : Type), A -> goal gs
  | AHyp: ∀ {A}, (A -> goal gs_any) -> goal gs_any
  | HypRem: ∀ {A}, A -> goal gs_any -> goal gs_any
  | HypReplace: ∀ {A B},  A -> (A = B) -> goal gs_any -> goal gs_any.
```

When the parameter of this type is `gs_active`, we statically know that a goal's typing context is synchronized with the typing context under the focus of the proof engine. The constructor `Metavar` represents the underlying meta-variable. The constructor `AHyp` extends the context with a new hypothesis of type `A` by putting the goal under a λ-abstraction accepting a value of type `A`. The constructor `HypRem` denotes the explicit deletion of an hypothesis as typically achieved by the `clear` tactic of Coq. The constructor `HypReplace` models the rewriting of a type by an equality proof. The type for "general" tactics is based on this type for goals:

```
Definition gtactic A := goal gs_active -> M (list (A * goal gs_any)).
```

As the traditional type of LCF tactics, this type captures the fact that a tactic reduces a goal into new goals. However, there no more function of type `thm list -> thm` to build the proof of the initial goal from the proofs of the new goals. Indeed, Coq's proof state takes care of this task, not the tactic user. In addition, the parameter of `goal` captures an additional invariant: the input goal must be `gs_active` while the returns goals are not necessarily synchronized with the current typing context. Finally, there is also the possibility for such a tactic to return a value of type `A`. This generalization allows us to turn `gtactic` into a monad and makes them compose smoothly.

To conclude this section, let us comment the definition of `cintro`, a well-typed tactic to introduce hypotheses in the typing context:

```
 1  Definition cintro {A} (var: name) (f: A -> gtactic unit) : gtactic unit := fun g =>
 2  mmatch g with
 3  | [? (P : A -> Type) e] Metavar gs_active (forall x : A => P x) e =>
 4    nu var None (fun x =>
 5      let Px := reduce (RedWhd [RedBeta]) (P x) in
 6      e' <- evar _ Px;
 7      nG <- abs_fun (P:=P) x e';
 8      _ <- exact nG g;
 9      f x (Metavar gs_active Px e') >>= add hyp x)
10  | _ => raise NotAProduct
11  end.
```

The tactic `cintro` takes a user-defined name as input and a function `f` that corresponds to a context in which that name is bound to an implicit type `A`. That function `f` will be used as a continuation for the script. As every tactic, `cintro` takes also a goal `g` as input. On Line 2, the tactic inspects `g` with `mmatch`: the goal must be a product starting with a quantification over `A`. The body of this product is named `P`. With `cintro`, we choose to solve this goal with a proof-term which is a λ-abstraction. For this reason, the tactic introduces a term-level variable `x` of type `A` on Line 4. On Line 5, it computes the type of the λ-abstraction body with a β-reduction on `P x`. On Line 6, `e'` is a new goal which amounts to find an inhabitant for `P x`. On Line 7 and 8, the λ-abstraction is assigned to `g`. On Line 9, We transmit e' to the tactic produced by f x. Finally, we call add hyp to wrap the subgoals generated by f with a constructor AHyp. This operation ensures that the scopes of these goals are correctly related with the current typing context.

4 Monads for certified effectful programming

In this section, we describe the main contributions of Guillaume Claret's Ph.D. about certified effectful programming. Again, we study techniques to complete monadic computations so that they can be specified, executed and certified within Coq. But this time, we tackle this problem through a perspective that is reminiscent of game semantics [35]: can we build an environment – an opponent – that answers for all the licit effects requested by an effectful computation? This point of view gives us a methodology to certify effectful programs. In his thesis, Guillaume Claret confronts this methodology with different kinds of computations: asynchronous, concurrent and blocking.

4.1 Specification of effectful computations by use cases

Let command be a type that enumerates the effects available to a programmer. We also assume the existence of answer : command -> Type, a function that relates each effect to the type of its return value. Then, we introduce the notion of interactive computations returning a value of type A as the inhabitants of the following inductive type:

```
Inductive C A :=
| Ret : A -> C A
| Call : forall (c : command), (answer c -> C A) -> C A.
```

The constructor Ret is the usual unit of the monad. The constructor Call represents a request to the environment for the execution of a command c and the execution of the continuation of type answer c -> C A when the answer to c is available.

A bind operator can easily be defined by induction over its first monadic argument:

```
Definition unit A x :=
  Ret A x.

Fixpoint bind A B (a : C A) (f : A -> C B) : C B :=
  match a with
  | Ret _ x => f x
  | Call _ c h => Call _ c (fun answer => bind A B (h answer) f)
  end.
```

Theorem 2
The type C equipped with unit and bind is a monad.

We now introduce a notion of run of a monadic computation defined by the following inductive type:

```
Inductive R A : C A -> Type :=
| RunRet : forall (x : A), R A (Ret x)
| RunCall : forall c (a : answer c) handler, R A (handler a) -> R A (Call c handler).
```

An inhabitant of this type exactly contains what is necessary to run a given interactive computations as witnessed by the following function:

```
Fixpoint eval A (c : C A) (r : R A c) : A :=
  match r with
  | RunRet _ x => x
  | RunCall _ _ _ _ r => eval _ _ r
  end.
```

In other words, a run is isomorphic to an execution trace of the monadic computation. The specification of an interactive program is a set of such traces that we accept as valid interactions. The question is then: how to define these sets of traces in a specification language that is declarative enough?

Guillaume Claret's point of view about this question is to take inspiration from "use cases", a (semi-)formalism model used in software engineering [15]. He formalizes this notion as follows.

Definition 2 (Use case [13])
A use case over a computation `c` of type `C A` is a run `r` parameterized by some type `P`, i.e. `r : P -> R A c`.

A use case is therefore a form of symbolic trace which is described as a sequence of interactions between the environment and the program. Being parameterized, a single definition can encompass an infinite set of traces that depends on the parameter of type `P`. From that, we can say that use cases are a generalization of unit tests since a unit test also specifies a sequence of interactions but with concrete values, not symbolic ones.

Guillaume Claret demonstrates his approach in the implementation, specification and verification of the high-level behaviors of CoqChick, a blog engine implemented in Coq [11]. He basically shows how to specify an API provided by the blog engine and how to verify that the corresponding Coq implementation is correct with respect to this API. For instance, here is a testing scenario in which an API call is made to retrieve the list of posts:

```
(** The index page when the list of posts is available. *)
Definition index_ok (cookies : Cookies.t) (post_headers : list Post.Header.t)
: Run.t (Main.server Path.Index cookies).
  (* The handler asks the list of available posts. We return `post_headers`. *)
  apply (Call (Command.ListPosts _ ) (Some post_headers)).
  (* The handler terminates without other system calls. *)
  apply (Ret (Response.Index (Cookies.is_logged cookies) post_headers)).
Defined.
```

Guillaume Claret encourages a specification style for runs which is based on proof scripts instead of Gallina specifications. The readability of these scripts is debatable but in Guillaume Claret's opinion, the possibility to execute proof scripts interactively provides a way to understand the specification and the program verification in an intuitive way.

Applying of this methodology on other computational models, Guillaume Claret successively defined a notion of interactive computations and a notion of runs for asynchronous computations, concurrent computations and potentially blocking computations. He refines this methodology along this exploration. For instance, he defines the notions of concurrent computation and concurrent run using the following coinductive types:

```
Record effect := {|
  command : Type;
  answer  : command -> Type
|}.

CoInductive C (E : effect) : Type -> Type :=
| Ret : forall A (e : A), C E A
| Call : forall c, C E (answer E c)
| Let : forall A B (e1 : C E A) (e2 : A -> C E B), C E B
| Join : forall A B (e1 : C E A) (e2 : C E B), C E (A * B).

CoInductive R (E : effect) : forall A, C E A -> A -> Type :=
| RRet : forall A (v : A), R (Ret v) v
| RCall : forall c (a : answer E c), R (Call c) a
| RLet : forall A B (e1 : C E A) v1 (e2 : A -> C E B) v2 ,
  R e1 v1 -> R (e2 v1) v2 -> R (Let e1 e2) v2
| RJoin : forall A B (e1 : C E A) v 1 (e2 : C E B) v2 ,
  R e1 v1 -> R e2 v2 -> R (Join e1 e2 ) (v1, v2).
```

In comparison with the notion of interactive computations presented earlier, the `Call` operator does not hold a continuation anymore. Actually, this change is meant to separate the operation of asking the environment for an effect and the composition of computations. This composition can now be done in two different ways: either sequentially with a `Let` or parallelly with a `Join`.

Another difference is the fact that these two types are now parameterized by a set of `effects` where the previous definitions were based on a fixed set of `commands`. This parameterization allows a finer control of the effects allowed in a given subprogram without compromising its injection in a context that uses more effects. A similar effect composition will be discussed in Section 5.

Finally, the new definitions are coinductive whereas the previous ones were inductive. On the one hand, this choice allows reasoning about non terminating computations. On the other hand, Coq has less facilities for coinduction. Again, this discussion is deferred to Section 5.

4.2 Limitations

The Ph.D. thesis of Guillaume Claret explores many practical aspects of effectful programming in Coq. To our knowledge, this work is the first to concretely use Coq as a general purpose language for applicative development.

However, specification and verification based on use cases suffers from an important drawback: it does not scale well to large system because it does not provide the mechanisms needed for modular specification, implementation and verification. Modular development requires a notion of interface and abstraction specification to respect the principle of information hiding. That is exactly this issue, that FreeSpec tackles.

Another questionable aspect of Guillaume Claret's final approach lies in the choice of a coinductive predicate to represent computations. This choice is similar to the one taken by the interaction trees [27, 47] and we will compare it to our current choice in FreeSpec in the discussion about the related work (Section 5).

5 FreeSpec

Inspired by the Ph.D. thesis of Guillaume Claret and by the `operational` package of Haskell, Thomas Letan's Ph.D. investigates the modularization of implementations, specifications and proofs based on free monads. This work leads to FreeSpec, a framework to develop robust systems based on certified effecful components using Coq. FreeSpec arranges known ingredients of the literature into a minimal and general set of mechanisms relevant to both certified programming-in-the-small and certified programming-in-the-large.

5.1 Certified programming-in-the-small

Syntax In FreeSpec, the effects of a given program are inhabitants of an interface. An interface is a mere type constructor parameterized by the type of effect answers. Just like the Claret's `effects`, interfaces can be composed with disjoint sums.

```
Definition Interface := Type -> Type.

Inductive ComposedInterface (I J: Interface) : Interface :=
| InL (e: I A) : ComposedInterface I J A
| InR (e: J A) : ComposedInterface I J A.
```

Example 1 (Interface for console)
An interface to interact with the terminal console can be defined as follows.

```
Inductive console : interface :=
| PrintLine : string -> Console unit
| ReadLine  : Console string.
```

A program over an interface I inhabits the inductive type `Program` defined as follows.

```
Inductive Program I A :=
| Pure : forall A, A -> Program I A
| Request : forall A B, I A -> (A -> Program I B) -> Program I B.
```

This definition resembles more or less Claret's computations except that it is inductive. FreeSpec refines some technical details like the ability to express bounded quantification over interfaces thanks to the typeclass `Use`. This quantification allows the automatic injection of an effectful program in a context that uses a larger set of effects.

```
Class Use (i ix: Interface) := { lift_eff (a:  Type): i a -> ix a }.
```

Without any surprise, one can prove that `Program I` is a monad. The combination of `Use` and the standard dot-notation of monadic computations allows to write programs following an imperative programming style [43]. The following example shows a typical usage of the `Console` interface to program a very basic interactive program that asks the name of the user and prints it back.

Example 2 (Interaction through the console)
The following program demonstrates the usage of each constructor.

```
(** An example program of basic interaction through the console. *)
Definition hello : Program Console unit :=
  Request ReadLine (fun name =>
  Request (PrintLine ("Hello " ++ name)) (fun _ =>
  Pure tt)).

(** The same program with notations and a generalized interface. *)
Definition hello {ix} `{Use Console ix} : Program ix unit :=
  name <- read_line;
  print_line ("Hello " ++ name).
```

Effect semantics Once again, we want to complete a monadic computation of type `Program I A` to produce a value of type `A`. FreeSpec's answer to this question is imported from the `operational` package cited earlier: we introduce the coinductive type of semantics whose role is to assign some meaning to every effect requests thanks to a (restricted) form of *effect handler*.

```
CoInductive Semantics I : Type :=
| handler (f: forall {A: Type}, I A -> Result I A): Semantics I

with Result I : Type -> Type :=
| mkRes {A}: A -> Semantics I -> Result I A.
```

An inhabitant of `Semantics I` is an handler capable of computing an answer for every effect of `I`. After each effect, the handler not only returns an answer to the effect request but also a new version of the semantics. This modified semantics accounts for the intrinsic statefulness of effects: the answers to two consecutive calls to the same effect can be distinct.

As an illustration, here is how we define a stateful semantics using a state transition function:

```
Definition PS {I : Interface} (State : Type) :=
  forall (A: Type), State -> I A -> (A * State).

CoFixpoint mkSemantics {I : Interface} {State : Type}
  (ps : PS State) (s : State) : Semantics I :=
  handler (fun (A: Type) (e: I A) =>
    mkRes (fst (ps A s e)) (mkSemantics ps (snd (ps A s e))
```

The type of `Semantics` is coinductive for two reasons. First, a semantics is morally a stream of handlers, and streams are more convenient when they are presented coinductively. Second, the semantics is the realization of the effectful environment, and since non termination is a special case of effects, a semantics must be able to conduct an infinite evaluation.

By completing a program with a semantics, we get an evaluation by an induction over the program syntax:

```
Fixpoint runProgram {I : Interface} {A : Type} (sig : Semantics I) (p: Program I A)
: Result I A :=
  match p with
  | Pure a =>
     mkRes a sig
  | Request e f =>
     let res := handle sig e in
     runProgram (next res) (f (result res))
  end.
```

As expected, the evaluation calls the handlers of the semantics to interpret the effects. After that, the evaluation continues with the next version of the semantics.

Representing programs inductively and semantics coinductively is a key design choice of FreeSpec. This choice commits ourselves in a very strict view of purity: the evaluation of a pure (closed) program must terminate and always returns the same value.

```
Fixpoint pipe {ix} `{Use Console.i ix} n : Program ix unit :=
  match n with
  | O => pure tt
  | S k => Console.scan >>= Console.echo;; pipe k
  end.

Axiom ocaml_scan : unit -> string.
Axiom ocaml_echo : string -> unit.

CoFixpoint ocaml_semantics :=
  handler (fun {A} (x : Console.i A) =>
    match x with
    | Console.Scan => mkRes (ocaml_scan tt) ocaml_semantics
    | Console.Echo s => mkRes (ocaml_echo s) ocaml_semantics
    end).

Definition run_pipe : unit := evalProgram ocaml_semantics (pipe 10).

Extraction "pipe.ml" run_pipe.
```

Figure 1.3: An executable semantics for terminal interaction based on OCaml extraction.

Theorem 3 (Pure programs are pure)
Let A be a type and $\bot$ the interface with no effect. The following assertion holds:

$$\forall\,(\mathsf{p} : \mathsf{Program}\,\bot\,\mathsf{A}), \exists(\mathsf{a} : \mathsf{A}), p \simeq \mathsf{Pure}\,\mathsf{a}$$

where $\simeq$ is a bisimilarity over the `Result` type.

There are four distinct ways to define a semantics: (a) by denotation, (b) by extraction, (c) by interpretation, and (d) by delegation. The denotational approach consists in modelizing the semantics as a mathematical function represented in Coq ; this is useful to reason about the program. The definition by extraction uses Coq's ability to produce OCaml programs where some axioms can be realized by manually defined OCaml functions (see Figure 1.3) ; this is useful to give an efficient executable semantics to effects. To avoid the burden of compiling extracted files, FreeSpec also provides a plugin that interprets inhabitants of `Program I` by combining Coq's own interpreter with a extensible effect handler interpreter. The final approach "semantics by delegation" will be discussed in the next subsection.

Logic Now that we have a syntax for our effecful programs and a semantics for their effects, we can reason directly on the evaluation of programs but that reasoning would depend on a specific effect semantics. On the contrary, we would prefer our proof about the program to be valid for any semantics of the effect interfaces. Hence, we need to complete interface declarations with a more informative description, a specification. What kind of specification can we assign to effects? FreeSpec uses a concept of abstract specification defined as follows:

```
Record Specification (W : Type) (I : Interface) : Type :=
{
  (* An interpreter for effects over abstract states. *)
  abstract_step : W -> forall A, I A -> A -> W;

  (* The effects that are compatible with a given abstract state. *)
  allowed_operation : W -> forall A, I A -> Prop;

  (* The effect answers that can arise in a given abstract state.  *)
  expected_answer : W -> forall A, I A -> A -> Prop
}.
```

An inhabitant of `AbstractSpecification W I` puts constraints on a semantics for an interface `I`. We write these constraints as predicates over an abstract state of type `W`. Just like abstract type

```
Inductive Door : Type -> Type := Open : Door unit | Close : Door unit.

Inductive AbstractDoorState := Opened | Closed.

Definition DoorSpecification : Specification AbstractDoorState Door := {|
  abstract_step A e x w :=
    match w, e with
    | Opened, Close => Closed
    | Closed, Open => Opened
    | _, _ => Closed (* Dead code. *)
   end;

  allowed_operation A e w :=
    match w, e with
    | Opened, Close | Closed, Open => True
    | _, _ => False
    end;

  expected_answer A e x w := True
|}.
```

Figure 1.4: An abstract specification for a door interface.

represents the type of a value without exposing its actual implementation, an abstract state represents the state of a semantics without compromising the principle of information hiding. The `abstract_step` function describes how each effect of the interface and its answer transforms the abstract state. This function is executed in lock-step with the semantics' handler gradually refining the static knowledge about the concrete semantics. The predicate `allowed_operation` determines which operations are permitted in a given abstract state: this is a form of precondition parameterized by effects. Finally, the predicate `expected_answer` specifies the possible answers that can be returned by a given effect in a given abstract state. Figure 1.4 contains a simple example specifying the protocol of interaction with a door. In a similar way as for interfaces, specifications can be composed using disjoint sums.

We relate a semantics with an abstract specification through the concept of compliance. To comply with a specification, the semantics of effects should be conformed to their specification in terms of the two predicates `allowed_operation` and `expected_answer`. Besides, this property must be stable by execution.

Definition 3 (Semantics compliance with respect to an abstract specification)
We write $s \models c[w]$ to denote the fact that a semantics s is compliant with a specification c in the abstract state w.

```
Definition conform W I (s : Semantics I) c (w : W) :=
  forall {A: Type} (e: I A),
    allowed_operation c e w -> expected_answer c e (evalEffect s e) w.

Definition stable_by_execution W I s (c : Specification W I) (w : W) P :=
  forall {A: Type} (e: I A),
    allowed_operation c e w -> P c (abstract_step c e (evalEffect s e) w) (execEffect s e).

CoInductive compliant_semantics {W: Type} {I: Interface} s (c: Specification W I) w : Prop :=
| enforced :
    conform W I sig c w ->
    stable_by_exection W I sig c w compliant_semantics ->
    compliant_semantics c w sig.
```

By introducing abstract specifications, we have decomposed the verification problem in two subproblems.

First, a program can be proved correct with respect to any semantics that is compliant with an abstract specification. This proof is often done by induction over the structure of the program. At each reasoning step, the abstract execution refines the abstract state. In addition, the two predicates

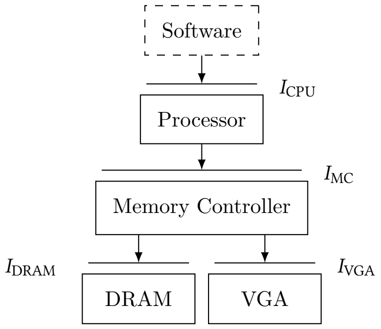


Figure 1.5: A hierarchical system build with FreeSpec "realization by delegation".

define what must be respected by the program to request and what property can be exploited about the effect answers. Thanks to some tactic-based machinery, FreeSpec provides a user experience similar to deductive systems by jumping in the program from one proof obligation to another, skipping all the syntax directed reasoning rules of the program logic.

Second, we must prove that a given semantics complies with an abstract specification. This proof can be done by reasoning over the denotation of the effects or by reasoning about a program that implements these effects. This last idea is the basis for the certification in the large that we present in the next section.

5.2 Certified programming-in-the-large

Component-oriented programming We build complex systems with numerous components connected to each others by interfaces. The mechanisms of the previous section are also relevant to such programming-in-the-large. Actually, a component with internal state `S` exposing an interface `I` and depending on an interface `J` can be seen as a program over the interface `J` that can implement the effects of interface `I`:

```
Definition Component I S J := S -> forall A, I A -> Program J A * S
```

Like interfaces and specifications earlier, we can compose two components that uses the same interface to implement two distinct interfaces into a component that implement both interfaces.

If we are given a semantics for `J`, the evaluation of the component is itself a semantic for the interface `I`:

```
Definition ComponentSemantics {I J S} (sc: Component I S J) (state: S) (s: Semantics J)
: Semantics I :=
  mkSemantics (fun {A: Type} (ss: (S * Semantics J)) (e: I A) =>
    let p := runProgram (snd ss) (sc A e (fst ss)) in
    (fst (result p), (snd (result p), next p)))
  (state, s).
```

This definition of a semantic is what we call definition-by-delegation: in that case, the effect handler used by a component is simply another component! This simple idea allows us to build hierarchical systems with FreeSpec as illustrated by Figure 1.5.

Since components are semantic realizations, their correctness is also expressed as a compliance with respect to an abstract specification. The invariant that implies the correctness of a component generally includes a precise relational predicate to capture the pairs of abstract states of the used interface and abstract states of the exposed interface that can occur during the execution. FreeSpec provides reasoning rules to perform deductive reasoning on components based on this notion of invariant.

Dealing with shared state The software decomposition strategy enforced by FreeSpec is strictly hierarchical: the dependency graph of a software system built with FreeSpec is always a tree. This prevents fine-grained modular decompositions as they require mutual recursion between components [40], and therefore cycles in the dependency graph.

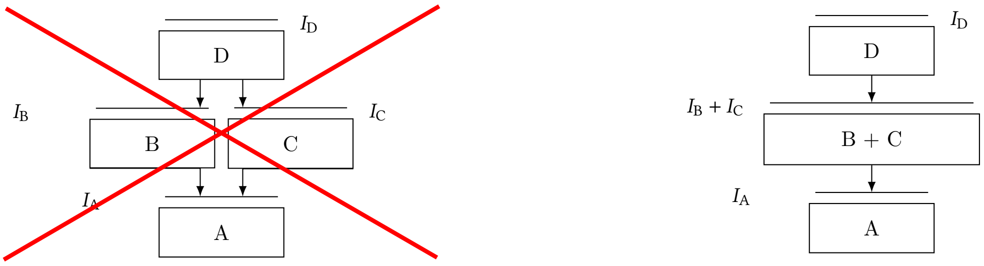


Figure 1.6: Diamond (on the left) are forbidden. Joining components is required (as on the right).

This architectural constraint also forbids distinct components to share a common dependency. Thus, two separated components cannot interact through a shared state. In other words, by default, two separated components have separated states.

We believe that this default separation of the components' states is actually a good non functional property of software systems. Indeed, many verification systems spent a lot of energy to characterize the separation between the fragment of the heap reachable by some component and the fragment that is not, typically by means of separation logic [44]. On the contrary, in FreeSpec, this separation of components' states is free as it is "by construction". This property is reminiscent of the notion of local states encapsulated into objects and closures except that no aliasing of this local state can occur if it is not explicitly demanded by the programmer.

Now, what if we really need to share some state between two components A and B? This is actually possible but this sharing requires the explicit construction of a third component C that joins A and B as in Figure 1.6. A semantics for the interface used by C will define a single state that will be shared by the implementations of the effects of both A and B.

To make this technical explanation more concrete, let us consider how two components can share a simple global counter. A natural interface for this counter would be:

```
Inductive Counter : Type -> Type :=
| Incr : Counter unit
| Read : Counter nat.
```

and this interface admits the following stateful semantics:

```
Definition counter_semantics : Sem.t Counter :=
  mkSemantics (fun A (s : nat) (op : Counter A) =>
    match op with
    | Incr => (tt, s + 1)
    | Read => (s, s)
    end) 0.
```

If `A` has type `Component IA StateA Counter` and `B` has type `Component IB StateB Counter`, then, to share a global counter between `A` and `B`, it suffices to first introduce a component `AB` equals to `A <+> B` and then to give a semantics to `AB` based on `counter_semantics`.

Reasoning about the semantics of `AB` will force to consider the interaction between `A` and `B` through the global counter since between each operations of `AB`, any operation of `A` or `B` could have an effect on the counter value.

Of course, the form of state sharing shown in this example seems restricted comparing to the sharing of dynamically allocated states. However, since components and semantics are first-class citizens in FreeSpec, there is *a priori* no limitation to also implement dynamic allocations of states and to share these states using the same idea. Confirming this intuition is left as future work.

6 Related work

Verification of large systems FreeSpec's concept of abstract specification takes its root into the seminal work of Parnas [41] in which the author states that

> "The main goal is to provide specifications sufficiently precise and complete that other pieces of software can be written to interact with the piece specified without additional information. The secondary goal is to include in the specification no more information than necessary to meet the first goal".

While Parnas [41] considers a module as "a device with a set of switch inputs and readout indicators", we extend the interaction with a module to be based on any type, including higher-order types, like functions. Moreover, FreeSpec's components are first-class while Parnas [41] are second-class: thus, more composition patterns are possible within FreeSpec. Parnas [41] "do not insist that machine testing be done, only that it could conceivably be done": FreeSpec (and other similar systems) shows that machine testing of interface specification can now be done, thanks to the advent of proof assistants.

Contract-based software development[36] or verification[32, 18, 6] introduced many reasoning mechanisms to verify object-oriented systems, especially to enforce as much proof reuse as possible along class hierarchies. Dealing with object-oriented programming mechanisms leads to the introduction of concepts – like contravariant subtyping relations or ownerships – which are hard to grasp for software engineers, the aim of FreeSpec is to avoid as much of this complexity by avoiding late binding, indirect recursions or implicitly shared states. On the contrary, as already argued in Subsection 5.2, the "default" computational mechanisms offered by Coq and FreeSpec simpler to verify than the ones of imperative settings, typically like the object-oriented systems. From that perspective, FreeSpec resembles the B-method [2] and it is also designed to enforce a refinement-based method of verification. Finally, FreeSpec also share this idea of restricting computational mechanisms with FoCaLiZe [42], a proof environment where proofs are attached to components and where programs are "functional programs with some object oriented features".

Kami [10] shares many concepts with FreeSpec, but implements them in a totally different manner: components are defined as labelled transition systems and can be extracted into FPGA bitstreams. Kami is hardware-specific, thus is not suitable to reason about systems which also include software components. However, it allows for expressing more composition pattern than FreeSpec (e.g. components cycle).

Heyman, Scandariato, and Joosen [22] have proposed a component-based modeling technique for Alloy [24], where components are primarily defined as semantics for a set of operations; a component is connected to another when it leverages its operations. Alloy exploits a model finder to verify a composition of these components against known security patterns, and to assume or verify facts about operations semantics; however, it lacks an extraction mechanism, which makes it unable to validate the model experimentally.

Representation of effecful computations in type theory The DeepSpec project recently introduced a new representation of effecful programs in Coq with the so-called *interaction trees* [48, 28]. This approach shares a lot of similarities with ours as both are variants of the free monad. However, interaction trees represent effectful computations through a *coinductive* datatype while, in contrast, Programs are defined inductively. Hence, interaction trees denotes potentially diverging computations while Programs always converge. Interaction trees are therefore expressive enough to denote general recursion and mutual recursion between effectful computations. This expressiveness comes at a cost: interaction trees do not compute inside Coq which makes reasoning about them less convenient than `Program` in our opinion. By contrast, `Program`s can indeed compute inside Coq since their realization is a mere inductive function. In addition, FreeSpec can express diverging computations too but non termination is seen as an effect which is witnessed by the type of `Program`s and implemented by a specific semantics. Contrary to the library of interaction trees which provides many reasoning principles to work with non terminating computations, FreeSpec has no specific support for such reasoning on infinite computations. This is left as future work.

Dylus, Christiansen, and Teegen [17] investigates the problem of reasoning about effectful computations in Coq where effects are modeled using monads. They fix a previous proposal by Abel et al. [1] as it was incompatible with the strict positivity restriction needed for the wellfoundness of inductive definitions. Like us, their final representation for effectful computations is based on a free monad but the parameterization of this free monad is slightly different. Indeed, instead of parameterizing the free monad with a type family of type `Type -> Type` as in FreeSpec, they use a container [45] whose role is to abstract away a functor. This functor can then contain several occurrences of the free monad type constructor without conflicting with the strict positivity checker of Coq. Intuitively, this general form of free monad seems more general than FreeSpec's because of its parameterization with respect to a container. A formal comparison still has to be conducted.

Algebraic effects and effect handlers led to a lot of research about verification of programs with side effects [7, 9], but to our surprise, we did not find any approach to write and verify programs with effects and effect handlers written for Gallina. However, other approaches exist. Ynot [39] is a framework for the Coq proof assistant to write, reason with and extract Gallina programs with side effects. Ynot side effects are specified in terms of Hoare preconditions and postconditions parameterized by the program heap, and does not dissociate the definition of an effect and properties over its realization. To that extent, FreeSpec abstract specification is more expressive (thanks to the abstract state) and flexible (we can define more than one abstract specification for a given interface).

Previous approaches from the Haskell community to model programs with effects using free monads [3, 26, 23] are the main source of inspiration for FreeSpec.

7 Conclusion and Future Work

With the first experiments of monadic programming in Coq through the Ph.D thesis of Guillaume Claret and now with the more ambitious, general and comprehensive approach of Thomas Letan's FreeSpec, we are more and more confident that Coq will become a general purpose programming language. To push further in that direction, we have three new projects: developing a standard library for applicative development in Coq, extending the CertiCoq [8] compiler to FreeSpec's programs, and importing the ideas of simulable monads into FreeSpec. On a more general perspective, I recently started to supervise Théo Zimmermann's Ph.D. thesis about the software engineering aspects of Coq development and of its ecosystem to better understand how we could help its user community to grow in the near future.

A standard library to develop applications in Coq For his master thesis, Vincent Tournier is studying the implementation, specification and implementation of the UNIX command line `ar` using FreeSpec. The purpose of this case study is to confront FreeSpec with the certified development of a small-sized application that interacts with an operating system. This program is based on an interface for file manipulation that includes the standard UNIX system calls (`open`, `close`, `read`, `write`, `seek`, ...). To show that `ar` correctly creates and extracts file archives, we design an abstract specification that accounts for the effects of these system calls on a model of POSIX file systems.

It is challenging to understand if this abstract specification correctly models POSIX. Since POSIX is only informally specified and extremely large, it is probably even impossible to solve this problem. To some extent, such a specification does not have to handle all the details of POSIX. What is essential though is to build specifications that are readable and auditable by experts. As FreeSpec allows various specifications for the same interface, we can even imagine an ecosystem of specifications for POSIX, with different degrees of abstraction and realism. This way, a programmer will be able to choose a specific set of assumptions on top of which he can build its application. His choice depends on the energy he wants to spend on the verification.

A certified compiler for FreeSpec's programs For the moment, FreeSpec programs are either extracted to OCaml, or interpreted directly using our plugin `FreeSpec.Exec`. This situation is convenient to experiment but it is not satisfying in the long run. First, the current trusted base of FreeSpec includes the OCaml runtime, the extraction and the interpreter of `FreeSpec.Exec`. This is quite a large amount of code to trust, especially if we want FreeSpec's certified software to run in safe-critical environments. Second, the interpretation layer to evaluate the monadic combinators and to convey the effect constructors to the handlers is a source of inefficiency. This layer seems to be erasable by a dedicated compiler using aggressive inlining of the combinators' definitions and of the handers' definitions.

CertiCoq is a certified compiler from Gallina to the C language developed at the University of Princeton. Composing CertiCoq with CompCert greatly reduces the trusted base to the sole Coq kernel, and the language specifications. We plan to extend CertiCoq with specific optimizations to obliterate the interpretation layer due to free monads and we also want to introduce a specific FFI to realize effect handlers with C code.

From simulable monads to simulable semantics As a follow-up of our work about simulable monads, we plan to investigate about how a FreeSpec semantics can be completed with information coming from an oracle to revalidate its execution. This revalidation would allow us to certify the execution of a system containing untrusted components.

References

[1] Andreas Abel et al. “Verifying haskell programs using constructive type theory”. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2005, Tallinn, Estonia, September 30, 2005.* Ed. by Daan Leijen. ACM, 2005, pp. 62–73.

[2] Jean-Raymond Abrial and Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 2005.

[3] Heinrich Apfelmus. *The `operational` package.* https://hackage.haskell.org/package/operational. 2010.

[4] Andrea Asperti et al. “A new type for tactics”. In: ().

[5] Robert Atkey. “Parameterised notions of computation”. In: *J. Funct. Program.* 19.3-4 (2009), pp. 335–376.

[6] Mike Barnett et al. “Specification and verification: the Spec# experience”. In: *Commun. ACM* 54.6 (2011), pp. 81–91.

[7] Andrej Bauer and Matija Pretnar. “Programming with Algebraic Effects and Handlers”. In: *Journal of Logical and Algebraic Methods in Programming* 84.1 (2015), pp. 108–123.

[8] Olivier Savary Bélanger and Andrew W. Appel. “Shrink fast correctly!” In: *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017.* Ed. by Wim Vanhoof and Brigitte Pientka. ACM, 2017, pp. 49–60.

[9] Edwin Brady. “Resource-dependent algebraic effects”. In: *International Symposium on Trends in Functional Programming.* Springer. 2014, pp. 18–33.

[10] Joonwon Choi et al. “Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification”. In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017), p. 24.

[11] Guillaume Claret. *CoqChick, a blog engine developed with Coq.* https://github.com/clarus/coq-chick-blog.

[12] Guillaume Claret and Yann Régis-Gianas. *Cybele plugin.* https://github.com/clarus/cybele.

[13] Guillaume Claret and Yann Régis-Gianas. “Mechanical Verification of Interactive Programs Specified by Use Cases”. In: *3rd IEEE/ACM FME Workshop on Formal Methods in Software Engineering, FormaliSE 2015, Florence, Italy, May 18, 2015.* Ed. by Stefania Gnesi and Nico Plat. IEEE Computer Society, 2015, pp. 61–67.

[14] Guillaume Claret et al. “Lightweight Proof by Reflection Using a Posteriori Simulation of Effectful Computation”. In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings.* Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 67–83.

[15] Brian Dobing and Jeffrey Parsons. “Understanding the Role of Use Cases in UML: A Review and Research Agenda”. In: *J. Database Manag.* 11.4 (2000), pp. 28–36.

[16] Gilles Dowek. “Higher-Order Unification and Matching”. In: *Handbook of Automated Reasoning (in 2 volumes).* Ed. by John Alan Robinson and Andrei Voronkov. Elsevier and MIT Press, 2001, pp. 1009–1062.

[17] Sandra Dylus, Jan Christiansen, and Finn Teegen. “One Monad to Prove Them All”. In: *Programming Journal* 3.3 (2019), p. 8.

[18] Cormac Flanagan et al. “PLDI 2002: Extended static checking for Java”. In: *SIGPLAN Notices* 48.4S (2013), pp. 22–33.

[19] Lourdes Del Carmen González-Huesca. “Incrementality and effect simulation in the simply typed lambda calculus. (Incrémentalité et simulation d’effets dans le lambda calcul simplement typé)”. PhD thesis. Paris Diderot University, France, 2015.

[20] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Vol. 78. Lecture Notes in Computer Science. Springer, 1979.

[21] Michael J. C. Gordon et al. “A Metalanguage for Interactive Proof in LCF”. In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski. ACM Press, 1978, pp. 119–130.

[22] Thomas Heyman, Riccardo Scandariato, and Wouter Joosen. “Reusable Formal Models for Secure Software Architectures”. In: *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA 2012, Helsinki, Finland, August 20-24, 2012*. 2012, pp. 41–50.

[23] Ralf Hinze and Janis Voigtländer, eds. *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*. Vol. 9129. Lecture Notes in Computer Science. Springer, 2015.

[24] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT press, 2012.

[25] Jan-Oliver Kaiser et al. “Mtac2: typed tactics for backward reasoning in Coq”. In: *PACMPL* 2.ICFP (2018), 78:1–78:31.

[26] Oleg Kiselyov and Hiromi Ishii. “Freer Monads, More Extensible Effects”. In: *ACM SIGPLAN Notices*. Vol. 50. 12. ACM. 2015, pp. 94–105.

[27] Nicolas Koh et al. “From C to interaction trees: specifying, verifying, and testing a networked server”. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. Ed. by Assia Mahboubi and Magnus O. Myreen. ACM, 2019, pp. 234–248.

[28] Nicolas Koh et al. “From C to interaction trees: specifying, verifying, and testing a networked server”. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. Ed. by Assia Mahboubi and Magnus O. Myreen. ACM, 2019, pp. 234–248.

[29] Thomas Letan. *FreeSpec: a Compositional Reasoning Framework for the Coq Theorem Prover*. https://github.com/lthms/speccert. 2018.

[30] Thomas Letan. “Specifying and Verifying Hardware-based Security Enforcement Mechanisms. (Spécifier et vérifier des stratégies d’application de politiques de sécurité s’appuyant sur des mécanismes matériels)”. PhD thesis. CentraleSupélec, Châtenay-Malabry, France, 2018.

[31] Sheng Liang, Paul Hudak, and Mark P. Jones. “Monad Transformers and Modular Interpreters”. In: *Conference Record of POPL’95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 333–343.

[32] Barbara Liskov and Jeannette M. Wing. “A Behavioral Notion of Subtyping”. In: *ACM Trans. Program. Lang. Syst.* 16.6 (1994), pp. 1811–1841.

[33] Assia Mahboubi and Magnus O. Myreen, eds. *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. ACM, 2019.

[34] Kenji Maillard et al. “Dijkstra Monads for All”. In: *CoRR* abs/1903.01237 (2019). arXiv: 1903.01237.

[35] Paul-André Melliès et al., eds. *Game Semantics and Program Verification, 20.06. - 25.06.2010.* Vol. 10252. Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2010.

[36] Bertrand Meyer. "Applying "Design by Contract"". In: *IEEE Computer* 25.10 (1992), pp. 40–51.

[37] Robin Milner. "LCF: A Way of Doing Proofs with a Machine". In: *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979.* Ed. by Jirí Becvár. Vol. 74. Lecture Notes in Computer Science. Springer, 1979, pp. 146–159.

[38] Eugenio Moggi. "Notions of Computation and Monads". In: *Inf. Comput.* 93.1 (1991), pp. 55–92.

[39] Aleksandar Nanevski et al. "Ynot: Dependent Types for Imperative Programs". In: *ACM Sigplan Notices.* Vol. 43. 9. ACM. 2008, pp. 229–240.

[40] Martin Odersky and Matthias Zenger. "Scalable component abstractions". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA.* Ed. by Ralph E. Johnson and Richard P. Gabriel. ACM, 2005, pp. 41–57.

[41] David Lorge Parnas. "A Technique for Software Module Specification with Examples (Reprint)". In: *Commun. ACM* 26.1 (1983), pp. 75–78.

[42] François Pessaux. "FoCaLiZe: inside an F-IDE". In: *arXiv preprint arXiv:1404.6607* (2014).

[43] Simon L. Peyton Jones and Philip Wadler. "Imperative Functional Programming". In: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993.* Ed. by Mary S. Van Deusen and Bernard Lang. ACM Press, 1993, pp. 71–84.

[44] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings.* IEEE Computer Society, 2002, pp. 55–74.

[45] Tarmo Uustalu. "Container Combinatorics: Monads and Lax Monoidal Functors". In: *Topics in Theoretical Computer Science - Second IFIP WG 1.8 International Conference, TTCS 2017, Tehran, Iran, September 12-14, 2017, Proceedings.* Ed. by Mohammad Reza Mousavi and Jirí Sgall. Vol. 10608. Lecture Notes in Computer Science. Springer, 2017, pp. 91–105.

[46] Philip Wadler. "Comprehending Monads". In: *Mathematical Structures in Computer Science* 2.4 (1992), pp. 461–493.

[47] Li-yao Xia et al. "Interaction Trees: Representing Recursive and Impure Programs in Coq (Work In Progress)". In: *CoRR* abs/1906.00046 (2019). arXiv: 1906.00046.

[48] Li-yao Xia et al. "Interaction Trees: Representing Recursive and Impure Programs in Coq (Work In Progress)". In: *CoRR* abs/1906.00046 (2019). arXiv: 1906.00046.

[49] Beta Ziliani. "Interactive typed tactic programming in the Coq proof assistant". PhD thesis. Saarland University, 2015.

CHAPTER 2

Incremental programming

> Little by little does the trick.
>
> Aesop

Contents

Related publications:

- *Incremental λ-calculus in cache-transfer style, static memoization by program transformation.*
  ESOP (2019) with Paolo Giarrusso, and Philipp Schustser

- *Incrementality and effect simulation in the simply-typed λ-calculus*
  Ph.D thesis of Lourdes del Carmen González Huesca (Chapters 4–7) (2015)

- *Certificates for incremental type checking*
  Ph.D thesis of Mathias Puech (Chapters 1–3) (2013)

- *Safe incremental type checking (short paper).*
  TLDI (2012) with Matthias Puech.

- *Towards typed repositories of proofs (short paper).*
  MIPS (2010) with Matthias Puech.

1 Introduction

1.1 What is incremental programming?

Data constantly change. Google engineers make one commit every 2 seconds on the same git repository. On average, Twitter receives approximatively 6000 tweets per second. A self-driving car typically updates 100MB of its state each second. Data fast evolution is also the reason why database systems are often ranked by the amount of concurrent requests they can cope with.

How do we program software systems to handle these changes? We commonly start by claiming that these systems must be *incremental* in the sense that they *must efficiently react to changes*. From that remark which puts a focus on the verb *react* [11], program designers commonly deduce that the program should be modelled as a function that turns a stream of inputs into a stream of outputs, that the program should follow a dataflow architecture and be realized by some form of circuit. This approach works pretty well to implement an embedded controller as in a self-driving car or to deal with a stream of flat and almost-independent data updates as with tweets.

Can we handle Google engineers' commits with a similar dataflow approach? That is not clear. Indeed, these commits are changing source code and source code is very different from a stream of tweets. First, it is a much structured form of data: a tweet is sequence of characters while source code forms a tree of interdependent syntactic objects. As such, linearization of source code does not make much sense. Second, source code processors – like compilers or static analyzers – require sophisticated algorithms which are very difficult to express as circuits, which is a central concept in reactive programming.

Does that mean that the programmer is bound to reinvent "from scratch" an *adhoc* solution when a circuit-based program does not apply? That is the path taken by the field of online optimization [18]. By contrast, the field of incremental programming hopes to offer some general mechanisms, reasoning principles and design guidelines for programmer to write incrementalized functions.

Incremental programming is about implementing change-centric software systems using first-class changes. From a base input and a change to this input, an incremental program computes a change of the base output. More formally, if f is a function of type $A \rightarrow B$ and if there is a type ΔA for changes over values of type A, and a type ΔB for changes over values of type B, an incrementalization $\mathcal{D}(f)$ of f is such that

$$f\,x \oplus \mathcal{D}(f)\,x\,dx = f\,(x \oplus dx)$$

where $\oplus$ is the application of a change to a value and $\mathcal{D}(f)$ has type $A \longrightarrow \Delta A \longrightarrow \Delta B$.

The litterature often names $\mathcal{D}(f)$ as a *derivative* of f[1]. If dx is small comparing to the input x and if the computational cost of a derivative mainly depends on dx, then it is often more efficient to compute $f\,(x \oplus dx)$ through $\mathcal{D}(f)$ instead of using f.

In some sense, incremental programming includes reactive programming. To see why simply take `'a stream` for A and `'a list` for ΔA. Then, a reactive program is a derivative of a function of type `'a stream -> 'b stream`, that is a function that must compute a new finite sequence of outputs of type `'b` from a finite sequence of new inputs of type `'a`. Reactive programmers take advantage of this restriction (and others, like synchronicity) to obtain bounds over responsiveness time and ressource consumptions. There are no such automatic guarantees for incremental programs, but they can be used in more general settings.

1.2 A concrete example

Figure 2.1 sets up a typical incrementalization problem. In that example, we are interested in finding a derivative for a function named `compile` that turns an arithmetic expression into a sequence of instructions for a stack machine.

On Line 18, the type `dexp` describes a rich set of changes that can be made on abstract syntax trees. The function `apply_dexp` gives a semantics to these changes: "`ReplaceEInt y`" turns "`EInt x`" into "`EInt y`", "`EIntToBinOp (op, lhs, rhs)`" turns "`EInt x`" into "`BinOp (op, lhs, rhs)`", etc. Notice that the last case of `apply_dexp` definition captures 10 situations of change applications that make no sense. For instance, applying "`ReplaceEInt y`" to "`BinOp (op, lhs, rhs)`" is impossible because we cannot change an integer literal that does not exist.

[1] The relationship between this notion and the standard derivative of analysis will be discussed in 5.

On Line 45, the type `dtarget` is inhabited by the changes over list of instructions. They are made of a list of more atomic changes that must be applied at a specific position of the list. With `Remove k`, we can remove `k` instructions from the target code. With `Insert l`, we can on the contrary insert a list of instructions `l` in the target code.

Now, here is a challenging incremental programming exercise for the reader: program `dcompile` such that

```
compile (apply_dexp e de) = apply_dtarget (compile e) (dcompile e de)
```

That is a challenging programming exercise, isn't it?

1.3 What is challenging about incremental programming?

Incrementalization is a difficult programming task needing support from the programming language. In this section, we explain why.

Derivatives are often partial functions. A change can be incompatible with a base value. Indeed, no one can remove an element from an empty list for instance. As a consequence, a natural precondition of a derivative is to only accept an input and a *valid change* for this input. These validity constraints appear everywhere as side-conditions of incremental computations and it is therefore easy for a programmer to forget one of them along a complex sequence of operations.

Since programming derivatives is an error-prone activity, it should therefore be backed up by a type system! As this type system must characterize the valid changes $\Delta(v)$ for each value v, incremental programming seems to require dependent types. While Coq and other implementations of type theory now offer good support for dependently-typed programming, that form of programming is still considered difficult.

Derivatives are defined by a large number of cases. If an inductive type A is defined by n data constructors and the change type ΔA is defined by m cases, then a derivative for a function of domain A will have to consider $n \times m$ cases.

It is easy for a programmer to miss one of these numerous cases. Thus, an incremental programmer needs a checker for exhaustiveness of case analysis. However, determining all the valid pairs such that $(v, dv) \in \Sigma x : v, \Delta(x)$ requires more than a syntactical check, as typically provided by the OCaml pattern-matching analyzer. Again, a dependent inversion principle over the validity relation would help incremental programming.

Efficient derivatives are often program-dependent. There is unfortunately no magic wand that will automatically incrementalize every program into an efficient derivative. Most of the time, an efficient derivative of f will use a specific mathematical property of f, e.g. its associativity, to smartly reuse the intermediate results of the base computation.

Incremental programming is therefore "mathematically-oriented" and that is why an incremental programming language should provide support to track and to express rich properties about functions. Otherwise, an incremental programmer could apply invalid optimizations to its incrementalization or miss an opportunity to get an efficient incrementalization of its program.

Incremental programming is algorithmically challenging. Each base computation must be instrumented to store information that is reusable for its derivatives. It is crucial for a derivative to have an efficient access to this information and to be able to quickly update it.

The data structure that stores this information about the base computation must more or less contain its execution trace but also provide operations to compute the impact of a change in the history of the base computation. Such a datastructure is said to be *retroactive*, as coined by Erik Demaine and its coauthors [8]. Efficient retroactivity is still an open research field in data structure design.

Incremental programming hardly scales to large programs. For primitives like list combinators or index lookup functions which are defined by a dozen lines of code, a programmer can manually write a smart derivative assuming she has a precise understanding of their mathematical properties. But what if we want to incrementalize a program made of millions of lines of code? It is unlikely that

```
(** Abstract syntax trees for arithmetic expressions. *)
type exp = EInt of int | EBin of op * exp * exp and op = Add | Mul

(** Instructions of a stack machine. *)
type instr = IPush of int | IAdd | IMul

(** We want a compiler from arithmetic expressions to instructions. *)
type source = exp and target = instr list

(** This compiler is easily defined by induction over arithmetic expressions. *)
let rec compile : source -> target = function
  | EInt d -> [IPush d]
  | EBin (op, lhs, rhs) -> compile lhs @ compile rhs @ [instr_of_op op]

and instr_of_op = function Add -> IAdd | Mul -> IMul

(** A rich set of changes for the abstract syntax trees. *)
type dexp =
    ReplaceEInt   of int
  | ReplaceOp     of op
  | ChangeLeft    of dexp
  | ChangeRight   of dexp
  | LeftInsertOp  of op * exp
  | RightInsertOp of op * exp
  | ProjLeft
  | ProjRight
  | BinOpToEInt   of int
  | EIntToBinOp   of op * exp * exp

(** Here is how some of these changes can be applied to ASTs. *)
let apply_dexp e de =
  match e, de with
  | EInt x, ReplaceEInt y -> EInt y
  | EInt x, EIntToBinOp (op, lhs, rhs) -> EBin (op, lhs, rhs)
  | EBin (b, lhs, rhs), BinOpToEInt x -> EInt x
  | EBin (b, lhs, rhs), ProjLeft -> lhs
  | EBin (b, lhs, rhs), ProjRight -> rhs
  | EBin (b, lhs, rhs), ReplaceOp b' -> EBin (b, lhs, rhs)
  | e, LeftInsertOp (op, lhs) -> EBin (op, lhs, e)
  | e, RightInsertOp (op, rhs) -> EBin (op, e, rhs)
  | _, _ -> failwith "Invalid change"

type dsource = dexp

type dtarget = Changes of (int * dtarget_atomic) list

and dtarget_atomic = Remove of int | Insert of instr list

let rec at n f l =
  if n = 0 then f l else List.(hd l :: at (n - 1) f (tl l))

let rec take n l =
  if n = 0 then l else take (n - 1) (List.tl l)

let apply_dtarget_atomic = function
  | Remove i -> take i
  | Insert is -> List.append is

let rec apply_dtarget instrs (Changes cs) =
  List.fold_left (fun i (pos, da) -> at pos (apply_dtarget_atomic da) i) instrs

let dcompile : source -> dsource -> dtarget = ?
```

Figure 2.1: A toy compiler incrementalization problem.

a development team can manually incrementalize such a large program in one go. Besides, there is no interesting mathematical property to exploit to incrementalize plumbing-code, that is code that composes more primitive software components.

To have incremental programming scale to large programs, incrementalization must be performed automatically when there is no obvious algorithmic opportunities that can be exploited. Importantly, this automatic differentiation must compose manually-written derivatives with no penalty on the performance. To enable separate processing of software units, automatic differentiation should be compositional and hence be applicable to higher-order programs.

The efficiency of incremental programs is difficult to reason about Computations are irregular: a tiny change on the inputs can have a brutal impact on the outputs. For this reason, the complexity of an incremental program generally makes more sense if it is expressed in terms of the difference between the base output and the modified output. In addition to this unusual parameterization of the algorithmic complexity, reasoning about the behavior of a derivative is hard because it implicitly refers to the two execution traces of $f\,x$ and $f\,(x \oplus dx)$.

Automatic differentiation can have a negative effect on performance predictability. Some automatic incrementalization techniques (typically self-adjusting computations, as we shall see in Section 5) introduce a low-level machinery to increase the reuse of intermediate results and to minimize the cost of change propagation. Since these mechanisms are hidden to the programmer, it is difficult to have a precise cost model in mind to predict performances.

1.4 Incremental functional programming

Our project is the design and the implementation of ΔCaml, an ML-like language with a `derive` keyword such that `derive f` is a derivative of `f`. Within this language, a programmer can also define a *change structure* for every type, i.e., one can associate a change type ΔT to every user-defined type T as well as some operations over changes. Change structures will be formally introduced in Section 2 but, roughly speaking, they provide an operation $\oplus$ to apply a change to a base value as well as a `nil` operator to construct the nil change of any value `v`.

Now, how to define `derive f`? As argued in the previous section, incremental programming is error-prone and difficult to scale to large program so the programmer either (i) needs an automatic mechanism to differentiate `f`, or (ii) the programmer needs support from the programming language to manually define correct-by-construction derivatives. To address (i), we have explored two distinct approaches to automatic differentiation, namely dynamic differentiation and static differentiation. To address (ii), our project includes the introduction of a library called ΔCoq to program correct-by-construction derivatives using the Coq proof assistant. For the moment, the problem (i) is tackled through the Ph.D. of Lourdes del Carmen Huesca and a collaboration with Paolo Giarrusso and Philipp Schuster. The problem (ii) is a work-in-progress whose some design choices are discussed in Section 2.

Lourdes del Carmen Huesca's Ph.D. studies a first approach to automatic differentiation. Following ideas imported from the differential λ-calculus of Ehrhard and Regnier [9], we design a λ-calculus with a *dynamic* differentiation operator. In that system, we have:

$$\texttt{derive}(\lambda x.t) = \lambda x\,\lambda dx.\frac{\partial t}{\partial x}$$

where $\frac{\partial \bullet}{\partial x}$ is an operator which dynamically incrementalizes a λ-term t with respect to the change of a variable x.

The expression $\frac{\partial t}{\partial x}$ captures the exact impact on the evaluation of t of a change dx applied to x. As we shall see in Section 3.1, this variant of the differential λ-calculus correctly models the incrementalization process of λ-terms and, as such, provides a rich framework to understand the foundations of incremental programming and to design reasoning principles over incremental computations.

However, the deterministic differential λ-calculus introduced by Gonzalez' Ph.D. suffers from the dynamical aspect of its differentiation: an interpreter for this language needs to introspect a function's body at runtime to differentiate it. This requirement significantly reduces the possibilities to use this calculus as a practical programming language. Indeed, with most compilation techniques, the source code is not available during program execution.

During the same period, Giarrusso and his coauthors [4] introduce the Incremental Lambda Calculus (ILC). This framework shares the same motivation as the deterministic differential λ-calculus of Gonzalez' Ph.D. but it incrementalizes λ-terms *statically*. In other words, Giarrusso's differentiation is a program transformation that turns a λ-term t into another λ-term which is a derivative of t. Even though this program incrementalization is less precise than the partial derivatives of the differential λ-calculus (in the sense that it differentiates with respect to all free variables simultaneously, not just one), ILC's differentiation has the critical advantage of being a mere program transformation which is perfectly compatible with standard compilation techniques. For this reason, we choose that path to implement ΔCaml.

The static differentiation of ILC is unfortunately too naive in practice. As Section 4 will explain, the derivatives generated by Giarrusso's static differentiation recompute values that have already been encountered by the base computation. That lack of communication between functions and their derivatives prevents efficient incrementalization (except on a very specific class of derivatives called *self-maintainable*).

To create a channel of communication between the function and its derivative, we generalize ILC in collaboration with Giarrusso and Schuster [13]. The basic idea, due to Liu, is to have a base computation produce a cache of its intermediate results and to pass this cache as an extra argument to the derivative to avoid recomputation. We generalize this form of static memoization to higher-order programs and integrate it to static differentiation of ILC, now renamed *static differentiation in Cache Transfer Style (CTS).*

Contributions Our contributions to the field of incremental programming are the following:

- In collaboration with Lourdes Gonzalez del Carmen Huesca, we introduce a deterministic variable of the differential λ-calculus meant to incrementalize computations.
- In collaboration with Paolo Giarrusso and Philipp Schuster, we give a new proof of soundness for static differentiation of λ-terms, which does not rely on simple types.
- In collaboration with Paolo Giarrusso and Philipp Schuster, we improve static differentiation of λ-terms with a Cache Transfer Style translation. This translation allows base computations and their derivatives to share computations.

2 Change structures

"Changing a value" often means "replacing a value by another one" but this interpretation is too coarse as it prevents manipulating changes independently of their applications. Change structures give an operational device to define computations over first-class changes.

2.1 Definitions

There are already several variants of change structures (with distinct names) in the literature. Let us briefly review them to explain our design choices in ΔCoq and ΔCaml.

The first definition of change structure we consider was proposed by Giarrusso et al. [4]. This definition requires the existence of dependent type family Δ which denotes the valid changes for every value of some type A. Thanks to this precise type, the change application is a total function. This definition also assumes an operator ⊖ to compute the difference between two values of type A. We call this strong property "completeness of the change structure".

Definition 4 (Complete change structure [4])
A *complete change structure* is a tuple $(A, \Delta, \oplus, \ominus)$ such that:

- A is a type.
- $\Delta : A \to \texttt{Type}$ where for all a of type A, the inhabitants of Δa are the valid changes for a.
- $\oplus : \forall (x : A), \Delta x \to A$ where $a \oplus da$ is the application of the change da to a.

- $\ominus : A \to \forall (x : A), \Delta x$ where $a \oplus (b \ominus a) = b$.

Independently to Giarrusso, Lourdes Gonzalez Huesca introduces a close definition for a notion that we call "displaceable type". In this setting, the type for changes in not dependent on the value it is applied to. As a consequence, change application is a partial function. Another difference with Giarrusso's definition is the requirement for an operator $\odot$ to compose changes as well as for a *nil change* $\mathbf{0}$ to serve as an identity for this composition. Giarrusso also introduces a notion of nil change for value v which must satisfy $v \ominus v = \mathbf{0}\, v$. By contrast, Gonzalez' nil change does not depend on a specific value.

Definition 5 (Displaceable types [15])
A type A is *displaceable* by $(\Delta A, \oplus, \ominus, \mathbf{0}, \odot)$ if

- ΔA is a type for changes.
- $M_\Delta = (\Delta A, \odot, \mathbf{0})$ is a monoid.
- $\oplus : A \times \Delta A \twoheadrightarrow A$ is an action of the monoid M_Δ on A.
- $\ominus : A \to A \to \Delta A$ where $a \oplus (b \ominus a) = b$.

Finally, Mario Alvarez and Luke Ong based their categorical study of change actions on a weaker definition. They do not ask for the change structure to be complete and they only consider *total* change applications.

Definition 6 (Change action [3])
A *change action* is a tuple $(A, \Delta A, \oplus, \odot, \mathbf{0})$ such that:

- ΔA is a type for changes.
- $M_\Delta = (\Delta A, \odot, \mathbf{0})$ is a monoid.
- $\oplus : A \times \Delta A \to A$ is an action of the monoid M_Δ on A.

2.2 Discussion

Which of these definitions is the most convenient for certified incremental programming? The differences between them put into light some technical difficulties that are tackled, or not, by the authors depending on their interests.

Should we consider $\oplus$ as a partial or a total function? While both Alvarez and Giarrusso require the totality of change application, Gonzalez allows it to be partial. Partiality can be dealt with on paper but in proof assistants like Coq which are based on Type Theory, there is no such thing as a partial function. Even though partiality can be represented in Type Theory[28, 2], this would force ΔCoq to be written in a monadic style. To avoid this constraint, we followed Alvarez and Giarrusso by considering change application as a total function in the design of ΔCoq.

Should we assign a simple type or a dependent type to $\oplus$? This question is related to the previous one: as we decide to consider change application as a total function, we cannot stay in a simply typed setting: otherwise, that would rule out the incrementalization of many programs, typically list-processing programs.

Unfortunately, it is also notoriously more difficult to use a dependently-typed signature than a simply typed one, not only because checking that the correct usage of a dependent type may induce proof obligations, but also because dependent types come with technical issues like the need for proof irrelevance. Giarrusso *et al.* already noted the problem and introduced an erasure of dependencies to relate dependently-typed incremental programs to simply-typed ones. This allows them to justify their static differentiation using Type Theory while actually programming in Scala using non dependent types.

We advocate for a slightly different choice. In ΔCoq, change application is dependently-typed even if ΔCoq is actually meant to program derivatives. As argued previously, we believe that manual incrementalization of programs is a difficult programming activity which is a typical case where a rich and expressive type system like Coq's is relevant. As we shall see, our definition of change structure has been tailored to ease dependently-typed programming as much as possible using mechanisms and techniques that the community has offered us recently. In particular, we reuse ideas from *dependent interoperability* [7, 6] to safely interface manually-defined derivatives extracted from ΔCoq to the automatically differentiated programs generated by ΔCaml.

Should we use a complete change structure? While Alvarez *et al.* look for a notion of *differentiability* compatible with change actions that would encompass the standard notion of differentiability of analysis, our goal is more pragmatical in the sense that we want to provide a programming framework for incrementalization. In that setting, it is immediate to turn all change actions into a complete one by introducing:

$$! : \forall x : A, A \to \Delta x$$

such that $x {\oplus} !y = y$. This change accounts for a replacement of a value x of type A with another one y. As noticed by Gonzalez, this replacement change allows a derivative to recompute a value from scratch when no efficient incrementalization is possible, typically when the control flow of the new computation is too far from the base computation's one. In most situations, this shortcut change gives an efficient implementation for $\ominus$ as $y \ominus x = !y$.

Is a change composition operator useful? Giarrusso *et al.* do not include a composition operator in their definitions. As the PhD of Gonzalez shown that the expression of partial derivatives require such composition operator, ΔCoq also requires the ability to compose changes. More generally, we believe that the existence of this operator is consistent with the idea of promoting changes as first class citizens.

2.3 Rich change structures

Our design choices eventually lead to a notion of change structure which is far from being minimal.

Definition 7 (Rich change structure [4])
A *rich change structure* is a tuple $(A, \Delta A, \mathcal{V}, \oplus, \odot, \mathbf{0}, \ominus)$ such that:

- A is a type and ΔA is a type for change.
- $\mathcal{V} : A \to \Delta A \to \texttt{Prop}$ is a validity predicate for change. It must be decidable.
- $\Delta : A \to \texttt{Type}$ is defined as a $\texttt{Prop}$ irrelevant subset type $\Delta x \triangleq \{dx : A \mid \text{decide } \mathcal{V}\, x\, dx = \mathbf{true}\}$
- $\oplus : \forall(x : A), \Delta x \to A$ where $a \oplus da$ is the application of the change da to a.
- $\odot : \forall(x : A)(dx : \Delta x) \to \Delta(x \oplus dx) \to \Delta x$ is an associative change composition operator.
- $\mathbf{0} : \forall(x : A), \Delta x$ is such that $\forall x, x \oplus \mathbf{0}\, x = x$ and is an identity for $\odot$.
- $\ominus : A \to \forall(x : A), \Delta x$ where $a \oplus (b \ominus a) = b$.

This definition contains several novelties comparing to the previous ones. First, we introduce a decidable predicate $\mathcal{V}$ for change validity. The decidability of this predicate is especially relevant to have the simply typed ΔCaml code interoperate safely with the code extracted from ΔCoq. Indeed, it is immediate to provide the following two conversion functions:

$$\begin{array}{lcl} \text{from\_rich\_change} & : & \forall x, \Delta x \to \Delta A \\ \text{to\_rich\_change} & : & \forall x, \Delta A \nrightarrow \Delta x \end{array}$$

and to use them at the interface between user-defined ΔCaml code and some derivatives proved and extracted from Coq. That way, the only possible point of failure is a misuse of a certified derivative by passing an invalid change to it. This problem is detected at the call site of the derivative, not through an erratic behavior of this derivative coming from a violation of its precondition.

Second, the dependent type Δx denoting the valid changes of some value x is defined using a subset type whose proposition is proof irrelevant. This concept has been introduced recently in Coq 8.10 as a by-product Gaëtan Gilbert[14]'s Ph.D. It basically ensures that two changes are convertible if their underlying representations of type ΔA are convertible, regardless of their respective proofs of validity with respect to x. Without proof irrelevance, it is really challenging to typecheck terms computing over subset types. Indeed, two changes computationally compatible can be separated by the typechecker only because their proofs of validity are not convertible, which is hard to control for the user. Of course, there remains convertibility issues, typically when one has to convert a change for x into a change y when x are only provably equivalent to y. Yet, that kind of conversion is manageable and actually relevant to better understand the incremental program logic.

2.4 Change structures over usual types

In this section, we give several examples of change structures with a particular emphasis on the design of a change structure for functions.

Example 3 (Naturals)
Any natural number n can be changed using a integer k by a mere addition. To make sense, this addition must not remove more than n units from n. Therefore, the validity predicate $\mathcal{V}\,n\,k$ is $(k < 0) \rightarrow (-k < n)$. We also define $\mathbf{0}\,n = 0_{\mathbb{Z}}$ and make use of the addition of integers to compose changes. Finally, the substraction over integers works perfectly to compute the change between two natural numbers.

Example 4 (Products)
If $(A, \Delta A, \mathcal{V}_A, \oplus_A, \odot_A, \mathbf{0}_A, \ominus_A)$ and $(B, \Delta B, \mathcal{V}_B, \oplus_B, \odot_B, \mathbf{0}_B, \ominus_B)$ are two change structures, then, by lifting the two set of operations to products, $(A \times B, \Delta A \times \Delta B, \mathcal{V}_{A\times B}, \oplus_{A\times B}, \odot_{A\times B}, \mathbf{0}_{A\times B}, \ominus_{A\times B})$ is also a change structure.

Example 5 (Sums)
Let $(A, \Delta A, \mathcal{V}_A, \oplus_A, \odot_A, \mathbf{0}_A, \ominus_A)$ and $(B, \Delta B, \mathcal{V}_B, \oplus_B, \odot_B, \mathbf{0}_B, \ominus_B)$ be two change structures. We use $\Delta A + \Delta B + A + B$ as a type for changes. The first (resp. second) case of change denotes a stable change from $\mathbf{in}_1\,a$ (resp. $\mathbf{in}_2\,b$) to another value of the form $\mathbf{in}_1\,a'$ (resp. $\mathbf{in}_2\,b'$) by a change da (resp. db). The third (resp. fourth) case of change denotes a complete replacement of $\mathbf{in}_1\,a$ (resp. $\mathbf{in}_2\,b$) by another value of the form $\mathbf{in}_2\,b$ (resp. $\mathbf{in}_1\,a$). Therefore, $\mathcal{V}_{A+B}\,s\,ds$ is

$$\begin{array}{rcl}(\exists\, a\,da, s = \mathbf{in}_1\,a \wedge ds = \mathbf{in}_1\,da) & \vee & (\exists\, b\,db, s = \mathbf{in}_2\,b \wedge ds = \mathbf{in}_2\,db)\ \vee \\ (\exists\, a', ds = \mathbf{in}_3\,a') & \vee & (\exists\, b', ds = \mathbf{in}_4\,b')\end{array}$$

The definitions of the operations over changes are left as an exercise to the reader. We have that $(A + B, \Delta A + \Delta B + A + B, \mathcal{V}_{A+B}, \oplus_{A+B}, \odot_{A+B}, \mathbf{0}_{A+B}, \ominus_{A+B})$ is change structure.

Example 6 (Lists)
If $(A, \Delta A, \mathcal{V}_A, \oplus_A, \odot_A, \mathbf{0}_A, \ominus_A)$ is a change structure, then let us take

$$\Delta\mathrm{list}\,A ::= \mathrm{Insert}_k\ a \mid \mathrm{Remove}_k\ a \mid \mathrm{Update}_k\ a\,da \mid \mathrm{Compose}\ dl\,dl \mid \mathrm{NilChange}$$

where we take $k \in \mathbb{N}$, $a \in A$, $da \in \Delta A$, and $dl \in \Delta\mathrm{list}\,A$.

This change type is made of atomic changes applied at some position k of the list and of composite changes. The validity predicate checks that positions are well-formed and that the changes on values of type A are valid. The operations are straighforward. We have shown that the tuple defined as $(\mathrm{list}\,A, \Delta\mathrm{list}\,A, \mathcal{V}_{\mathrm{list}\,A}, \oplus_{\mathrm{list}\,A}, \odot_{\mathrm{list}\,A}, \mathbf{0}_{\mathrm{list}\,A}, \ominus_{\mathrm{list}\,A})$ is a change structure.

Example 7 (Functions)
Let $(B, \Delta B, \mathcal{V}_B, \oplus_B, \odot_B, \mathbf{0}_B, \ominus_B)$ be a change structure. The type $A \rightarrow \Delta B$ can be used for the changes over $A \rightarrow B$. Change application is then defined as:

$$(f \oplus df)\,x = f\,x \oplus df\,x$$

which constraints $\mathcal{V}\,f\,df$ to enforce $\forall x, \mathcal{V}_B\,(f\,x)\,(df\,x)$. From this, it comes that $\mathbf{0}\,f = \lambda x.\mathbf{0}\,(fx)$. The composition of changes is simply the point-wise application of $\odot_B$ and similarly the substraction between functions resorts to a point-wise application of $\ominus_B$.

Interestingly, this change structure over functions introduced in the Ph.D. thesis of Gonzalez is not the same as Giarrusso's. They instead assume that A is equipped with a change structure for A, i.e., $(A, \Delta A, \mathcal{V}_A, \oplus_A, \odot_A, \mathbf{0}_A, \ominus_A)$ and they use $A \rightarrow \Delta A \rightarrow \Delta B$ as change types for functions. This choice leads to the following definition for change application:

$$(f \oplus df)\,x = f\,x \oplus df\,x\,(\mathbf{0}\,x)$$

and the validity predicate is defined as:

$$\mathcal{V}\, f\, df = \begin{cases} \forall a\, da, \mathcal{V}_A\, a\, da \to \mathcal{V}_B\, (f\, a)\, (df\, a\, da) \land \\ \forall a\, da, f\, a \oplus df\, a\, da = f\, (a \oplus da) \oplus df\, (a \oplus da)\, (\mathbf{0}\, (a \oplus da)) \end{cases}$$

As we shall see, these two design choices is explained by a different treatment of applications between Gonzalez' dynamic differentiation and Giarrusso's static differentation.

3 Differentiation of functional programs

3.1 Dynamic differentiation

The Ph.D. thesis of Lourdes del Carmen Gonzalez Huesca introduces a deterministic differential λ-calculus. Its syntax includes an operator written $\mathcal{D}(\bullet)$ to *dynamically differentiate* λ-terms.

Morally, we could simply define this operator as follows:

$$\mathcal{D}(\lambda x.t) = \lambda x\, dx.t[x \mapsto x \oplus dx] \ominus t$$

because this function is a valid derivative. Indeed, it clearly computes how to change the base output of the function when its formal argument x is changed. However, this derivative is extremely inefficient since it recomputes both the base output, the modified output and the difference between them!

As said earlier, we are instead looking for a differentiation that incrementalizes as efficiently as possible by (i) using manually defined derivatives when they are available or else by (ii) using a generic differentiation mechanism that carefully composes user-defined derivatives. The reduction rules of Gonzalez differential λ-calculus are tailored to deal with that second aspect.

Definition 8 (A deterministic differential λ-calculus)
We equip the standard call-by-value λ-calculus extended with $\mathcal{D}(\bullet)$ ruled by the following reduction:

$$\mathcal{D}(\lambda x.t) \to \lambda x\, dx.\frac{\partial t}{\partial x} \quad \text{where} \quad \begin{cases} \dfrac{\partial y}{\partial x} = \begin{cases} dx & \text{if } y = x \\ \mathbf{0}\, y & \text{otherwise} \end{cases} \\ \dfrac{\partial(\lambda y.t)}{\partial x} = \lambda y.\dfrac{\partial t}{\partial x} & \text{if } x \neq y \\ \dfrac{\partial \mathcal{D}(t)}{\partial x} = \mathcal{D}(\dfrac{\partial t}{\partial x}) \\ \dfrac{\partial(r\, s)}{\partial x} = \left(\mathcal{D}(r)\, s\, \dfrac{\partial s}{\partial x}\right) \odot \left(\dfrac{\partial r}{\partial x}\, (x \oplus \dfrac{\partial s}{\partial x})\right) \end{cases}$$

The rule for variables distinguishes whether the variable is the one on which the differentiation is focusing: indeed, the variation is dx in that case ; and is nil otherwise. The rule for abstractions is consistent with Gonzalez' change structure for functions as it produces a variation which is parameterized by the exact same argument. Since the differentiation of a term t is part of the term language, a rule must therefore tackle its partial differentiation with respect to a free variable of t. Fortunately, $\mathcal{D}(\bullet)$ commutes with partial differentiation. Finally, the most complex rule deals with applications. In that case, the variation is composed of two changes: first, the derivative of the function is applied on the argument and its variation ; second, another change must take the variation of the function itself into account.

Contrary to the naive definition given earlier, this differentiation is defined by recursion over the structure of the λ-term. As a consequence, for some function f, if $\mathcal{D}(f)$ is user-defined, then $\mathcal{D}(\lambda x.f\, x)$ will reduce to:

$$\lambda x\, dx.\frac{\partial(f\, x)}{\partial x} = \lambda x\, dx.\mathcal{D}(f)\, x\, dx \odot \frac{\partial f}{\partial x}\, (x \oplus dx) = \lambda x\, dx.\mathcal{D}(f)\, x\, dx \odot \mathbf{0} f\, (x \oplus dx) = \lambda x\, dx.\mathcal{D}(f)\, x\, dx$$

which will actually call the user-defined derivative of $\mathcal{D}(f)$ as expected.[2]. In other words, this automatic dynamic differentiation fits its purpose: composing derivatives in a higher-order setting. The chain rule is another illustration of this good behavior:

Theorem 4 (Chain rule)
The chain rule holds for the deterministic differential λ-calculus.

$$\mathcal{D}(\lambda x.(f \circ g)\, x) \to \lambda x\, dx.\mathcal{D}(f)\,(g\, x)\,(\mathcal{D}(g)\, x\, dx)$$

Theorem 5 (Soundness of dynamic differentiation[15])
Let f be function. The following equation holds:

$$f\,(x \oplus dx) = f\, x \oplus \mathcal{D}(f)\, x\, dx$$

where the equality stands for the definitional equivalence induced by the operational semantics.

Gonzalez defines a partial derivation for fixpoints, pattern-matching and application of data constructors. These constructions are already sufficient to incrementalize interesting programs and to reason about this incrementalization.

However, as said in the introduction, the fact that differentiation is done dynamically prevents us from implementing a realistic compiler for this programming language since it would require code inspection at runtime, which is hard to set up with traditional techniques. In addition to this practical issue, the fact that $\mathcal{D}(\bullet)$ must sometimes be differentiated imposes the existence of changes for changes, and of course the existence of changes of changes of changes, and so on and so forth. Again, this requirement is not convenient in practice since it forces the programmer to implement infinite families of changes, which is far from obvious.

3.2 Static differentiation

In his Ph.D. thesis [12], Giarrusso studies a program transformation which produces a derivative for any λ-term. Contrary to Gonzalez' technique, this procedure differentiates a λ-term with respect to all its free variables at once. The definition of this transformation is stunningly simple compared to dynamic differentiation:

$$\begin{array}{rcl} \mathcal{D}(x) & = & dx \\ \mathcal{D}(t\, u) & = & \mathcal{D}(t)\, u\, \mathcal{D}(u) \\ \mathcal{D}(\lambda x.t) & = & \lambda x\, dx.\mathcal{D}(t) \end{array}$$

The case for variables illustrates the fact that the differentiation is done with respect to all the free variables. Indeed, the variation of each variable can contribute to the final variation computed by the derivative. The case for abstraction is a natural consequence of the expected shape for a derivative.

The case for applications is the important simplification comparing to dynamic differentation since there is only one term and this term is immediately computed by the static differentation. By contrast, the dynamic differentiation inserts a call to the differentiation operator that expanses to even more complex terms.

Two ingredients make that simplification possible: first, Giarrusso's changes for functions are able to take the variation of the function inputs into account while Gonzalez's changes are not ; second, the term $\mathcal{D}(t)$ where t is a function and contains free variables is more general than the derivative of t since it does not only consider the variation of the inputs but also the variation of all the free variables.

In the initial presentation of this static differentiation, Cai et al. [4] proved the correctness of this static differentiation on the simply-typed λ-calculus in Agda using a shallow-embedding of its denotational semantics, thus restricting themselves to terminating functions.

Theorem 6 (Soundness of static differentiation [4])
If $f : A \to B$, $a : A$ and $da : \Delta A$ is a valid change for a, then the following holds:

$$f\,(a \oplus da) \simeq f\, a \oplus \mathcal{D}(f)\, a\, da$$

were $\simeq$ denotes the (definitional) equality of denotations.

[2] Again, this definition is sound under Gonzalez' change structures for functions, not Giarrusso *et al.*'s

We contribute to this work by generalizing this result to an untyped setting. This proof requires a ternary step-indexed logical relation which acts as an invariant relating the base computation and the modified computation using the evaluation of the derivative. The basic idea of this proof is to include the validity of changes into the logical relation and to make sure, as usual, that it stays stable by function application, i.e. that valid function changes relate valid changes to valid changes. Proofs about step-indexed logical relations, especially ternary ones, can quickly become technical. For this reason, we mechanize this development using the Coq proof assistant. We give an informal statement for this theorem because introducing the logical relation would force us to enter to many details for this document:

Theorem 7 (Soundness of static differentiation on untyped λ-calculus [13])
If dE is a valid change environment from the base environment E to the modified environment E' and if the evaluation of term t converges under E to a value v and under E' to a value v', then $\mathcal{D}(t)$ converges to a change value dv such that dv is a valid change from v to v'.

4 Incremental programming in Cache-Transfer-Style

Both the dynamic differentiation of Gonzalez' Ph.D. and the static differentiation of Giarrusso's Ph.D. suffer from a critical inefficiency. Indeed, the differentiation of applications include the arguments of the base computation in the derivative. As a consequence, the derivative recomputes these arguments even though this evaluation has already been carried out by the base function. Let us consider the following function which computes the average value of a list of integers in OCaml:

```
let average : int list -> int = fun xs ->
  let s = sum xs in
  let n = length xs in
  let d = div s n in
  d
```

This definition is written in a variant of A-normal form [27] to identify intermediate results explicitly. Applied to `average`, the static differentiation produces the following derivative:

```
let daverage : int list -> (int, Δint) Δlist -> Δint = fun xs dxs ->
  let s = sum xs and ds = dsum xs dxs in
  let n = length xs and dn = dlength xs dxs in
  let d = div s n and dd = ddiv s ds n dn in
  dd
```

Since all derivatives of `div` will need the values of `s` and `n` to compute `dd`, the derivative `daverage` needs to recompute `sum xs` and `length xs`[3]. Thus, this redundancy makes the derivative slower than the base function!

The base function should communicate such intermediate results to the derivative to avoid this inefficiency. To that end, we could use memoization, the process of remembering intermediate results, a standard technique to share computations. Usually, we implement this technique through dynamic insertions and lookups in an associative data structure. However, in this specific setting, the set of cached values is known statically from the shape of the base function's code. This property opens the opportunity for static memoization as pioneered by Liu, Stoller, and Teitelbaum [22].

Static memoization transforms a program in "Cache Transfer Style" (written CTS from now). In that style, every function returns a piece of information about its execution in the form of a cache of a very precise type, which follows the let-binding structure of the function. We share computations by transferring the cache to another part of the program. In contrast with dynamic memoization, the lookup on the cache does not require dynamic tests because the cache type is precise enough to convey the exact location of each intermediate result.

Let us see the effect of this static memoization on our running example. We first rewrite the base function so that it returns a cache of intermediate results in addition to its result:

```
let cts_average : int list -> int * cache_average = fun xs ->
  let s, cache_sum = cts_sum xs in
  let n, cache_length = cts_length xs in
  let d, cache_div = cts_div s n in
  (d, (s, cache_sum, n, cache_length, d, cache_div))
```

[3] This remark also holds under a lazy evaluation strategy.

$$\begin{array}{rcl}
 & & \textbf{Cache of terms} \\
\mathcal{C}(\mathbf{let}\ y = f\,x\ \mathbf{in}\ t) & = & ((\mathcal{C}(t), y), c^{y}_{f\,x}) \\
\mathcal{T}_t(x) & = & () \\
 & & \\
 & & \textbf{CTS translation of terms} \\
\mathcal{T}_t(\mathbf{let}\ y = f\,x\ \mathbf{in}\ t) & = & \mathbf{let}\ y, c^{y}_{f\,x} = f\,x\ \mathbf{in}\ \mathcal{T}_t(f\,x) \\
\mathcal{T}_t(x) & = & (x, \mathcal{C}(t)) \\
 & & \\
 & & \textbf{CTS translation of values} \\
\mathcal{T}(E[\lambda x.t]) & = & \mathcal{T}(E)[\lambda x.\mathcal{T}_t(t)] \\
 & & \\
 & & \textbf{Cache update of terms} \\
\mathcal{U}(\mathbf{let}\ y = f\,x\ \mathbf{in}\ t) & = & ((\mathcal{U}(t), y \oplus dy), c^{y}_{f\,x}) \\
\mathcal{U}(x) & = & () \\
 & & \\
 & & \textbf{CTS differentation of terms} \\
\mathcal{D}_t(\mathbf{let}\ y = f\,x\ \mathbf{in}\ t) & = & \mathbf{let}\ dy, c^{y}_{f\,x} = df\ c^{y}_{f\,x}\ x\ dx\ \mathbf{in}\ \mathcal{D}_t(f\,x) \\
\mathcal{D}_t(x) & = & (dx, \mathcal{U}(t)) \\
 & & \\
 & & \textbf{CTS differentiation of values} \\
\mathcal{D}(E_f[\lambda x.t]) & = & \mathcal{D}(E_f)[\lambda \mathcal{C}(t)\,x\,dx.\mathcal{D}_t(t)]
\end{array}$$

Figure 2.2: Translation and static differentiation in Cache Transfer Style.

The cache of a function is a tuple containing both intermediate results and a cache for each function application it performs. Roughly speaking, the cache is isomorphic to an execution trace structured by tree of function calls.

The derivative takes the cache as an input and, given that its code follows the same shape as the base function, the position of each intermediate results is known statically:

```
let cts_daverage : cache_average -> int list -> (int, Δint) Δlist -> Δint * cache_average
= fun cache xs dxs ->
  let (s, cache_sum, n, cache_length, d, cache_div) = cache in
  let ds, cache_sum = dsum cache_sum xs dxs in
  let dn, cache_length = dlength cache_length xs dxs in
  let dd, cache_div = ddiv cache_div s ds n dn in
  (dd, (s ⊕ ds, cache_sum, n ⊕ dn, cache_length, d ⊕ dd, cache_div))
```

This time the derivative extracts the value `s` and `n` from the cache instead of recomputing them. Hence, the derivative has a complexity which is proportional to the size of `dxs`, not `xs`. The derivative must also update the cache for the next change. In that case, this update also enjoys a constant-time complexity.

Liu's work was dedicated to first-order programs. We generalize static memoization to higher-order programs and we specialize static differentiation to CTS functions. The definitions of the CTS translation and the corresponding CTS differentiation are straightforward. In Figure 2.2, we give these definitions on a core language, an untyped λ-lifted λ-calculus where terms are written in a variant of A-normal form.

A program in that core language is a list of toplevel function definitions of the form $f = \lambda x.t$ and a single term t which is the entry point of the program. This term can contain free variables. Now, how do we reason about the incrementalization of such a program using CTS static differentiation?

We first construct an evaluation environment E containing the closures of the form $E_f[\lambda x.t]$ representing the toplevel functions. These functions must then be translated into CTS: this can be done by applying $\mathcal{T}$ on each function definition of E.

Then, we extend the resulting environment $\mathcal{T}(E)$ with base values for the free variables of t. We obtain an environment F under which we evaluate $\mathcal{T}_t(t)$. If this evaluation converges, we get a value v and a cache C.

We can now change F by replacing each binding $x \mapsto v$ with a new binding $x \mapsto v \oplus dv$ to obtain an environment F'. Again, if the evaluation of t under F' converges, it produces a value v' and a cache C'.

The soundness of our static differentiation guarantees that this recomputation can be avoided by differentiating each closure of $\mathcal{T}(E)$, by extending the resulting environment with two bindings $x \mapsto v$ and $dx \mapsto dv$ for each free variable of t as well as all the bindings contained into the cache C, and by evaluating $\mathcal{D}(t)$. This evaluation will eventually produce the same value v' and the same cache C' as the ones that would have been produced by the recomputation.

This property turns out to be formally stated as follows.

Theorem 8 (Soundness of CTS static differentiation [13])
If the following hypotheses hold,

1. dE is a valid change from E to E',
2. $E \vdash t \Downarrow v$ and
3. $E' \vdash t \Downarrow v'$,

then there exists dv, C and C', such that

1. $\mathcal{T}(E) \vdash \mathcal{T}_t(t) \Downarrow (\mathcal{T}(v), C)$,
2. $\mathcal{T}(E') \vdash \mathcal{T}_t(t) \Downarrow (\mathcal{T}(v'), C')$, and
3. $\mathcal{T}(dE), \mathcal{C}(t) \mapsto C \vdash \mathcal{D}_t(t) \Downarrow (\mathcal{T}(dv), C')$ where $v' = v \oplus dv$.

Even though the definitions are simple, the correctness proof of this program transformation is challenging[4]. Indeed, the proof amounts to show a simulation relation between two triples of reductions, the first of these triples is made of the traces of the base computation, of the modified computation and of the derivative while the second of these triples corresponds to the same traces written in CTS. The invariant which makes the simulation proof work needs many details which include (a) a predicate which relates the validity relations on both side ; (b) a predicate which models the updating of caches during the evaluation of derivatives. Using the Coq proof assistant to conduct this proof is to me the only reasonable way to deal with that level of technicalities.

5 Related work

Memoization Memoization is a well-known technique to avoid recomputation. The basic idea consists in remembering the value of $f(v)$ the first time this application is performed and returning this value directly for the next evaluations of $f(v)$. If f is recursive and if $f(v)$ and $f(v \oplus dv)$ share some recursive calls, memoization already offers some basic incrementalization. For instance, imagine that `List.memo_map` is a memoized version of the standard function `List.map`[5]. If we compute `List.memo_map f l` and `List.memo_map f (v :: l)` after that, then the second expression will only consume a constant time since `List.memo_map f (v :: l)` is equal to `f v :: List.memo_map f l`. This trick works as long as the two lists share a common suffix but it fails otherwise, even if the two lists share most of their elements.

Differentiation provides a richer form of incrementalization. Typically, whatever `dl` is, we know that the result of `List.dmap f df l dl` is equal to Δ`List.map f df dl` because `List.map` is a morphism. We therefore have the guarantee that the complexity of the derivative is proportional to the size of `dl`. Of course, the application of this change to the original list takes linear time but this application is seldom mandatory. Actually, the very purpose of an incremental programming language is to express computations on changes, not on base values. Hence, in a fashion that is reminiscent of deforestation techniques, base values tend to disappear from the intermediate computations of incremental programs.

[4] This is often the case. For instance, closure conversion or translation to continuation-passing style are also simple and technically challenging to prove sound.

[5] `List.map f [a1;...;aN] = [f a1; ...; f aN]`

Self-adjusting computations The framework of self-adjusting computations[1] provides fully automatic incrementalization for imperative programs. To develop an incremental program in that setting, the programmer must base its implementation on "modifiable" references. Roughly speaking, modifiable references are similar to mutable cells except that they serve as parameters to adjust a computation, i.e., to externally modify the data and the decision made during the evaluation.

Self-adjusting computations are instrumented to build a memoized dynamic dependency graph. This graph includes the modifiable and the relationship between their values. To adjust a computation, the runtime efficiently propagates mutations of modifiable references in the graph. This programming technique can be implemented as a library, as for instance in the `incremental` library of JaneStreet [19]. Nevertheless, characterizing the worst-case complexity of change propagation for a given program requires a clear understanding of the shape of the dynamic dependency graphs that can arise during all the possible evaluations. Despite some efforts to provide a cost semantics [21] to help reasoning about the complexity of change propagation, this reasoning is far from being mechanically tractable.

By contrast, we believe that incrementalization by static differentiation will lead to simpler reasoning. Indeed, the derivative is a "mere" program on which we can apply standard reasoning techniques. Contrary to dynamic dependency graphs, a program is a static object on which deductive and compositional analysis can be performed.

The future will say if the aggresive optimizations found in libraries for self-adjusting computations will be transferrable to the setting of static differentiation. One critical aspect is the optimization of the time-space trade-off, i.e., to control the memory consumed by caches. This has been a recent central issue [16, 5] in the implementation of self-adjusting computations.

Differential λ-calculus Ehrhard and Regnier introduces a differential λ-calculus in a quest to give a new denotational semantics to λ-calculus that approximates functions by polynomials instead of finite functions as in Scott domains.

Gonzalez' differential λ-calculus shares some similarity with the calculus of Ehrhard and Regnier as they both use a partial derivation operator, which computes a variation of a term with respect to the variation of one of its free variables. However, the derivatives of Ehrhard and Regnier are linear approximations of functions while our derivatives compute the exact difference between $f\,(x \oplus dx)$ and $f\,(x)$. Besides, the notion of variation considered by Ehrhard and Regnier is the replacement of a value by another one whereas the notion of variation captured by a change structure is closer to an abstract notion of finite difference. Finally, Gonzalez' calculus is deterministic while Ehrhard and Regnier's is non-deterministic. Alvarez-Picallo and Ong [3] try to reconciliate the two approaches in a more general theory through their study of change actions. However, the exact connections between these two differential calculus still have to be formally explored.

Automatic differentiation Automatic differentiation of functions from $\mathbb{R}^n$ to $\mathbb{R}^m$ is a well-studied problem which has received attention lately given its critical role in the implementation of machine learning systems. The challenge of automatic differentiation is to lift derivatives of primitive numerical functions (trigonometric, exponentials, ...) to programs that make use of them. This challenge is the same as ours except that we focus on higher-order functional programs while these technique traditionally focus on imperative and iterative programs.

A noticeable exception is the work of Elliott [10] who revisited automatic differentiation through the eyes of categorical abstractions. Following Lambek [20], Conal Elliott translates λ-terms into their categorical forms and shows that the interpretation of a function can include its derivative by choosing a well-suited target category. In addition to that, this work rephrases the mechanism of differentiation mode (forward or reverse) as a choice between direct or indirect style for the interpretation of the program control flow. Unfortunately, most of these target categories are not cartesian closed which prevent higher-order programs to be differentiated using this technique.

6 Conclusion and Future Work

The implementations of ΔCaml and ΔCoq are still work-in-progress. Along the Ph.D. of Lourdes Gonzalez and the collaboration with Paolo Giarrusso, we have solved many show-stopper issues that prevented the realization of this project: we now have a clear compilation chain in mind and also the firm belief that incremental programs written in cache-transfer-style can be both provable and efficient. We are therefore confident that a prototypical functional language with a derivation operator will be implemented very soon.

To conclude this chapter, we describe the next step of that project: guiding the implementation by relevant use cases. Our preliminary experiments [13] show encouraging results regarding the efficiency of CTS derivatives. However, we must guide the design of our implementations by more realistic applications to make sure that the approach scales to a large class of applications.

Incremental list processing Since the very first time of functional programming, language designers give a high importance to list processing facilities. In his master thesis, Olivier Martinot implements a module for incremental list processing to be included in the ΔCaml distribution. While simple first-order functions like `List.length` are easy to incrementalize, the higher-order function `List.fold_left` highlights interesting challenges. Indeed, to our knowledge, there is no efficient derivative for `List.fold_left`. Intuitively, the impact of a modification of the accumulator value at a specific step of the iteration can propagate to all the steps that are coming next. Thus, the worst-case complexity of the derivative is linear.

Even though the general case of `List.fold_left` incrementalization behaves badly, many specific cases can enjoy a much satisfiying algorithmic complexity. For instance, if the iterated function is commutative and reversible, the actual position of an input list change has no impact on the final result: we can assume that it has been applied on the last position and update the accumulator with a single step. If the iterated function is associative, the derivative of `List.fold_left` can be given a logarithmic complexity thanks to a new variant of finger trees [17].

Certified incremental type checking, reloaded Our original interest for incremental computations was motivated by the desire to implement incremental type checkers. Indeed, during my Ph.D. thesis, we developed several prototypes for rich type systems that ultimately reduced type-checking to decision procedures, typically SAT solvers or automatic provers for first-order logic. Such tools significantly increase the expressive power of type systems allowing them for instance to check that generated programs are well-scoped [24], to remove array bound checking [29] or to verify logical assertions [26].

However, these decision procedures are computationally expensive: it is not uncommon to wait a dozen of seconds to get an answer from them on realistic proof obligations. Their introduction in the workflow of programmers is therefore compromised from a usability perspective. For instance, when I proved OCaml's `Set` module using Higher-Order Hoare Logic [26], almost seven hundred proof obligations were generated during each type-checking, after each small modifications of the source code.

Since most of these proof obligations were only slightly modified and morally equivalent to their counterparts generated by the previous typechecking: the tool was using some heuristics to decide which proofs should be reused and which proof obligations should be transmitted to the provers. This situation was not satisfying because the heuristics were a bit complex and hence bugs found their way to their implementations.

The objective of a certified incremental type checker is to decide proof derivation reuse with strong guarantees that this reuse is licit. I started a collaboration with Mathias Puech on that topic and, inspired by Contextual Modal Type Theory[23], we developed a generic verifier for incremental type checking certificates based on a variant of LF with contexts. This logical framework was able to model proof subderivation reuse and to check for the validity of these reuses in an incremental way [25]. However, this work was wrongly assuming that writing incremental type checkers was the easy part. We now realize that it is not the easy part and we hope to be able to write an incremental typechecker using ΔCaml very soon.

References

[1] Umut A. Acar and Ruy Ley-Wild. "Self-adjusting Computation with Delta ML". In: *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Ed. by Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra. Vol. 5832. Lecture Notes in Computer Science. Springer, 2008, pp. 1–38.

[2] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. “Partiality, Revisited - The Partiality Monad as a Quotient Inductive-Inductive Type”. In: *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings.* Ed. by Javier Esparza and Andrzej S. Murawski. Vol. 10203. Lecture Notes in Computer Science. 2017, pp. 534–549.

[3] Mario Alvarez-Picallo and C.-H. Luke Ong. “Change Actions: Models of Generalised Differentiation”. In: *Foundations of Software Science and Computation Structures - 22nd International Conference, FOSSACS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings.* Ed. by Mikolaj Boja’nczyk and Alex Simpson. Vol. 11425. Lecture Notes in Computer Science. Springer, 2019, pp. 45–61.

[4] Yufei Cai et al. “A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014.* Ed. by Michael F. P. O’Boyle and Keshav Pingali. ACM, 2014, pp. 145–155.

[5] Yan Chen, Umut A. Acar, and Kanat Tangwongsan. “Functional programming for dynamic and large data with self-adjusting computation”. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014.* Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, 2014, pp. 227–240.

[6] Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. “Foundations of dependent interoperability”. In: *J. Funct. Program.* 28 (2018), e9.

[7] Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. “Partial type equivalences for verified dependent interoperability”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016.* Ed. by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ACM, 2016, pp. 298–310.

[8] Erik D. Demaine, John Iacono, and Stefan Langerman. “Retroactive data structures”. In: *ACM Trans. Algorithms* 3.2 (2007), p. 13.

[9] Thomas Ehrhard and Laurent Regnier. “The differential lambda-calculus”. In: *Theor. Comput. Sci.* 309.1-3 (2003), pp. 1–41.

[10] Conal Elliott. “The simple essence of automatic differentiation”. In: *PACMPL* 2.ICFP (2018), 70:1–70:29.

[11] Sigbjørn Finne and Simon Peyton Jones. “Programming Reactive Systems in Haskell”. In: *Proceedings of the 1994 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, September 12-14, 1994.* Ed. by Kevin Hammond, David N. Turner, and Patrick M. Sansom. Workshops in Computing. Springer, 1994, pp. 50–65.

[12] Paolo G. Giarrusso. “Optimizing and incrementalizing higher-order collection queries by AST transformation”. PhD thesis. University of Tubingen, 2018.

[13] Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. “Incremental λ -Calculus in Cache-Transfer Style - Static Memoization by Program Transformation”. In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings.* Ed. by Luís Caires. Vol. 11423. Lecture Notes in Computer Science. Springer, 2019, pp. 553–580.

[14] Gaëtan Gilbert et al. “Definitional proof-irrelevance without K”. In: *PACMPL* 3.POPL (2019), 3:1–3:28.

[15] Lourdes Del Carmen González-Huesca. “Incrementality and effect simulation in the simply typed lambda calculus. (Incrémentalité et simulation d’effets dans le lambda calcul simplement typé)”. PhD thesis. Paris Diderot University, France, 2015.

[16] Matthew A. Hammer and Umut A. Acar. “Memory management for self-adjusting computation”. In: *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, Tucson, AZ, USA, June 7-8, 2008.* Ed. by Richard E. Jones and Stephen M. Blackburn. ACM, 2008, pp. 51–60.

[17] Ralf Hinze and Ross Paterson. “Finger trees: a simple general-purpose data structure”. In: *J. Funct. Program.* 16.2 (2006), pp. 197–217.

[18] Patrick Jaillet and Michael R Wagner. *Online Optimization.* Springer Publishing Company, Incorporated, 2012.

[19] JaneStreet. *The incremental library for self-adjusting computations in OCaml.* `https://github.com/janestreet/incremental`.

[20] Joachim Lambek. “Cartesian Closed Categories and Typed Lambda- calculi”. In: *Combinators and Functional Programming Languages, Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985, Proceedings.* Ed. by Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet. Vol. 242. Lecture Notes in Computer Science. Springer, 1985, pp. 136–175.

[21] Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. *A cost semantics for self-adjusting computation.* Ed. by Zhong Shao and Benjamin C. Pierce. 2009.

[22] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. “Static Caching for Incremental Computation”. In: *ACM Trans. Program. Lang. Syst.* 20.3 (1998), pp. 546–585.

[23] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. “Contextual modal type theory”. In: *ACM Trans. Comput. Log.* 9.3 (2008), 23:1–23:49.

[24] François Pottier. “Static Name Control for FreshML”. In: *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings.* IEEE Computer Society, 2007, pp. 356–365.

[25] Matthias Puech. “Certificates for Incremental Type Checking”. PhD thesis. University of Bologna, 2013.

[26] Yann Régis-Gianas and François Pottier. “A Hoare Logic for Call-by-Value Functional Programs”. In: *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings.* Ed. by Philippe Audebaud and Christine Paulin-Mohring. Vol. 5133. Lecture Notes in Computer Science. Springer, 2008, pp. 305–335.

[27] Amr Sabry and Matthias Felleisen. “Reasoning about Programs in Continuation-Passing Style”. In: *Lisp and Symbolic Computation* 6.3-4 (1993), pp. 289–360.

[28] Tarmo Uustalu and Niccolò Veltri. “Partiality and Container Monads”. In: *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings.* Ed. by Bor-Yuh Evan Chang. Vol. 10695. Lecture Notes in Computer Science. Springer, 2017, pp. 406–425.

[29] Hongwei Xi. “Dependent ML An approach to practical programming with dependent types”. In: *J. Funct. Program.* 17.2 (2007), pp. 215–286.

CHAPTER 3

Certified software evolutions

Initial commit.

The most frequent (and useless?) commit message on GitHub [24].

Contents

Related publications:

- *Verifiable semantic difference languages.*

  PPDP (2017) with Thibaut Girka, and David Mentré.

- *A mechanically checked generation of correlating programs directed by structured syntactic differences.*

  ATVA (2015) with Thibaut Girka, and David Mentré.

1 Introduction

Context Software is built by successive modifications of its source code. These changes are meant to extend its features, to adjust its behavior, or to fix an inadequacy between its implementation and its specification. For instance, in July 2019, the Linux kernel is the outcome of 853, 850 patches.

Standard development tools characterize code source patches in a textual way: the tool `diff` [15] represents the difference between two versions A and B of a program by enumerating the lines to be removed or to be inserted on the version A to get the version B.

Textual differences are formally defined, but the programmer must make a significant effort of analysis to determine if a given textual difference indeed matches his intent. As a matter of fact, nothing guarantees the correspondence between these intents and the real impact of a code source change on the program behavior: a new function may incorrectly implement the desired new feature, a bug fix can be invalid or incomplete, ...

This analysis of software evolutions is a daily task for developers. Hence, code review is taken seriously and handled by dedicated collaborative development tools like GitLab [13]. In most of today's development processes, the developer produces a *Pull Request* that must be reviewed and often refined before it gets merged into the master code base.

Unfortunately, this analysis of evolutions is informal. The intents of the programmer are written using natural language in commit messages and no tool[1] assists the reviewer to check the adequacy between code source modifications and the patch intents.

The absence of tools dedicated to the validation of software evolutions is terrible for software quality: if the code reviewing process increases the code readability and catches some classes of programming errors, its efficiency is conditioned by systematic reviews conducted by expert developers [18, 1]. Hence, a good code review is an expensive development activity. Besides, some empirical studies seem to demonstrate that code reviews hardly led to the detection of conceptually deep problems, typically the ones related to security [9].

This absence of tools to check software evolutions is even more critical in the context of certified software: industrial software certification instances consider that any modification of the source code of certified software invalidates the whole certification process. The certification must then be restarted from the beginning. The cost of this revalidation is gigantic and the critical software industry would drastically reduce its certification costs and its delay if it existed a valid process to certify the evolution of software.

The industrial static analyzers are not designed for the analysis of changes [19] because they analyze only one program at a time. For this reason, several research teams recently focalized their efforts on the *relational analysis of programs*. In this field of research, we aim at finding theoretical and practical tools to show that a given relation holds between two close programs.

If the analysis of a single program often amounts to the formalization of its invariants, an extra problem appears when we want to establish a relationship between two programs. Indeed, not only we need to find an (invariant) relation between them, but we also need to find a specific synchronization of the two compared programs. This new problem is interdependent with the problem of invariant generation, so it adds a significant dose of complexity in the whole certification process.

Contributions Certified software evolution is the central topic of Thibaut Girka's Ph.D. thesis.

In Section 2, we present the first contribution [11] of this thesis which is the design and the correctness proof of a generation algorithm for correlating programs. This work fixes an unsound algorithm of the literature [20]. The challenging aspect of this generation algorithm comes from the high number of corner cases which are quite subtle to deal with. To avoid falling in that traps too, the proof of this algorithm has been conducted using the Coq proof assistant.

Section 3 is about a second contribution [12] of Thibaut Girka's work to the field of relational program analysis. We generalize Barthe's notion of product program to be able to compare programs that diverge and programs that get stuck with other programs. To that end, we introduce correlating oracles which are a sort of bi-interpreters written in Type Theory whose role is to realize a correlation between the traces of two programs. This work led to a Coq library that provides a framework to define verifiable differences languages. Such a language can be used to specify and to verify the relation between two close programs. We define such a language for a toy imperative programming language.

2 Certification of a correlating program generator

2.1 Correlating programs

Consider the following two programs:

```
sum = 0;
x = -x;
y = 0;
while (sum < x) {
  y = y + 1;
  sum = sum + 1 + 2 * y;
}
sum = 0;
```

```
sum = 0;
x = -x;
count = 0;
while (sum < x) {
  count = count + 1;
  sum = sum + 1 + 2 * y;
}
sum = 0;
```

[1] Except the essential continuous integration tools like Travis-CI [25] or Circle-CI [8] which checks that the patches do not introduce a regression with respect to the testsuite.

A pair of exercised eyes can have correctly guessed the difference between these two programs: the variable identifier y has been renamed into count, probably to improve the readability of the source code. If the difference does not pop up, the diff utility can be useful as it produces:

```
  sum = 0;
  x = -x;
- y = 0;
+ count = 0;
  while (sum < x) {
-   y = y + 1;
+   count = count + 1;
-   sum = sum + 1 + 2 * y;
+   sum = sum + 1 + 2 * count;
  }
  sum = 0;
```

which highlights removed lines by prefixing them with a minus and inserted lines by prefixing them with a plus.

As explained in the introduction, this textual difference is not directly exploitable to formally compare the behaviors of the two programs. Partush and Yahav [20] introduce a transformation that takes as input a textual difference and produces a so-called *correlating program*. This program statically interleaves the instructions of the two input programs to expose a relation between their variables.

There are obviously many ways to statically schedule the instructions of the two input programs. Partush and Yahav [20] propose heuristics which favor the factorization of the loop structures of the two programs. Whatever the chosen scheduling, the correlating must be sound in the following sense.

Definition 9 (Soundness for correlating programs)
Let P_1 and P_2 be two imperative input programs that terminate normally with two final stores S_1 and S_2. A correlating program is *sound* if it terminates normally with a final store $S = S_1 \uplus S_2$.

For instance, here is the correlating program obtained for our example:

```
sum = 0; T_sum = 0;
x = -x; T_x = - T_x;
y = 0; T_count = 0;
guard = (sum < x); T_guard = (T_sum < T_x);
while (guard || T_guard) {
  if (guard) y = y + 1;
  if (T_guard) T_count = T_count + 1;
  if (guard) sum = sum + 1 + 2 * y;
  if (T_guard) T_sum = T_sum + 1 + 2 * T_count;
  if (guard) guard = (sum < x);
  if (T_guard) T_guard = T_sum < T_x;
}
```

We first notice that the variables identifiers of the second program have been prefixed by T_ to avoid conflicts with the identifiers of the first program. At the beginning of this program, we also recognize the first three assignments of the two input programs: these two sequences of assignments are interleaving in lock-step so that it becomes “clear” that the unchanged lines are semantically related. The two loops are significantly rewritten as a single loop whose condition is defined by the disjunction of two boolean variables, named *guards*. This loop accounts for an alignment of the two input loops and the disjunction between their guards allows the generated loop to iterate as long as the longest input iteration is executed. When the shortest loop is finished, the instructions of its body must be deactivated, hence their execution is guarded by a conditional statement.

It is easy to convince ourselves that our example correlating program is sound. We can then prove a relationship between the execution traces of the two input programs by proving a property on this correlating program. As said earlier, the benefits of reasoning about a single program to compare two programs is the opportunity to reuse static analyzers. Partush and Yahav [20] use their own abstract interpreter to show relational properties between the variables of the two input programs.

2.2 Unsound generation of correlating programs

Unfortunately, the procedure described by Partush and Yahav [20] is unsound. As a matter of fact, if we carefully follow their instructions to build a correlating program for the following two:

```
void fail (int x) {
  i = 0;
  while (i <= 1) {
    i = i + 1;
    x = x + 1;

  }
}
```

```
void fail (int x) {
  i = 0;
  while (i <= 1) {
    i = i + 1;
    x = x + 1;
    break;
  }
}
```

we obtain the following correlating program:

```
1  void fail (int x) {
2    int T_x = x;
3    int i = 0; int T_i = 0;
4  L1: T_L1: ;
5    guard G0 = (i <= 0);
6    guard T_G0 = (T_i <= 1);
7    if (G0) i = i + 1;
8    if (T_G0) T_i = T_i + 1;
9    if (G0) x = x + 1;
10   if (T_G0) T_x = T_x + 1;
11   if (T_G0) goto T_L3;
12   if (G0) goto L1;
13   if (T_G0) goto T_L1;
14 L3: T_L3: ;
15 }
```

This correlating program is conspicuously wrong. Consider the execution of `fail (2)` in the first program: it stops with `i = 2`. Here, because of the goto on Line 11, the correlating program misses one loop iteration and stops with `i = 1`.

The culprit is the translation of `break` which should not be translated by a `goto` instruction. In our opinion, Partush and Yahav [20]'s is also fundamentally fragile because it is based on a line-by-line textual difference which is only remotely related to the abstract syntax trees of the two input programs. More generally, the definition of such generation algorithms is error-prone because – not unlikely to incremental programs presented in the previous chapter – they must consider a large number of comparison cases.

2.3 A new, mechanically checked, correlating program generator

In the Ph.D. thesis of Thibaut Girka, we propose a new (and mechanically checked in Coq [2]) algorithm to generate correlating programs based on the ideas of Partush and Yahav [20]. This fixed algorithm exploits several ideas that we will explain in this section.

But before that, we can have a look at the correlating program generated by this new algorithm on the two input programs of the previous section:

[2] The development is downloadable at https://www.irif.fr/~thib/atva15/.

```
void fail (int O_x) {
  int T_O_x = O_x;
  int O_i = 0; int T_O_i = 0;
  guard G1 = 1; guard T_G1 = 1;
  guard G0 = (O_i <= 1); guard T_g0 = (T_o_i <= 1);
  while (G0 || T_G0) {
    if (G0) G1 = 1;
    if (T_G0) T_G1 = 1;
    if (G0) if (G1) O_i = O_i + 1;
    if (T_G0) if (T_G1) T_O_i = T_O_i + 1;
    if (G0) if (G1) O_x = O_x + 1;
    if (T_G0) if (T_G1) T_O_x = T_O_x + 1;
    if (T_G0) if (T_G1) T_G0 = 0; // encodes break.
    if (G0) G0 = (O_i <= 1);
    if (T_G0) T_G0 = (T_O_i <= 1);
  }
}
```

This new program has more or less the same ingredients as the wrong one of the previous section: the execution of instructions are similarly conditioned by guard variables. This time, however, `while`-loops are not replaced by `goto`s. Their conditions are also encoded by two guards, one per input program. By setting one of this guard to `0`, we only deactivate the instructions of one of the loop, not both loops. Moreover, with another couple of guards (`G1` and `T_G1` here) we can deactivate the instructions of a loop for the current iteration only.

The correlating programs generated by our algorithm seems to better handle `while`-loops, `break` and `continue` but we could have missed a corner case and for this reason, we prefer to formally mechanize the soundness proof of our generation algorithm. The formalization is defined over a variant of the standard Imp language extended with `continue` and `break`. The syntax for the commands of this language is defined as follows.

Definition 10 (Imp Syntax)

$$
\begin{array}{lr}
c ::= a \mid c; c \mid \textbf{while}\ (b)\ c \mid \textbf{if}\ (b)\ c\ \textbf{else}\ c & (\textsc{Commands}) \\
a ::= \textbf{skip} \mid x = e \mid \textbf{break} \mid \textbf{continue} & (\textsc{Atomic commands}) \\
b ::= \textbf{true} \mid \textbf{false} \mid b \,\&\&\, b \mid !b \mid e \leq e & (\textsc{Boolean Expressions}) \\
e ::= x \mid \textbf{n} \mid e + e & (\textsc{Arithmetic Expressions})
\end{array}
$$

Our algorithm takes structured syntactic program differences as input where Partush and Yahav [20] expect a textual difference. The algorithm proceeds by induction over this structured difference to unambiguously produce a correlating program. The generation algorithm is not responsible for choosing a static scheduling between the two input programs because the structured difference already encodes the chosen interleaving. As a consequence, we get a general result which is independent from the specific heuristic chosen to correlate the compared programs. We do propose a heuristic for static scheduling but it is omitted here by lack of space: it follows the same strategy as Partush and Yahav [20], i.e., it tries to factorize the loops of the two programs as much as possible.

Definition 11 (Structured difference language)

$$
\begin{array}{l}
\Delta ::= \pm[c]; \Delta \mid \pm\Delta; [c] \mid \Delta; \Delta \mid a \rightarrow a' \\
\quad \mid \textbf{if}\ (b \rightarrow b')\ \Delta\ \textbf{else}\ \Delta \mid \textbf{while}\ (b \rightarrow b')\ \Delta \\
\quad \mid \pm[\textbf{if}\ (b)\ c\ \textbf{else}]\ \Delta \mid \pm[\textbf{if}\ (b)]\ \Delta\ [\textbf{else}\ c] \mid \pm[\textbf{while}\ (b)]\ \Delta \\
\pm ::= + \mid -
\end{array}
$$

With the first two cases, we can insert or remove a command on the left or on the right of a sequencing operator. The third case maps two changes on the left and the right of a sequence operator.

c	o	$CI(gl, \pi, c, o)$
skip	o	**skip**
$x = e$	o	**if** (gl) $x = e$
c_1; c_2	o	$CI(gl, 0 \cdot \pi, c_1, o)$; $CI(gl, 1 \cdot \pi, c_1, o)$
if (b) c_1 **else** c_2	o	**if** (gl) $\pi = b$; $CI(gl \wedge \pi, 1 \cdot \pi, c_1, o)$; $CI(gl \wedge \neg\pi, 0 \cdot \pi, c_2, o)$

while (b) c	o	**if** (gl) $\pi = b$;
		while $(gl \wedge \pi)$ {
		if $(gl \wedge \pi)$ $1 \cdot \pi =$ **true;**
		$CI(gl \wedge \pi \wedge (1 \cdot \pi), 1 \cdot 1 \cdot \pi, c,$ **Some** $\pi)$;
		if $(gl \wedge \pi)$ $\pi = b$ }

break	**Some** π'	**if** (gl) $\pi' =$ **false**
continue	**Some** π'	**if** (gl) $1 \cdot \pi' =$ **false**

Figure 3.1: Guarded form translation function *CI*.

The fourth case is a replacement of an atomic instruction by another one. The fifth and the sixth cases denote modifications of a condition. The last three cases correspond to the insertion of a new conditional instruction or of a `while`-loop around an existing command. We can extract the two programs related by a difference Δ by writing $\Pi_1(\Delta)$ and $\Pi_2(\Delta)$. The definition of these two functions are left as exercise for the reader.

The target language of the transformation is an imperative language with guarded instructions. To simplify the invariants required for the proofs, this language separates explicitly standard variables and guard variables. Guard variable identifiers are not taken in a usual set of names but are more structured : they are words of bits, i.e. $\pi \in (0|1)^\star$. This representation has good properties as it makes the generation of fresh names easy and, more crucially, as it allows to maintain a mapping between the names of the guard and the positions of the instructions it controls in the abstract syntax tree.

Definition 12 (Syntax for a guarded imperative language)

$$\begin{aligned}
c_G &::= c_G;\ c_G \mid \textbf{skip} \mid \textbf{while}\,(g_\vee)\,c_G \mid \textbf{if}\ (g_\wedge)\ ac_G && (\textsc{Commands})\\
ac_G &::= x = e \mid g = b && (\textsc{Atomic commands})\\
g_\wedge &::= g_\ell \mid g_\ell \wedge g_\wedge && (\textsc{Guard conjunctions})\\
g_\vee &::= g_\wedge \mid g_\wedge \vee g_\vee && (\textsc{Guard disjunctions})\\
g_\ell &::= g \mid \neg g && (\textsc{Guard literals})
\end{aligned}$$

The algorithm proceeds in two steps: first, it renames in the syntactic difference all the variable identifiers of the two programs so that they cannot conflict when injected in the target program. This operation is named “tagging”. Then, the function *CP* transforms the structured difference into a program written in the guarded imperative language defined above.

The definition of *CP* is given in Figure 3.2 and Figure 3.3 and it uses another function named *CI* defined in Figure 3.1. By lack of space, we cannot explain every cases of these two functions. However, let us take the time to explain their signatures.

The function *CI* translates a single command in a guarded form. *CI* expects the current guard condition gl, the current guard identifier π, the command c to be translated and the guard of the innermost loop o which is **None** if c is outside a loop, and **Some** π if c is inside of loop of guard π. By looking at the case for sequence, we notice that the guard identifier π indeed follows the path of the command in the abstract syntax tree. (Since the nodes of this tree have at most two children, a sequence of bits is enough to encode a position in that tree.) There is a specific convention for **while**-loop: if π is the guard for a **while**-loop, $1 \cdot \pi$ is reserved to encode the semantics of **continue**. Thus, we use $1 \cdot 1 \cdot \pi$ for the guard of the loop body.

The function *CP* implements the actual generation of the correlating program. It takes a structured difference Δ, the guard identifiers π_0 and π_1 for the two input programs, and two loop-guards o_0 and o_1 whose values follow the same convention as in *CI*.

Δ	$CP(\Delta,\ \pi_0,\ \pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$
$-[c]; \Delta$	$CI(gl_0,\ 0 \cdot \pi_0,\ c,\ o_0);$ $CP(\Delta,\ 1 \cdot \pi_0,\ \pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$
$-\Delta; [c]$	$CP(\Delta,\ 0 \cdot \pi_0,\ \pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1);$ $CI(gl_0,\ 1 \cdot \pi_0,\ c,\ o_0)$
$+[c]; \Delta$	$CI(gl_1,\ 0 \cdot \pi_1,\ c,\ o_1);$ $CP(\Delta,\ \pi_0,\ 1 \cdot \pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$
$+\Delta; [c]$	$CP(\Delta,\ \pi_0,\ 0 \cdot \pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1);$ $CI(gl_1,\ 1 \cdot \pi_1,\ c,\ o_1)$
$\Delta_0; \Delta_1$	$CP(\Delta_0,\ 0 \cdot \pi_0,\ 0 \cdot \pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$ $CP(\Delta_1,\ 1 \cdot \pi_0,\ 1 \cdot \pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$
$\textbf{if}\ (b_0 \longrightarrow b_1)\ \Delta_0\ \textbf{else}\ \Delta_1$	$\textbf{if}\ (gl_0)\ \pi_0 = b_0;$ $\textbf{if}\ (gl_1)\ \pi_1 = b_1;$ $CP(\Delta_0,\ 1 \cdot \pi_0,\ 1 \cdot \pi_1,\ gl_0 \wedge \pi_0,\ gl_1 \wedge \pi_1,\ o_0,\ o_1);$ $CP(\Delta_1,\ 0 \cdot \pi_0,\ 0 \cdot \pi_1,\ gl_0 \wedge \neg\pi_0,\ gl_1 \wedge \neg\pi_1,\ o_0,\ o_1)$
$\textbf{while}\ (b_0 \longrightarrow b_1)\ \Delta$	$\textbf{if}\ (gl_0)\ \pi_0 = b_0;$ $\textbf{if}\ (gl_1)\ \pi_1 = b_1;$ $\textbf{while}\,((gl_0 \wedge \pi_0) \vee (gl_1 \wedge \pi_1))\,\{$ $\quad \textbf{if}\ (gl_0 \wedge \pi_0)\ 1 \cdot \pi_0 = \textbf{true};$ $\quad \textbf{if}\ (gl_1 \wedge \pi_1)\ 1 \cdot \pi_1 = \textbf{true};$ $\quad CP(\Delta,\ 1 \cdot 1 \cdot \pi_0,\ 1 \cdot 1 \cdot \pi_1,\ gl_0 \wedge \pi_0 \wedge (1 \cdot \pi_0),$ $\qquad\qquad gl_1 \wedge \pi_1 \wedge (1 \cdot \pi_1),\ \textbf{Some}\ \pi_0,\ \textbf{Some}\ \pi_1);$ $\quad \textbf{if}\ (gl_0 \wedge \pi_0)\ \pi_0 = b_0;$ $\quad \textbf{if}\ (gl_1 \wedge \pi_1)\ \pi_1 = b_1;$ $\}$
$a_0 \longrightarrow a_1$	$CI(gl_0,\ \pi_0,\ a_0,\ o_0);$ $CI(gl_1,\ \pi_1,\ a_1,\ o_1)$
$-[\textbf{if}\ (b)\ c\ \textbf{else}]\ \Delta$	$\textbf{if}\ (gl_0)\ \pi_0 = b;$ $CI(gl_0 \wedge \pi_0,\ 1 \cdot \pi_0,\ c,\ o_0);$ $CP(\Delta,\ 0 \cdot \pi_0,\ \pi_1,\ gl_0 \wedge \neg\pi_0,\ gl_1,\ o_0,\ o_1)$
$+[\textbf{if}\ (b)\ c\ \textbf{else}]\ \Delta$	$\textbf{if}\ (gl_1)\ \pi_1 = b;$ $CI(gl_1 \wedge \pi_1,\ 1 \cdot \pi_1,\ c,\ o_1);$ $CP(\Delta,\ \pi_0,\ 0 \cdot \pi_1,\ gl_0,\ gl_1 \wedge \neg\pi_1,\ o_0,\ o_1)$
$-[\textbf{if}\ (b)]\ \Delta\ [\textbf{else}\ c]$	$\textbf{if}\ (gl_0)\ \pi_0 = b;$ $CP(\Delta,\ 1 \cdot \pi_0,\ \pi_1,\ gl_0 \wedge \pi_0,\ gl_1,\ o_0,\ o_1);$ $CI(gl_0 \wedge \neg\pi_0,\ 0 \cdot \pi_0,\ c,\ o_0)$
$+[\textbf{if}\ (b)]\ \Delta\ [\textbf{else}\ c]$	$\textbf{if}\ (gl_1)\ \pi_1 = b;$ $CP(\Delta,\ \pi_0,\ 1 \cdot \pi_1,\ gl_0,\ gl_1 \wedge \pi_1,\ o_0,\ o_1);$ $CI(gl_1 \wedge \neg\pi_1,\ 0 \cdot \pi_1,\ c,\ o_1)$

Figure 3.2: Difference directed correlating program generation function *CP*.

Δ	$CP(\Delta,\ \pi_0,\ \pi_1,\ gl_0,\ gl_1,\ o_0,\ o_1)$

$-$[**while** (b)] Δ	**if** (gl_0) $\pi_0 = b$;
	if $(gl_0 \wedge \pi_0)$ $1 \cdot \pi_0 =$ **true**;
	$CP(\Delta,\ 1 \cdot 1 \cdot \pi_0,\ \pi_1,\ gl_0 \wedge \pi_0 \wedge (1 \cdot \pi_0),\ gl_1,$ **Some** $\pi_0,\ o_1)$;
	if $(gl_0 \wedge \pi_0)$ $\pi_0 = b$;
	while $(gl_0 \wedge \pi_0)$ {
	  if $(gl_0 \wedge \pi_0)$ $1 \cdot \pi_0 =$ **true**;
	  $CI(gl_0 \wedge \pi_0 \wedge (1 \cdot \pi_0),\ 1 \cdot 1 \cdot \pi_0,\ \Pi_0(\Delta),$ **Some** $\pi_0)$;
	  if $(gl_0 \wedge \pi_0)$ $\pi_0 = b$; }

$+$[**while** (b)] Δ	**if** (gl_1) $\pi_1 = b$;
	if $(gl_1 \wedge \pi_1)$ $1 \cdot \pi_1 =$ **true**;
	$CP(\Delta,\ \pi_0,\ 1 \cdot 1 \cdot \pi_1,\ gl_0,\ gl_1 \wedge \pi_1 \wedge (1 \cdot \pi_1),\ o_0,$ **Some** $\pi_1)$;
	if $(gl_1 \wedge \pi_1)$ $\pi_1 = b$;
	while $(gl_1 \wedge \pi_1)$ {
	  if $(gl_1 \wedge \pi_1)$ $1 \cdot \pi_1 =$ **true**;
	  $CI(gl_1 \wedge \pi_1 \wedge (1 \cdot \pi_1),\ 1 \cdot 1 \cdot \pi_1,\ \Pi_1(\Delta),$ **Some** $\pi_1)$;
	  if $(gl_1 \wedge \pi_1)$ $\pi_1 = b$; }

Figure 3.3: Difference directed correlating program generation function CP (continued).

Theorem 9 (Soundness of CP)
If $S_1 \vdash \Pi_1(\Delta) \Downarrow S_1'$ and $S_2 \vdash \Pi_2(\Delta) \Downarrow S_2'$ hold, then $S_1 \uplus S_2 \vdash CI(\Delta, \mathbf{true}, \mathbf{true}, 0, 1, \mathbf{None}, \mathbf{None}) \Downarrow S_1' \uplus S_2'$ holds.

The proof architecture of this theorem is depicted in Figure 3.4. Roughly speaking, the proof is a forward simulation proof, similar to the proofs found in a certified compiler, except that the target program is shown to simulate not only one but two programs at the same time. This double simulation is justified by an invariant that properly relates an instruction of the target program to an instruction of one of the two input programs.

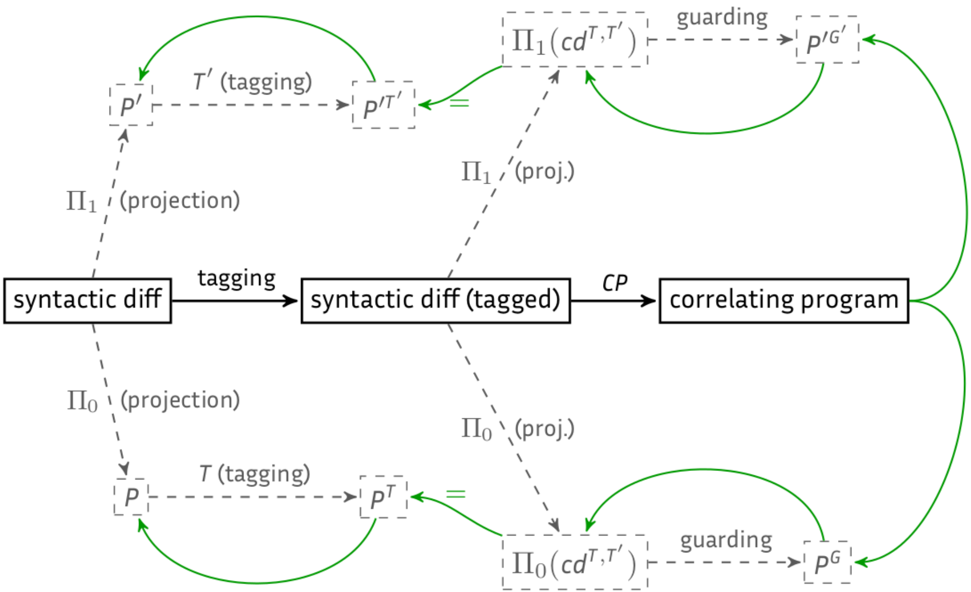


Figure 3.4: Correctness proofs of correlating programs generation.

3 Verifiable semantic differences between programs

The correlating programs of the previous section assume no *a priori* knowledge about the two programs. On the contrary, they try to infer a relationship between the two programs so that the programmer can understand the impact of a change. We switch to another setting now where the programmer asserts that a given difference of behavior exists between two programs and wants to verify that this difference actually holds. This typically requires that we provide the programmer with a specification language for semantic differences and that we also give a program logic for confront this specification with the two programs it is supposed to relate.

The field of deductive relational program analysis creates theoretical and practical tools for the mechanization of program comparison. This line of work started with the seminal work of Benton [6] about Relational Hoare Logic which generalizes the standard Hoare logic triples to characterize pairs of commands instead of single commands. This new judgment is:

$$\{P\}c_1 \sim c_2\{Q\}$$

and can be read like this: “If the disjoint union of the stores of c_1 and c_2 satisfies the precondition P, then the execution of both c_1 and c_2 will result in a disjoint union of two stores satisfying the postcondition Q.”

Relational Hoare Logic is not equipped with the same tools to mechanize deductive verification as the standard Hoare Logic. Fortunately, Barthe, Crespo, and Kunz [3] devise a notion of product program which encodes as a single program the static scheduling of a specific proof of a Relation Hoare Logic triple. Proving an Hoare triple about the product program is then equivalent to the proof of the Relational Hoare Logic triple. This verification technique – quite similar to the one based on correlating programs presented in the previous section – enables the reuse of existing verification tools [16, 17, 10] to prove relational properties.

There is actually no canonical way to define a Relational Hoare Logic [7] because a finite derivation tree cannot encode an arbitrary scheduling of the two program instructions. Even though Relational Hoare Logic has been further extended with more rules and thus more flexibility in the static scheduling [3], none of these generalizations is complete until a self-composition rule of the form:

$$\frac{\{P\}c_1; c_2\{Q\}}{\{P\}c_1 \sim c_2\{Q\}}$$

is introduced in the proof system. This rule is a fallback to a standard proof on a single program based on a Hoare logic, and is therefore a failure confession. Indeed, it expresses the fact that the closedness of the two programs cannot be exploited to compare them. More fundamentally, this rule does not extend its application to non terminating programs since if c_1 diverges, we cannot assert anything about c_2.

Another limitation of Relational Hoare Logic is its inability to compare a program that gets stuck with other program. This problem can be illustrated with the following two programs:

```
m = a % b;
if (m == 0)
  is_multiple = 1;
else
  is_multiple = 0;
```

```
if (b == 0) {
  is_multiple = 0;
} else {
  m = a % b;
  if (m == 0)
    is_multiple = 1;
  else
    is_multiple = 0;
}
```

The program on the left has a manifest bug if `b = 0`. This bug is fixed in the program on the right which defensively handles that case with a specific instruction. Here is the product program generated for these two programs:

```
if (r_b == 0) {
   r_is_multiple = 0;
   l_m = l_a % l_b;
   if (l_m == 0)
     is_multiple = 1;
   else
     is_multiple = 0;
} else {
  l_m = l_a % l_b;
  r_m = r_a % r_b;
  assert ((l_m == 0) == (r_m == 0));
  if (l_m == 0) {
    l_is_multiple = 1;
    r_is_multiple = 1;
  } else {
    l_is_multiple = 0;
    r_is_multiple = 0;
  }
}
```

If `r_b = 0` and `l_b = 0`, the product program will execute a division by zero coming from the left program. Therefore, the product program crashes and we cannot prove that the second program does not.

3.1 Correlating oracles

The objective of Girka's Ph.D. thesis was to find an expressive framework to characterize the difference of semantics between programs. The question of proof automation, which is central to the work of correlating programs and product programs, was explicitly put out of scope for a better separation of concerns. The result of that work is a Coq library which defines a programming language agnostic notion of verifiable semantic differences[3].

In that setting, a semantic difference denotes an arbitrarily complex relation between two reduction traces that can be terminated normally or on stuck configuration, or be infinite. In practice, we found ourselves already satisfied by a specific presentation of trace relations that we call γ-correlations.

Definition 13 (γ-correlation)
If γ is a binary relation over configurations of two programming languages $\mathcal{L}_1$ and $\mathcal{L}_2$, a trace relation is a γ-correlation if it is defined by the following co-inductive rules:

$$\frac{|\overline{C}_1| + |\overline{C}_2| > 0 \qquad \lceil \overline{C}_1 \cdot \mathcal{T}_1 \rceil \gamma \lceil \overline{C}_2 \cdot \mathcal{T}_2 \rceil \qquad \mathcal{T}_1 \overset{\gamma}{\sim} \mathcal{T}_2}{\overline{C}_1 \cdot \mathcal{T}_1 \overset{\gamma}{\sim} \overline{C}_2 \cdot \mathcal{T}_2} \qquad\qquad \frac{c_1 \gamma c_2}{c_1 \overset{\gamma}{\sim} c_2}$$

where $\mathcal{T}$ represents a trace, i.e. a nonempty sequence of configurations, $\overline{C}$ a possibly empty sequence of configurations, and $\lceil \bullet \rceil$ represents the first configuration of a trace.

A visual illustration of a γ-correlation is given in Figure 3.5. Intuitively, a γ-correlation is a trace relation made of two components: a relational invariant γ between configurations and a correlation strategy which decides which configurations are related by γ.

Presenting trace relations as γ-correlations has two advantages over a less restricted formulation. First, we separate the reduction relation into a static part, the relation γ, and a dynamic part, the correlation strategy. This makes the definitions of γ-correlation easier to read and to understand. Second, if the correlation strategy is expressed as an inhabitant of Type Theory and if γ is indeed an invariant of the stream of configuration pairs it correlates when applied to two programs P_1 and P_2, then we have a proof that the γ-correlation holds between P_1 and P_2!

A correlating oracle is a realization in Type Theory of a specific correlation strategy. An oracle can be seen as a bi-interpreter that *dynamically schedule* the instructions of two programs to make explicit all the pairs of configurations that are correlated. Correlating oracles improve over product programs

[3] To be precise, our framework is restricted to deterministic programming languages.

$$\begin{tikzcd}
s_l^1 \arrow[r] \arrow[d, dash, "\gamma"] & s_l^2 \arrow[r] \arrow[dr, dash, "\gamma"] & s_l^3 \arrow[r] \arrow[d, dash, "\gamma"] & s_l^4 \arrow[r] & s_l^5 \arrow[dl, dash, "\gamma"] & \cdots \\
s_r^1 \arrow[r] & s_r^2 \arrow[r] & s_r^3 \arrow[r] & s_r^4 \arrow[r] & s_r^5 & \cdots
\end{tikzcd}$$

Figure 3.5: A γ-correlation between two relation traces.

for two reasons. First, as the correlating oracle acts as an interpreter, a crash of one of two programs cannot have an impact on the oracle's execution. Second, since the oracle can use its own logic to dynamically schedule the instructions of the two programs, it is not constrained to follow the same control flow as them. These two points increase the class of trace relations definable using a correlating oracle compared to the ones definable with a product program.

Since there are as many different correlation strategies as they are ways to compare two programs, we introduce a notion of oracle languages, where each oracle language is a family of oracles following the same strategy:

Definition 14 (Correlating Oracles)
Given two programming languages $\mathcal{L}_1$ and $\mathcal{L}_2$. Let us write $\mathcal{E}_1$ (resp. $\mathcal{E}_2$) for the evaluation function of $\mathcal{L}_1$ (resp. $\mathcal{L}_2$). An oracle language definition between $\mathcal{L}_1$ and $\mathcal{L}_2$ is a 6-uple ($\mathbb{O}$, $\mathbb{S}$, π_1, π_2, σ, $\mathbb{I}$) such that:

- $\mathbb{O}$ is the type of oracle configurations.
- $\mathbb{S}$ is the interpretation function of type $\mathbb{O} \ \rightarrow \mathbb{O}$.
- π_1 (resp. π_2) is a projection from the oracle configurations to the configurations of $\mathcal{L}_1$ (resp. $\mathcal{L}_2$).
- σ is a difference inference function $\mathcal{L}_1 \times \mathcal{L}_2 \ \rightarrow \mathbb{O}$.
- $\mathbb{I}$ is an invariant over oracle configurations.

with the following additional requirements ensuring its soundness:

1. $\mathbb{I}$ is preserved by $\mathbb{S}$ i.e.:
$$\forall o : O, \mathbb{I}(o) \rightarrow \mathbb{I}(\mathbb{S}(o))$$
2. $\mathbb{S}$ leads to correct and productive predictions i.e.:
$$\forall o, o' : O, \mathbb{I}(o) \rightarrow \mathbb{S}(o) = o' \rightarrow \exists n_1 n_2, \begin{cases} \pi_1(o') = \mathcal{E}_1^{n_1}(\pi_1(o)) \\ \pi_2(o') = \mathcal{E}_2^{n_1}(\pi_2(o)) \\ n_1 + n_2 > 0 \end{cases}$$
3. the oracle is complete, in the sense that it only terminates when both underlying programs themselves terminate, i.e. :
$$\forall o : O, o \notin \mathrm{dom}(\mathbb{S}) \rightarrow \pi_1(o) \notin \mathrm{dom}(\mathcal{E}_1) \wedge \pi_2(o) \notin \mathrm{dom}(\mathcal{E}_2)$$

The three technical requirements must be fulfilled to formally turn a correlating oracle into a relational proof scheme as coined above.

Theorem 10 (Adequacy of γ-correlations represented by correlating oracles)
Let O a correlating oracle taken in an oracle language ($\mathbb{O}$, $\mathbb{S}$, π_1, π_2, σ, $\mathbb{I}$) and let P_1 and P_2 be two programs from respectively $\mathcal{L}_1$ and $\mathcal{L}_2$. If P_1 and P_2 can be compared with O, i.e., if there exists an initial configuration such that $o_i = \sigma(P_1, P_2)$, then the execution trace of O realizes a γ-correlation such that:
$$\gamma(s_1, s_2) = \exists o, \mathbb{I}(o) \wedge \pi_1(o) = s_1 \wedge \pi_2(o) = s_2$$

$$\begin{tikzcd}
s_l^1 \arrow[rr] & & s_l^2 \arrow[rr] & & s_l^3 \arrow[rr] & & s_l^4 \arrow[rr] & & s_l^5 & & \cdots \\
o_1 \arrow[u, "\pi_L"'] \arrow[d, "\pi_R"] \arrow[rrr, "\mathbb{S}"] & & & o_2 \arrow[ul, "\pi_L"] \arrow[dr, "\pi_R"'] \arrow[r, "\mathbb{S}"] & o_3 \arrow[u, "\pi_L"'] \arrow[d, "\pi_R"] \arrow[rrr, "\mathbb{S}"] & & & o_4 \arrow[ur, "\pi_L"] \arrow[dl, "\pi_R"'] & & & \\
s_r^1 \arrow[rr] & & s_r^2 \arrow[rr] & & s_r^3 \arrow[rr] & & s_r^4 \arrow[rr] & & s_r^5 & & \cdots
\end{tikzcd}$$

Figure 3.6: A correlating oracle realizes a γ-correlation.

The Figure 3.6 illustrates this theorem.

To sum up, each oracle language encapsulate a specific proof scheme for a specific class of traces. When we have two programs, it suffices to call σ to determine if they can be related by a given semantic differences.

3.2 Verifiable difference language for Imp

We instantiate our generic framework on the Imp programming language. To exercise the expressivity of the framework, we define several classes of semantic differences whose union gives us a language for semantic differences for Imp. The syntax of this difference language is given in Figure 3.7. A difference δ can be either a primitive difference δ_p, a composition of two differences $\delta ; \delta$ or a superposition of two differences $\delta \,\&\, \delta$.

The syntax for primitive differences enumerates a collection of *builtin* differences. This choice of primitives is *adhoc* and there is no guarantee that they match all the software changes that can happen in a real software development. Nevertheless, we define four categories of changes, depending on their level of abstraction. The two categories named *syntactic refactorings* and *syntactic changes* contain primitive differences that can be expressed as program transformations. The next two categories, *extensional changes* and *abstract changes* relate two programs by exploiting a proof that a specific relation holds between their configurations during the evaluation.

Syntactic refactorings are program transformations that preserve the semantics of the source program for a given notion of program equivalence. They include (i) any renaming with respect to a bijection between their variable identifiers, (ii) any swap between two consecutive independent assignments found at a program point characterized by a context $\mathbb{C}$ and (iii) any swap between the branches of a conditional statement provided that the condition of this statement is negated in the target program.

Syntactic changes are program transformations that modify the meaning of the source program. The difference **fix off-by-one at** $\mathbb{C}$ is a program transformation that applies to a `while`-loop whose last iteration crashes and that modifies its condition to avoid this last buggy iteration. The difference **fix with defensive condition at** $\mathbb{C}$ is a local program transformation that inserts a conditional statement at a source program location characterized by the context $\mathbb{C}$ which precedes a statement c that triggers a crash. This conditional statement makes the evaluation of the target program avoid the evaluation of the statement c. The difference **change values of** $\overline{x}$ is a program transformation which modifies the assignments of any variable in $\overline{x}$ in the source program provided that this variable has no influence on the control flow.

Extensional changes are modifications of the source program that are not necessarily instances of a program transformation but for which a proof can be exploited to show that a specific relation holds between the two programs' traces. The difference **ensure equivalence at** $\mathbb{C}$ relates two equivalent subprograms located at context $\mathbb{C}$ in the two programs. This difference exploits a proof of extensional equivalence to show that the target program is a refactoring of the source program. The difference **assume** $\{P\}$**ensure**$\{Q\}$ relates two programs for which a relational Hoare logic proof validates the precondition P and the postcondition Q. As said in the introduction, by referring to the variables of both programs in P and Q, such a proof establishes that a specific relation holds between the reduction traces of two *a priori* inequivalent programs. This last difference language is also a constructive proof that our framework is at least as expressive that Relational Hoare Logic.

The category of abstract changes corresponds to a class of generic differences that allows us to have an interoperability between our oracle-centric framework and proofs about trace relations that are not based on oracles. These difference languages are based on general definition like the following.

$$
\begin{array}{lcll}
 & & & \boxed{\textsc{Composite differences}} \\
\delta & ::= & \delta_p & \text{Primitive} \\
 & | & \delta ; \delta & \text{Composition} \\
 & | & \delta \,\&\, \delta & \text{Superposition} \\
 & & & \\
 & & & \boxed{\textsc{Primitive differences}} \\
\delta_p & ::= & & \underline{\textit{Syntactic refactorings}} \\
 & | & \textbf{rename}\,\overline{x \leftrightarrow y} & \\
 & | & \textbf{swap assign at } \mathbb{C} & \\
 & | & \textbf{swap branches at } \mathbb{C} & \\
 & & & \\
 & & & \underline{\textit{Syntactic changes}} \\
 & | & \textbf{fix off-by-one at } \mathbb{C} & \\
 & | & \textbf{fix with defensive condition at}\,\mathbb{C} & \\
 & | & \textbf{change values of } \overline{x} & \\
 & & & \\
 & & & \underline{\textit{Extensional changes}} \\
 & | & \textbf{ensure equivalence at } \mathbb{C} & \\
 & | & \textbf{assume}\,\{P\}\textbf{ensure}\{Q\} & \\
 & & & \\
 & & & \underline{\textit{Abstract changes}} \\
 & | & \textbf{refactor with respect to } \gamma & \\
 & | & \textbf{crash fix} & \\
 & | & \textbf{optimize} & \\
\end{array}
$$

Figure 3.7: Syntax of ΔImp.

Definition 15 (γ-refactoring)
Let γ be an equivalence relation over configurations. A difference δ is a *refactoring* with respect to γ if for every pair of programs (p_1, p_2) for which δ is sound, $\mathcal{T}(p_1)$ and $\mathcal{T}(p_2)$ are γ-correlated.

With that definition of refactoring, we expect to capture meaning-preserving transformations applied to converging, diverging and stuck programs. For instance, if a γ-correlation ρ only relates the final configurations of converging programs with respect to the equivalence of stores, then ρ corresponds to the usual observational equivalence. With that notion, one can also capture transformations that preserve bugs or transformations that turn an infinite computation into another infinite computation which shares with it an infinite number of correspondences. In general, this notion of parameterized refactoring allows for the specification of "points of interest" which must be equivalent in the source program and its refactored version. Similar definitions can be exploited to show that a patch is indeed an optimization or a bugfix.

Definition 16 (Optimization)
A difference δ is an *optimization* (reducing execution time) if for every pair of programs (p_1, p_2) for which δ is sound, $\mathcal{T}(p_1)$ and $\mathcal{T}(p_2)$ are finite and the length of $\mathcal{T}(p_2)$ is smaller than the length of $\mathcal{T}(p_1)$.

Definition 17 (Crash fix)
A difference δ is a *crash fix* if for every pair of programs (p_1, p_2) for which δ is sound, $\mathcal{T}(p_1)$ is crashed and $\mathcal{T}(p_2)$ is not crashed.

4 Related work

We can split related work about relational analysis of programs into two families: on the one hand, we have lines of work about differential program analysis [23, 21, 22, 14] that try to *compute a relation* between two programs without *a priori* knowledge about the desired relation ; on the other and, the lines of work about *relational deductive verification* [4, 7] checks that a given relation is valid.

Differential static analysis A differential static analysis looks for *simple relations* between the variables of two programs. Typically, such an analysis can try to determine which variables are unchanged between the programs, or which variables are changed by a simple offset.

Once the relational domain is decided, the static analyzers use heuristics to determine the right scheduling of the compared programs instructions. The static scheduling builds a third program that statically schedules the two compared programs. As said earlier, this approach is often incomplete but allows existing static analyzers to be reused. Dynamic heuristics are more expressive but require a (at least partial) reimplementation of a static analyzer.

Relational deductive verification Relational proof systems are more expressive than differential static analysis: relational deduction rules seems to be able to prove any logically definable relation to hold between two programs[4]. These deductive systems [7] are used in a similar way as usual proof systems: the syntaxes of the two programs and the relational property to prove are decomposed thanks to inference rules that introduce new proof goals until axioms are reached.

However, as said earlier, the construction of the proof derivation is here potentially directed by two concurrent syntactic processes: on the one hand, we find the usual decomposition of commands to be verified ; on the other hand, there is also a search for the right scheduling of the instructions of the two programs.

Following these two reasoning processes simultaneously is not easy, and that's why Barthe, Crespo, and Kunz [2] suggest to separate these two processes into two distinct reasoning steps. The first step builds a *product program* that statically schedules the two programs following a manually defined strategy. The second phase proves the validity of the relation by reasoning about the product program using traditional deductive proof systems. Again, this technique allows existing provers to apply.

[4] As far as we know, there is however no relative completeness results about existing relational proof systems, hence the usage of "seems to" in that sentence.

5 Conclusion and Future Work

Certifying software evolutions requires new theoretical and practical tools for relational program analysis. With the Ph.D. of Thibaut Girka, we have some strong arguments in favor of γ-correlations because they seem to provide an expressive enough theoretical framework to specify semantic differences occurring during software development. Correlating oracles offer proof schemes for a large class of trace relations.

Our long-term goal is to certify the evolutions of Coq developments, typically to characterize the impact of changes in definitions on the theorem statements and proofs. Before that, many problems are to be solved. We first envision to attack the following two research projects.

Verifying Beck's book about Test-Driven Development In collaboration with David Mentré and Kostia Chardonnet, we are defining a difference language for the large subset of Java used in the famous software engineering book of Kent Beck entitled "Test-Driven Development by Example" [5]. We are specifying and verifying the examples of the book as a new use case for our framework. In particular, we want to check the scalability of our approach on a language that is more realistic than Imp.

Type-directed semantic differences for typed functional programming languages When a typed functional programmer modifies a type definition, the typechecker highlights all the places where the code must be updated. That behavior is probably one of the most appreciated feature of typed functional languages. Sometimes, this type-directed process is incorrectly called a refactoring while it does not preserve the semantics of the program. A language of semantic differences for that specific form of type-directed source change could better specify their effects on the program behavior. Besides, the verification of these semantic differences could profit from the typechecker to discharge some proof obligations.

References

[1] Alberto Bacchelli and Christian Bird. "Expectations, outcomes, and challenges of modern code review". In: *Proceedings of the 2013 international conference on software engineering*. IEEE Press. 2013, pp. 712–721.

[2] Gilles Barthe, Juan Manuel Crespo, and César Kunz. "Beyond 2-safety: Asymmetric product programs for relational program verification". In: *International Symposium on Logical Foundations of Computer Science*. Springer. 2013, pp. 29–43.

[3] Gilles Barthe, Juan Manuel Crespo, and César Kunz. "Product programs and relational program logics". In: *J. Log. Algebr. Meth. Program.* 85.5 (2016), pp. 847–859.

[4] Gilles Barthe et al. "Coupling proofs are probabilistic product programs". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 2017, pp. 161–174.

[5] Kent Beck. *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam, 2002.

[6] 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 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. ACM, 2004, pp. 14–25.

[7] Nick Benton. "Simple Relational Correctness Proofs for Static Analyses and Program Transformations (Revised, Long Version)". In: (2005).

[8] *Circle-ci*. `http://circle-ci.org`.

[9] Anne Edmundson et al. "An empirical study on the effectiveness of security code review". In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2013, pp. 197–212.

[10] Jean-Christophe Filliâtre and Andrei Paskevich. “Why3 - Where Programs Meet Provers”. In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 125–128.

[11] Thibaut Girka, David Mentré, and Yann Régis-Gianas. “A Mechanically Checked Generation of Correlating Programs Directed by Structured Syntactic Differences”. In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2015, pp. 64–79.

[12] Thibaut Girka, David Mentré, and Yann Régis-Gianas. “Verifiable semantic difference languages”. In: *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017*. Ed. by Wim Vanhoof and Brigitte Pientka. ACM, 2017, pp. 73–84.

[13] *GitLab*. http://auto.gitlab.com.

[14] Chris Hawblitzel et al. *Mutual Summaries: Unifying Program Comparison Techniques*. Tech. rep. 08/2011.

[15] James Wayne Hunt. *An algorithm for differential file comparison*.

[16] Claire Le Goues, K. Rustan M. Leino, and Michal Moskal. “The Boogie Verification Debugger (Tool Paper)”. In: *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*. Ed. by Gilles Barthe, Alberto Pardo, and Gerardo Schneider. Vol. 7041. Lecture Notes in Computer Science. Springer, 2011, pp. 407–414.

[17] K. Rustan M. Leino. “Program proving using intermediate verification languages (IVLs) like boogie and why3”. In: *Proceedings of the 2012 ACM Conference on High Integrity Language Technology, HILT '12, December 2-6, 2012, Boston, Massachusetts, USA*. Ed. by Ben Brosgol, Jeff Boleng, and S. Tucker Taft. ACM, 2012, pp. 25–26.

[18] Shane McIntosh et al. “The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM. 2014, pp. 192–201.

[19] Sebastiano Panichella et al. “Would static analysis tools help developers with code reviews?” In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2015, pp. 161–170.

[20] Nimrod Partush and Eran Yahav. “Abstract Semantic Differencing for Numerical Programs”. English. In: *Static Analysis Symposium*. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 238–258.

[21] Nimrod Partush and Eran Yahav. “Abstract semantic differencing via speculative correlation”. In: *ACM SIGPLAN Notices*. Vol. 49-10. ACM. 2014, pp. 811–828.

[22] Suzette Person et al. “Differential symbolic execution”. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM. 2008, pp. 226–237.

[23] Chris Hawblitzel Shuvendu Lahiri Kenneth McMillan. “Differential Assertion Checking”. In: ACM, 08/2013.

[24] *The Most Frequent Commit Messages on GitHub are Mostly Useless*.

[25] *Travis-ci*. http://travis-ci.org.

CHAPTER 4

Co-recursive programming

Eternity is a long time, especially towards the end.

An infinite list of authors.

Contents

Related publications:

- *Copattern matching and first-class observations in OCaml, with a macro.* **PPDP (2017)** with Paul Laforgue.

1 Introduction

Context Algebraic datatypes and pattern matching are useful mechanisms especially to specify and to define programs that work on finite and inductive objects like lists or trees. The functional programmer appreciates the mathematical flavor and the computational simplicity of these mechanism because they are really good ingredients for a safe and efficient programming.

In OCaml, when we come to infinite objects, the picture is less rosy. Let us take the case for infinite sequences and let us try to define the sequence of natural numbers:

```
let rec from : int -> int list = fun n -> n :: from (succ n)
let naturals = from 0
let rec nth : int -> `a list -> `a = fun n s -> match n with
  | 0 -> List.hd s
  | m -> nth (pred m) (List.tl s)
```

As OCaml is a strict language, the evaluation of `naturals` provokes a divergence because the program tries to build all natural numbers in a single step. By contrast, we would prefer if the program would produce these numbers lazily, i.e., only on demand, when a computation really needs to *observe* a finite fragment of this infinite sequence.

In Haskell, programs are lazy by default but in OCaml, laziness must be explicitly programmed. Using the keyword `lazy`, a programmer can systematically delay the construction of infinite sequences, and can invoke `Lazy.force` to compute the first elements of this sequence, one-by-one, only when it is strictly required:

```
type `a stream = (`a cell) Lazy.t and `a cell = Cell of `a * `a stream
let rec from : int -> int stream = fun n -> lazy (Cell (n, from (succ n)))
let naturals = from 0
let rec nth : int -> `a stream -> `a = fun n s ->
  let Cell (hd, tl) = Lazy.force s in
  match n with
  | 0  -> hd
  | m  -> nth (pred m) tl
```

Although the types and the operations for lazy computations allow computing over infinite structures, they obfuscate the definitions because they introduce low-level aspects about the order of computations. These aspects are absent from the usual mathematical definitions: a high-level language should allow us to omit them!

Copattern matching [2] is a generalization of pattern matching. In OCaml equipped with copatterns[1], an infinite object is defined with no particular precaution with respect to divergence:

```
1   type 'a stream = {Head: 'a; Tail: 'a stream}
2   let corec from : int -> int stream with
3     | (.. n)#Head -> n
4     | (.. n)#Tail -> from (succ n)
5   let naturals = from 0
6   let rec nth : int -> 'a stream -> 'a = fun n s -> match n with
7     | 0 -> s#Head
8     | m -> nth (pred m) s#Tail
```

Contrary to algebraic datatype which are defined by constructors, coalgebraic datatypes are defined by destructors, also called observations. On Line 1, the coalgebraic datatype 'a stream has two observations: Head which produces a value of type 'a and Tail which produces a value of type 'a stream.

On Line 2, from is defined by copattern matching. To that end, the programmer reason by cases over the different observations, Head and Tail, that can be made on from n. In a copattern, the notation .. refers to the observed infinite object. The identifier starting with an uppercase letter Obs is the observation that is under definition in the current branch. Hence, Line 3 is read:

"To observe the head of from n, returns n."

and Line 4 is read:

"To observe the tail of from n, returns from (succ n)."

Notice that these two branches do not share the same type contrary to the branches of a standard pattern matching. Indeed, the first branch has type 'a while the second has type 'a stream.

Only observation applications can trigger computations over infinite structures. We can find such applications in the definition for nth: s#Head triggers the computation of the head of s and s#Tail the computation of the tail of s.

Copattern-matching is a high-level construction to define infinite objects and it comes with equational reasoning principles that are immediately deduced from the code. How can we implement this beautiful mechanism in our favorite functional programming language?

Contributions With Paul Laforgue, we show that copattern-matching can be integrated in any functional programming language with generalized algebraic datatypes and with second-order polymorphism. This integration is done by a purely local transformation, i.e., a macro. As a by-product of our encoding of copatterns in OCaml, we give to observations the status of first class citizens. This opens new programming opportunities.

2 Transformation

As soon as a functional programming language is equipped with generalized algebraic datatypes [14] and with second-order polymorphism, we can extend its pattern matching as a copattern matching with a purely syntactic and local transformation [8]. In a nutshell, this transformation translates

[1] Our prototype can be installed using opam with the compiler switch 4.04.0+copatterns.

every coalgebraic value defined by a copattern matching as a function defined by case on the shape of the observations of these coalgebraic datatype. In this section, we present this result informally by describing it using examples.

The translation of the type declaration starts with the introduction of a new type `stream_obs` corresponding to observations of the type `stream`:

```
type ('a, 'o) stream_obs =
| Head : ('a, 'a) stream_obs
| Tail : ('a, 'a stream) stream_obs
```

This type owns two constructors: `Head` and `Tail`. The first type parameter `'a` simply corresponds to the type parameter of `stream`, i.e., the type of its elements. The second parameter `'o` is more interesting: it corresponds to the return type of observations. Notice that the two constructors instantiate differently `'o`: for `Head`, `'o` is `'a` whereas for `Tail`, `'o` is `'a stream`. This variability of constructor type schemes is a characteristic of GADTs.

The coalgebraic datatype `stream` is translated as an algebraic datatype with a single constructor named `Stream` by convention. This constructor expects a function as argument. This function takes an observation as input and is polymorphic with respect to the return type of this observation. This polymorphism occurs on the left of an arrow in the type scheme for `Stream`, which is therefore using second-order quantification over types. In OCaml, such second-order polymorphism must be introduced using a record type with a polymorphic field. In the end, we get the following definition:

```
and 'a stream = Stream of { dispatch : 'o. ('a,'o) stream_obs -> 'o }
```

Here, the `dispatch` function has the polymorphic type `forall 'a. ('a, 'o) stream_obs -> 'o`. This type, which is reminiscent of a CPS translated type, witnesses a control inversion between the evaluation environment and the coalgebric value.

Naturally, we translate copattern matching by following closely the translation of type declarations. When a copattern matching is translated, we systematically introduce a function defined by pattern matching that has the same structure as this copattern matching.

```
let rec from : int -> int stream = fun n ->
  let dispatch : type o.(int, o) stream_obs -> o = function
    | Head -> n
    | Tail -> from (succ n)
  in Stream {dispatch}
```

Notice that this function is well-typed because (i) in the first branch `o` and `int` are equivalent and therefore `n` has the expected type `o` ; (ii) in the second branch `o` and `int stream` and equivalent so `from (succ n)` which has type `int stream` also has the expected type `o`.

Last ingredient for our translation: the case for observation applications. They are mere applications of the `dispatch` function. By introducing,

```
let head {dispatch} = dispatch Head
let tail {dispatch} = dispatch Tail
```

we can translate `s##Head` into `head s`.

3 First order and first class destructors

In the previous section, we have shown that for each coalgebraic datatype declaration, our transformation introduces a GADT to represent its observations. The definition of this GADT respects a naming convention which is documented. The programmer can therefore explicitly refer to these constructors and this GADT in the source code. This way, destructors can be used in the program as any other value. Besides, since they are represented by mere data constructors, they can be compared and serialized.

To illustrate this mechanism, consider the following scenario. The programmer defines a record type `loc` with three fields (`name` of type `string`, an abscissa `x` and an ordinate `y` of type `int`) and the programmer would like to build projection and field update functions. Without copattern matching, this task forces some code repetition:

```
 type loc = {name: string; x : int; y : int}
 let select_name  lc = lc.name and update_name  s lc = {lc with name = s}
 let select_x     lc = lc.x    and update_x     b lc = {lc with x = b}
 let select_y     lc = lc.y    and update_y     n lc = {lc with y = n}
```

This problem comes from the fact that labels are not first class object in OCaml. We cannot write generic combinators like `select` and `update` capable of accepting a label as argument.

Let us consider now the same scenario but in which, this time, instead of a record type, we use a coalgebraic data type:

```
 type loc = {Name : string; X : int; Y: int}
```

The combinator `select` is then easily defined as it suffices to extract the `dispatch` function from a value of type `loc`, and to apply it to a destructor passed as argument:

```
 let select (d : 'a loc_obs) (Loc {dispatch} : loc) : 'a = dispatch d
```

Defining `update` requires more effort. As a first approximation, we could write:

```
let update (type a) (d1 : a loc_obs) (x : a) (Loc {dispatch}) =
   let dispatch : type o. o loc_obs -> o = fun d2 ->
        if d1 = d2 then x else dispatch d2
   in Loc {dispatch}
```

but this definition is ill-typed! First, `d1` and `d2` do not necessarily have the same type. Second, nothing informs the typechecker that `x` has the type `o`. Fortunately, we can easily write a richly-typed comparison function `eq_loc` [2] with a return type more precise than a simple boolean: in the positive case, this function returns a type equality proof between the indices of the two destructors:

```
type (_,_) eq = Eq : ('a,'a) eq
val eq_loc : type a b. a loc_obs * b loc_obs -> ((a,b) eq) option
```

The function `update` can now be fixed using `eq_loc`:

```
let update (type a) (d1 : a loc_obs) (x : a) (Loc {dispatch}) =
   let dispatch : type o. o loc_obs -> o = fun d2 -> match eq_loc (d1,d2) with
     | Some Eq -> x
     | _ -> dispatch d2
   in Loc {dispatch}
```

This function is well-typed since `eq_loc` accepts arguments whose types are different, and in the case `Some Eq`, the expression `x` is typed under a typing context enriched with the equality `a = o`, so it indeed has type `o`.

4 Related work

CoCaml [7] is an extension of the OCaml programming language with facilities to define functions over coinductive datatypes. We obviously share the same motivations as CoCaml: tackling the lack of support for infinite structures in this language. However, our approaches differ significantly. CoCaml only deals with *regular* coinductive datatypes, that is the subset of infinite values representable using cyclic values. By restricting itself to regular coinductive datatypes, CoCaml can reuse the ability of OCaml to define cyclic values. These representations are more efficient than ours since they are defined eagerly, not on demand. Another effect of this restriction is the opportunity for CoCaml to introduce a new construction `let corec[solver] f x = match x with ...` which transforms the pattern matching inside `f` into a set of equations which are subsequently solved using a `solver` specified by the programmer. This approach offers stronger guarantees than ours since the different solvers can check for instance that the defined computations over regular coinductive datatypes are terminating or preserve the regularity of infinite values.

We implement the same syntax as the original copatterns paper[2] but, even if we share the same syntax for full copattern matching, the operational semantics are different because we consider branches as a sequence, not as a set of equations. The system [1] introduces a notion of *generalized abstraction*

[2] Left as an exercise to the reader.

which is similar to the functions and corecursive functions of our source language. The paper of Setzer et al. [11] presents a program transformation to unnest deep copatterns into simple copatterns. We also do rewrite nested copatterns into simpler one but in a slightly different way because we want to stick with the ML pattern matching convention. From the typing perspective, our system is closer to the original system of indexed codata types[13].

To our knowledge, none of the existing languages with copatterns provides first-class first-order observation queries like ours and none of them studies the encoding of copattern matching in terms of pattern matching. Notice that our metatheoretical study is simpler than previous work about copatterns. Since we are extending OCaml and not a proof assistant like Coq [3] or Agda [9], we are looking for type safety and the correctness of our translation, rather than strong normalization or productivity. Recently, Downen et al. [5] explore other encodings for coalgebraic datatypes based on Church encodings. They do not provide first-class observations like us but they do not require GADTs.

5 Conclusion and Future Work

Copattern matching is an elegant extension of pattern matching to program with infinite data. We extended OCaml with copatterns with a mere local transformation. By lack of space, we did not present the advanced aspects of copattern matching, namely (i) indexed coalgebraic datatype which are the equivalent of GADTs in the coalgebraic world ; (ii) nested copatterns; (iii) lazy copattern-matching which allows to memoize observations. These mechanisms are also handled by our transformation. The curious reader can discover them in our paper [8].

We plan to continue working on this encoding of copatterns on two projects. The first one is the introduction of copatterns in Coq. The second one is about extending Higher-Order Hoare Logic to deductively reason about infinite objects using copatterns and first-class observations.

Copattern matching in Coq Equations [12] is a plugin which permits Agda-style dependently-typed programming in Coq. We plan to generalize the syntax of Equations' dependent patterns to include dependent copatterns. We already know that our transformation does not work as is within Coq because of a universe inconsistency. Indeed, the following definitions:

```
Definition codata A (obs : Type -> Type -> Type) :=
  forall B, obs A B -> B.

Fail Inductive obs A : Type -> Type :=
| Head : obs A A
| Tail : obs A (codata A obs).
```

is rejected by Coq because no universe can be assigned to the second parameter of `obs`. We plan to work around this limitation by following the same methodology as in the Paco library [6] about coinduction, that is, by reconstructing a theory about infinitary objects in Coq defined with first-class observations.

Reasoning about infinite programs Higher-Order Hoare Logic [10] is a program logic for functional programs equipped with pattern matching. We plan to extend this deductive proof system to handle copattern matching. This extension raises the question of well-formed infinitary proofs in presence of properties that mix inductive and coinductive predicates. The Ph.D. thesis of Amina Doumane [4] recently improves our understanding of this kind of logics, and we should be able to apply it to improve both the expressiveness of the specification logic and the verification techniques for copatterns-based programs.

References

[1] Andreas M Abel and Brigitte Pientka. “Wellfounded recursion with copatterns: A unified approach to termination and productivity”. In: *18th ACM SIGPLAN International Conference on Functional Programming*. Vol. 48. 9. ACM. 2013, pp. 185–196.

[2] Andreas Abel et al. “Copatterns: programming infinite structures by observations”. In: *42nd ACM SIGPLAN conference on Principle of Programming Languages*. Vol. 48. 1. ACM. 2013, pp. 27–38.

[3] Coq Development Team. *The Coq Proof Assistant Reference Manual.* INRIA-Rocquencourt, 2012.

[4] Amina Doumane. “On the infinitary proof theory of logics with fixed points. (Théorie de la démonstration infinitaire pour les logiques à points fixes)”. PhD thesis. Paris Diderot University, France, 2017.

[5] Paul Downen et al. “Codata in Action”. In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings.* Ed. by Luís Caires. Vol. 11423. Lecture Notes in Computer Science. Springer, 2019, pp. 119–146.

[6] Chung-Kil Hur et al. “The power of parameterization in coinductive proof”. In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’13, Rome, Italy - January 23 - 25, 2013.* Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 193–206.

[7] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. “CoCaml: Functional Programming with Regular Coinductive Types”. In: *Fundamenta Informaticae* 150 (2017), pp. 347–377.

[8] Paul Laforgue and Yann Régis-Gianas. “Copattern matching and first-class observations in OCaml, with a macro”. In: *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017.* Ed. by Wim Vanhoof and Brigitte Pientka. ACM, 2017, pp. 97–108.

[9] Ulf Norell. “Dependently Typed Programming in Agda”. In: *4th International Workshop on Types in Language Design and Implementation.* TLDI ’09. Savannah, GA, USA: ACM, 2009, pp. 1–2.

[10] Yann Régis-Gianas and François Pottier. “A Hoare Logic for Call-by-Value Functional Programs”. In: *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings.* Ed. by Philippe Audebaud and Christine Paulin-Mohring. Vol. 5133. Lecture Notes in Computer Science. Springer, 2008, pp. 305–335.

[11] Anton Setzer et al. “Unnesting of copatterns”. In: *International Conference on Rewriting Techniques and Applications.* Springer. 2014, pp. 31–45.

[12] Matthieu Sozeau. “Equations: A Dependent Pattern-Matching Compiler”. In: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings.* Ed. by Matt Kaufmann and Lawrence C. Paulson. Vol. 6172. Lecture Notes in Computer Science. Springer, 2010, pp. 419–434.

[13] David Thibodeau, Andrew Cave, and Brigitte Pientka. “Indexed codata types”. In: *21st ACM SIGPLAN International Conference on Functional Programming.* ACM. 2016, pp. 351–363.

[14] Hongwei Xi, Chiyan Chen, and Gang Chen. “Guarded recursive datatype constructors”. In: *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003.* Ed. by Alex Aiken and Greg Morrisett. ACM, 2003, pp. 224–235.

CHAPTER 5

Cost annotating compilation

All magic comes with a price.

Rumpelstilskin

Contents

Related publications:

- *Certified complexity (CERCO).*

  FOPARA (2013) with Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli.

- *Synthesizing* CerCo*'s cost annotations: Lustre case study.*

  University Paris Diderot (2013) with Roberto Amadio and Nicolas Ayache.

- *Certifying and reasoning on cost annotations of functional programs.*

  HOSC (2013) with Roberto M. Amadio.

- *Certifying and reasoning on cost annotations in C programs.*

  FMICS (2012) with Nicolas Ayache and Roberto M. Amadio

- *Certifying and reasoning on cost annotations of functional programs.*

  FOPARA (2011) with Roberto M. Amadio

1 Introduction

Context CompCert is a mechanically verified compiler for a large subset of a C compiler. The theorem of semantic preservation proved by Leroy [3] ensures that the behaviors of the compiled code includes the behaviors of the source code plus some extra behaviors that are left unspecified by the semantics of the source language. This result significantly increases the value of certified C programs since CompCert guarantees that the proofs also hold on the compiled code. This property is especially important to guarantee several safety properties and even functional correctness of critical embedded systems since they are commonly implemented using the C language.

The safety of an embedded system may also depend on our ability to properly allocate computational resources for their execution. A typical example is the case of C programs produced by compilers for synchronous programming languages like ESTEREL or LUSTRE : these programs usually consist of a single step function which is supposed to be executed at a well-specified frequency to react to some input signals under strict temporal constraints. Even if static analyzers are able to produce precise over-approximations of the Worst-Case Execution Time (WCET) of binaries, these bounds are difficult to relate to the source programs and are therefore hard to exploit for the programmer. Transporting this cost information as annotations in the source program, provided that these cost annotations are correct of course, is an improvement of this situation. That is the purpose of a cost annotating compiler.

The CERCO european project, coordinated at University Paris Diderot by Roberto Amadio, aims at providing a certified cost annotating compiler for the C language. Unfortunately, in that case, COMPCERT's existing proof of semantic preservation is not directly applicable since it only characterizes the observable behaviors in terms of the source language semantics. For this reason, a new proof technique is needed to justify the correctness of cost annotations.

Contributions In collaboration with Roberto Amadio and Nicolas Ayache, we propose an architecture for cost annotating compilers as well as a proof technique to modularly extend a proof of semantic preservation to show the validity of cost annotations. As we will see in Section 2, this technique crucially relies on a so-called labelling method, that constructs a mapping between the execution paths of the source program and the ones of the compiled program through the compilation passes.

In Section 3, we describe the implementation of a prototype cost annotating compiler for the C language and we provide evidence of the usability of the generated cost annotations through their put into use in a FRAMA-C plugin to automatically infer trustworthy logical assertions about the concrete worst case execution cost of programs. These logical assertions are synthetic in the sense that they characterize the cost of executing the entire program, not only constant-time fragments, and may depend on the size of the input data.

Finally, a third contribution is the application of this labelling method to another compilation chain, this time dedicated to the compilation of a functional language.

2 Certifying cost annotations in compilers

A cost annotating compiler takes as input a source program and returns an instrumented version of this program in addition to the usual compiled code. This instrumented program is not meant to be executed but to reason about the resources consumed by the compiled program execution. The instrumentation simply consists in introducing a global variable for each resource for which we have a cost model in the target language, and in inserting increments of these global variables at some well-chosen program locations. The instrumented source program is thus equivalent to the source program except that it self-monitors resource consumption. The instrumentation of a simple C program is given in Figure 5.1. In that example, the C program is compiled to the Intel 8051 microprocessor [10] for which we have a specification of the exact number of cycles consumed by each assembly instruction. Therefore the integer numbers used in increments are exactly the number of cycles consumed so far (minus a constant because as we shall see, the value of the global variable is always incremented a bit earlier with respect to the reality).

We first define what it means for an instrumentation to be sound and precise.

Definition 18 (Instrumentation soundness)
An instrumentation is sound if the value of each global variable is an upper bound for the actual cost it predicts.

Definition 19 (Instrumentation precision)
An instrumentation is precise if the difference between the actual cost and the predicted cost is bounded by a constant that only depends on the program.

The next question is of course how do we generate sound and precise cost annotations. As the cost model is defined on the target language, the instrumentation cannot happen before the compiled code is generated. Yet, at this point of the compilation, we have lost the precise mapping between the source program and the compiled code. To recover this information, we propose to extend each intermediate language of the compiler with labels. The compiler inserts these labels at strategic points of the source program and conveys them through each pass. Once we reach the target language, we associate a cost

```
int is_sorted (int *tab, int size) {
  int i, res = 1;
  for (i = 0 ; i < size-1 ; i++)
     if (tab[i] > tab[i+1]) res = 0;
  return res;
}
```

```
int cost = 0;
int is_sorted (int *tab, int size) {
  int i, res = 1;
  cost += 97;
  for (i = 0; i < size-1; i++) {
    cost += 91;
    if (tab[i] > tab[i+1]) {
      cost += 104; res = 0;
    }
    else cost += 84;
  }
  cost += 4; return res;
}
```

Figure 5.1: The instrumentation of a C program. (On the right.)

to each label. This is the cost of every execution path starting at that label and ending just before the execution of another labelled instruction, or the end of the program. An instrumentation function $\mathcal{I}$ then replaces the labels in the source program with the proper increments as explained earlier.

For this cost to be well-defined, all the execution paths from each label must be composed of a finite sequence of instructions. Besides, to be sound, the cost of a label ℓ must be the maximal cost of all the execution paths starting from ℓ. To be precise, all the execution paths from ℓ must cost the same. These requirements constrain the labelling function $\mathcal{L}$ of the compiler to insert a label inside each loop to be sound by avoiding infinite execution paths, and also at the beginning of each branch of conditional instructions to guarantee precision.

Extending the intermediate languages and the compilation passes with labels is immediate. The operational semantics of each intermediate languages now produces a sequence λ of labels which corresponds to the labels encountered during the execution (in addition to their usual outcomes).

Can we extend the compiler proof of correctness as easily? By taking compositionality seriously, we devise a proof technique that allows for a modular extension of the compiler proof of correctness. The idea of this proof technique is illustrated by the following diagram:

$$\begin{tikzcd}
L_1^\ell \arrow[r, "C_1^\ell"] \arrow[d, bend right, "\mathcal{I}"'] \arrow[d, bend left, "\mathcal{E}_1"] & L_2^\ell \arrow[r, "C_2^\ell"] \arrow[d, bend left, "\mathcal{E}_2"] & \cdots & L_k^\ell \arrow[r, "C_k^\ell"] \arrow[d, bend left, "\mathcal{E}_k"] & L_{k+1}^\ell \arrow[d, bend left, "\mathcal{E}_{k+1}"] \\
L_1 \arrow[u, "\mathcal{L}"] \arrow[r, "C_1"] & L_2 \arrow[r, "C_2"] & \cdots & L_k \arrow[r, "C_k"] & L_{k+1}
\end{tikzcd}$$

Each intermediate language L_i has an extension with labels that we name L_i^ℓ ,and for each extension we have a label erasure function $\mathcal{E}_i$ such that

$$\mathcal{E}_{i+1} \circ C_i^\ell = C_i \circ \mathcal{E}_i$$

Besides, the labelling function $\mathcal{L}$ must be a right inverse of $\mathcal{E}_1$, i.e. $\mathcal{E}_1 \circ \mathcal{L} = id_{L_1}$.

With that diagram in mind, we simply have to extend an existing correctness proof for a compiler defined by $C = C_1 \circ \ldots \circ C_{k+1}$ to deal with labels and then show that this extension commutes with the label erasure functions. Then, by diagram chasing, we get the following theorem.

Theorem 11

If cost $\mapsto c \in M$, $M \vdash \mathcal{I}(\mathcal{L}(P)) \Downarrow M'$ and cost $\mapsto c + \delta \in M'$, then $M \vdash C^\ell(\mathcal{L}(P)) \Downarrow_\ell (M'', \lambda)$ and cost $\mapsto c \in M''$ where λ costs δ.

Besides, the diagram commutation and the correctness of the compilation functions allow us to conclude that the erasure of $C^\ell(\mathcal{L}(P))$ is semantically equivalent to $C(P)$. This result shows a form of non interference between the labelling process and the original compilation.

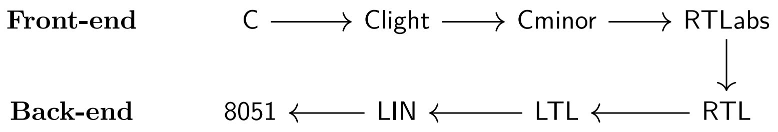


Figure 5.2: The compilation chain of the CerCo certified compiler.

```
int cost = 0;
/*@ ensures (cost <= old(cost) + (101 + (0 < size - 1 ? (size - 1) * 195 : 0))); */
int is sorted (int *tab, int size) {
  int i, res = 1, cost tmp0;
  cost += 97; cost tmp0 = cost;
  /*@ loop invariant (0 < size - 1) -> (i <= size - 1);
    @ loop invariant (0 >= size - 1) -> (i == 0);
    @ loop invariant (cost <= cost tmp0 + i * 195);
    @ loop variant (size - 1) - i; */
  for (i = 0; i < size - 1; i++) {
    cost += 91;
    if (tab[i] > tab[i + 1]) { cost += 104; res = 0; }
    else cost += 84;
  }
  cost += 4; return res;
}
```

Figure 5.3: A cost annotated function generated by the Frama-C plugin.

3 CerCo, a cost annotating compilers for a subset of C

Cost annotating compiler The formalization of the labelling method is done on the simple imperative language Imp. We conduct a larger experiment on a prototype compiler for the C language for understand if the approach scales well. Its architecture, depicted in Figure 5.2, is largely inspired from the CompCert compiler except that it targets the Intel's 8 bits microprocessor called 8051, and that the intermediate languages have small differences with respect to the language with the same name in CompCert. One advantage of following closely the architecture of CompCert is to simplify a possible integration. This compilation chain has later been formally checked using the Matita proof assistant by our colleagues of the University of Bologna and Edinburgh.

The labelling extension of Clight and Cminor introduces labelled statements and also labelled expressions. We have to label expressions because of ternary expressions since they branch and can therefore endanger cost precision. The next assembly-like languages, from RTLabs to 8051, are equipped with the instruction "emit ℓ" which does nothing except the observation that a label is encountered.

The labelling of Clight programs poses only few difficulties with respect to Imp's. First, as said earlier, each branch of ternary expressions is labelled to stay precise. Second, each goto label is labelled by a cost label: this ensures soundness because any potential loop iteration realized by a goto will have to cross this cost label. Third, we notice that function calls could be a source of unsoundness, typically because of recursion. However, by maintaining the invariant that every function's implementation contains at least one label, we know that the execution of a function call, even to a statically unknown function, will eventually meet a label before returning. This is also sufficient to get soundness.

Inference of synthetic cost annotations The cost annotations inserted by the CerCo compiler characterize the cost of constant-time sequence of instructions. Of course, the global cost of a function of argument x may depend on x. To some extent, an abstract interpreter on a symbolic domain can compute an expression for this global cost even though it depends on the values of program variables. Our Frama-C plugin follows such a strategy, that we cannot detail by lack of space. The Figure 5.3 shows the synthetic cost annotation inferred for the instrumented C program of Figure 5.1.

$$\begin{tikzcd}
\lambda^{\mathcal{M}} & \lambda^{\ell} \arrow[l, "\mathcal{I}"] \arrow[r, "C^{\ell}_{cps}"] \arrow[d, "\mathcal{E}"] & \lambda^{\ell}_{cps} \arrow[r, bend left, "C^{\ell}_{vn}"] \arrow[d, "\mathcal{E}_{cps}"] & \lambda^{\ell}_{cps,vn} \arrow[l, bend left, "\mathcal{R}"] \arrow[r, "C^{\ell}_{cc}"] \arrow[d, "\mathcal{E}_{vn}"] & \lambda^{\ell}_{cc,vn} \arrow[r, "C^{\ell}_{vn}"] \arrow[d, "\mathcal{E}_{cc}"] & \lambda^{\ell}_{h,vn} \arrow[d, "\mathcal{E}_{h}"] \\
 & \lambda \arrow[r, "C_{cps}"] & \lambda_{cps} \arrow[r, bend left, "C_{vn}"] & \lambda_{cps,vn} \arrow[l, bend left, "\mathcal{R}"] \arrow[r, "C_{cc}"] & \lambda_{cc,vn} \arrow[r, "C_{vn}"] & \lambda_{h,vn}
\end{tikzcd}$$

Figure 5.4: A compilation chain for a higher-order functional language.

$$\begin{array}{rcl}
\mathcal{L}(t) & = & \mathcal{L}_0(t) \\
\mathcal{L}_i(x) & = & x \\
\mathcal{L}_i(\lambda x^+.t) & = & \lambda x^+.\ell > \mathcal{L}_1(t) \\
\mathcal{L}_i(t(u^+)) & = & \begin{cases} \mathcal{L}_0(t)(\mathcal{L}_0(u^+)) > \ell & \text{if } i = 0 \text{ and } \ell \text{ fresh.} \\ \mathcal{L}_0(t)(\mathcal{L}_0(u^+)) & \text{if } i = 1 \end{cases} \\
\mathcal{L}_i(t^+) & = & (\mathcal{L}_i(t))^+ \\
\mathcal{L}_i(\pi_k(t)) & = & \pi_k(\mathcal{L}_0(t)) \\
\mathcal{L}_i(\mathbf{let}\ x = t\ \mathbf{in}\ u) & = & \mathbf{let}\ x = \mathcal{L}_0(t)\ \mathbf{in}\ \mathcal{L}_i(u)
\end{array}$$

Figure 5.5: A sound and precise labelling of λ-term.

4 A cost annotating compiler for a functional language

Cost annotating compiler We apply the labelling method to another compiler, this time for a functional programming language. More precisely, we concentrate our formal study on an untyped call-by-value λ-calculus with polyadic functions and tuples. In a prototype implementation of this compiler, we further extend the language with recursion and pattern-matching.

The compilation chain is described by a diagram given in Figure 5.4. The global structure of the chain follows the requirements of the labelling approach: the base compilation chain at the bottom of the diagram is again extended by a compilation on labelled intermediate languages. The compiler is the composition of the following standard translations: CPS translation, translation to A-normal form, closure conversion and hoisting. The target language of the compiler is close to the intermediate language RTLabs of CerCo and be further compiled to assembly code by the back-end of CerCo.

How should we label a λ-term to get a sound and precise labelling with respect to this compilation chain? We could apply blindly the same technique as for C by only inserting a label at the beginning of each λ-abstraction body. However, that labelling is not sufficient for soundness. Indeed, the CPS translation introduces new λ-abstractions that encode the continuation of subterms that are not necessarily λ-abstractions themselves. Typically, the CPS translation of an application which is not immediately surrounded by an abstraction will introduce such a new λ-abstraction. Besides, a given subterm's continuation is by definition evaluated after this subterm. For this reason, in addition to the term labelling construction $\ell > t$, we need a construction $t > \ell$ for a term t *post-labelled* by a label ℓ. A post-labelled term t emits a label after the evaluation of t. In the end, the syntax for terms of λ^{ℓ} is:

$$t \quad ::= \quad x \mid \mathbf{let}\ x = t\ \mathbf{in}\ t \mid \lambda x^+.t \mid t\,(t^+) \mid (t+) \mid \pi_i(t) \mid \ell > t \mid t > \ell$$

The labelling process is specified in Figure 5.5. Its definition is based on an auxiliary function $\mathcal{L}_i$ whose index i allows to decide if a label must be inserted in the next subterm. As said earlier, labels are only inserted at the beginning of abstraction's bodies and at the end of applications.

The instrumentation function $\mathcal{I}$ cannot use global variables to monitor resources since λ has no imperative mechanisms. To cope with that issue, we perform a call-by-value monadic translation [9] on the labelled source program. Given a monoid $(\mathbb{C}, \oplus, 0)$ representing the resources quantities, the instrumentation transforms the term so that each computation returns a pair of a result and an element of $\mathbb{C}$. Each label emission is then interpreted as an increment of this element. The addition of the monoid is handy to join the costs coming from different subterms. The precise definition of this function is given in Figure 5.6.

Theorem 12 (Correctness of the cost annotating compiler for λ)
If $\pi_2(\mathcal{I}(\mathcal{L}(t)))$ converges to m, then the cost of $\mathcal{C}(t)$ is m.

$$
\begin{array}{rcl}
\mathcal{I}(x) & = & x \\
\mathcal{I}(\lambda x^{+}.t) & = & \lambda x^{+}.\mathcal{I}(t) \\
\mathcal{I}(t^{+}) & = & (\mathcal{I}(t))^{+} \\
\mathcal{I}(t\,(u_0,\ldots,u_n)) & = & \mathbf{let}\ (x_0, m_0) = \mathcal{I}(u_0)\ \mathbf{in} \\
& & \mathbf{let}\ (x_1, m_1) = \mathcal{I}(u_1)\ \mathbf{in} \\
& & \ldots \\
& & \mathbf{let}\ (x_n, m_n) = \mathcal{I}(u_n)\ \mathbf{in} \\
& & \mathbf{let}\ (x, m) = x_0(x_1,\ldots,x_n)\ \mathbf{in} \\
& & (m_0 \oplus \ldots \oplus m_n \oplus m, x) \\
\mathcal{I}(\pi_i(t)) & = & \mathbf{let}\ (x, m) = \mathcal{I}(t)\ \mathbf{in} \\
& & (\pi_i(x), m) \\
\mathcal{I}(\ell > t) & = & \mathbf{let}\ (x, m) = t\ \mathbf{in} \\
& & (x, m_\ell \oplus m) \\
\mathcal{I}(t > \ell) & = & \mathbf{let}\ (x, m) = t\ \mathbf{in} \\
& & (x, m \oplus m_\ell) \\
\mathcal{I}(\mathbf{let}\ x = t\ \mathbf{in}\ u) & = & \mathbf{let}\ (x, m_0) = \mathcal{I}(t)\ \mathbf{in} \\
& & \mathbf{let}\ (y, m_1) = \mathcal{I}(u)\ \mathbf{in} \\
& & (y, m_0 \oplus m_1)
\end{array}
$$

Figure 5.6: Instrumentation of λ through a call-by-value monadic translation in the cost monad.

The proof follows exactly the same steps as the similar proof of correctness for Imp cost annotating compiler.

Verification of synthetic cost annotations Higher-order Hoare Logic [5] is a deductive proof system for purely functional programs developed during my Ph.D. thesis. This deductive system reduces the verification of a functional program annotated with logical assertions to the a set of proof obligations. We adapt this deductive reasoning framework to reason about (concrete) complexity.

Logical assertions are written in a typed higher-order logic whose syntax is given in Table 5.1. From now on, we assume that our source language is also typed. The metavariable τ ranges over simple types, whose syntax is $\tau ::= \iota \mid \tau \times \tau \mid \tau \to \tau$ where ι are the basic types including a data type cm for the values of the cost monoid. The metavariable θ ranges over logical types. prop is the type of propositions. Notice that the inhabitants of arrow types on the logical side are purely logical (and terminating) functions, while on the programming language's side they are computational (and non-terminating) functions. Types are lifted to the logical level through a logical reflection $\lceil \bullet \rceil$ defined in Table 5.1.

We write "$\mathsf{let}\ x : \tau / F = t\ \mathsf{in}\ u$" to annotate a let definition by a postcondition F of type $\lceil \tau \rceil \to \mathsf{prop}$. We write "$\lambda(x_1 : \tau_1)/F_1 : (x_2 : \tau_2)/F_2.\ M$" to ascribe to a λ-abstraction a precondition F_1 of type $\lceil \tau_1 \rceil \to \mathsf{prop}$ and a postcondition F_2 of type $\lceil \tau_1 \rceil \times \lceil \tau_2 \rceil \to \mathsf{prop}$. Computational values are lifted to the logical level using the reflection function defined in Table 5.1. The key idea [5] of this definition is to reflect a computational function as a pair of predicates consisting in its precondition and its postcondition. Given a computational function f, a formula can refer to the precondition (resp. the postcondition) of f using the predicate $\mathsf{pre}\, f$ (resp. $\mathsf{post}\, f$). Thus, pre (resp. post) is a synonymous for π_1 (resp. π_2).

To improve the usability of our tool, we define a surface language by extending λ with several practical facilities. First, terms are explicitly typed. Therefore, the labelling $\mathcal{L}$ must be extended to convey type annotations in an explicitly typed version of λ^{ℓ}. The instrumentation $\mathcal{I}$ defined in Table 5.6 is extended to types by replacing each type annotation τ by its monadic interpretation $\llbracket \tau \rrbracket$ defined by $\llbracket \tau \rrbracket = \mathsf{cm} \times \overline{\tau}$, $\overline{\iota} = \iota$, $\overline{\tau_1 \times \tau_2} = (\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket)$ and $\overline{\tau_1 \to \tau_2} = \overline{\tau_1} \to \llbracket \tau_2 \rrbracket$.

Second, since the instrumented version of a source program would be cumbersome to reason about because of the explicit threading of the cost value, we keep the program in its initial form while allowing logical assertions to implicitly refer to the instrumented version of the program. Thus, in the surface language, in the term "$\mathsf{let}\ x : \tau / F = M\ \mathsf{in}\ M$", F has type $\lceil \llbracket \tau \rrbracket \rceil \to \mathsf{prop}$, that is to say a predicate over pairs of which the first component is the execution cost.

Third, we allow labels to be written in source terms as a practical way of giving names to the labels introduced by the labelling $\mathcal{L}$. By that means, the constant cost assigned to a label ℓ can be symbolically used in specifications by writing $\mathsf{costof}(\ell)$.

Syntax

$$\begin{array}{lcll}
F & ::= & \mathsf{True} \mid \mathsf{False} \mid x \mid F \wedge F \mid F = F \mid (F, F) & \text{(Formulae)} \\
 & & \mid \pi_1 \mid \pi_2 \mid \lambda(x : \theta).F \mid F\,F \mid F \Rightarrow F \mid \forall(x : \theta).F & \\
\theta & ::= & \mathsf{prop} \mid \iota \mid \theta \times \theta \mid \theta \rightarrow \theta & \text{(Types)} \\
v & ::= & \mathsf{x} \mid \lambda(\mathsf{x} : \tau)^{+}/F : (y : \tau)/F.t \mid (v^{+}) & \text{(Values)} \\
t & ::= & v \mid @(t, t^{+}) \mid \mathbf{let}\ \mathsf{id} : \tau/F = t\ \mathbf{in}\ u \mid (t^{+}) \mid \pi_i(t) & \text{(Terms)}
\end{array}$$

Logical reflection of types

$$\begin{array}{rcl}
\lceil \iota \rceil & = & \iota \\
\lceil \tau_1 \times \ldots \times \tau_n \rceil & = & \lceil \tau_1 \rceil \times \ldots \lceil \tau_n \rceil \\
\lceil \tau_1 \longrightarrow \tau_2 \rceil & = & (\lceil \tau_1 \rceil \longrightarrow \mathsf{prop}) \times (\lceil \tau_1 \rceil \times \lceil \tau_2 \rceil \longrightarrow \mathsf{prop})
\end{array}$$

Logical reflection of values

$$\begin{array}{rcl}
\lceil id \rceil & = & id \\
\lceil (V_1, \ldots, V_n) \rceil & = & (\lceil V_1 \rceil, \ldots, \lceil V_n \rceil) \\
\lceil \lambda(x_1 : \tau_1)/F_1 : (x_2 : \tau_2)/F_2.\ M \rceil & = & (F_1, F_2)
\end{array}$$

Table 5.1: The surface language.

```
let rec exists (p : nat -> bool, l: list) { forall x, pre p x }
: bool {
   ((result = BTrue) <=> (exists x c : nat, mem x l ∧ post p x (c, BTrue))) ∧
   (forall k : nat, bounded p k ∧ (result = BFalse) ->
    exists k0 k1, cost <= k0 + (k + k1) * length (l))
} =
  ℓm> match l with
  | Nil -> ℓnil> BFalse
  | Cons (x, xs) -> ℓc> match p (x) > ℓp with
     | BTrue -> BTrue
     | BFalse -> ℓf> (exists (p, xs) > ℓr)
  end
end
```

Figure 5.7: Two function programs with certified cost annotations.

Finally, as a convenience, we write "$x : \tau/F$" for "$x : \tau/\lambda(\text{cost} : \mathsf{cm}, x : \lceil[\![\tau]\!]\rceil).F$". This improves the conciseness of specifications by automatically allowing reference to the cost variable in logical assertions without having to introduce it explicitly.

An example program is given in Figure 5.7. This recursive boolean higher-order function tests if there exists an element that satisfies a predicate `p` in a list of natural numbers. The precondition of `exists` asks the predicate to be defined for any natural number. The postcondition of `exists` is the conjunction of two properties. The first one is the functional correctness of this higher-order function while the second position asserts that if the cost of `p` is bounded by a constant `k` then the cost of `exists` is bounded by a linear function of the length of the input list. The verification condition generator produces 53 proof obligations out of the function `exists`; 46 of these proof obligations are discharged by automatic provers and 7 of them are manually proved in Coq.

5 Related work

WCET for imperative programs Worst Case Execution Time (WCET) tools [7, 11] analyze executable binaries with very precise cost models to assign a tight upper bound on the execution time of programs. These tools are usually based on manual annotations of the binary code : for instance, the user has to provide the number of iterations for each loop. These annotations are of course not certified, and the declared numbers are constants, not expressions depending on the variables of the source program.

We unfortunately cannot combine these tools with CerCo. Indeed, our cost annotating compiler needs information about the execution cost of relatively small sequences of instructions and these costs must be additive. WCET tools do not provide such guarantees because their analysis are not compositional, and for good reasons. One of them is due to the fact that they take memory caches into account to obtain precise upper bound on the execution time: caches dramatically optimize execution time when they contain relevant information, thus, a sequence of instructions can run faster depending on what precedes it.

Back-end optimizations like loop unrolling and code inlining can duplicate the labels introduced by CerCo and subsequent optimizations like constant propagation or arithmetic simplification can significantly reduce the execution cost associated to one occurrence of a duplicated label with respect to the others. In that case, CerCo takes the maximal cost of these different occurrences. While this choice is sound, this is a source of imprecision. Tranquilli [8] extends the labelling method of CerCo with *indexed labels*. These labels are parameterized with respect to the source program loop indexes so that the costs associated to each label can depend on the the value of these indices.

Complexity for functional programs Most of the work about the complexity of higher-order programs do not take the compilation process into account. They usually consider a cost model defined in terms of the source language to conduct asymptotic complexity analysis [2, 6] instead of worst-case execution time analysis. When the objective is to determine if a functional program lives in given complexity, the rich domain of implicit complexity offers interesting dedicated proof systems [4]. When the objective is to get finer characterization of a program's asymptotic complexity, one can resort to assisted reasoning [2, 12].

Hume [1] is sharing the same motivations as CerCo which is to reason about the concrete complexity of functional programs. This is specialized purely functional program which is compiled to an abstract machine. By conducting a WCET on the C implementation of this abstract machine, the authors build up a precise cost model for the target language of Hume. However, Hume mimicks the WCET tools for imperative programs as it only provides constant execution bounds, i.e., execution bounds that do not depend on the source program values. In addition, there is no proof of correctness of the approach.

6 Conclusion and Future Work

The project CerCo demonstrates that compiler mechanized verification can not only show extensional properties about compiled programs but also intentional properties about the way source programs are compiled. In particular, the proof techniques based on the labelling method naturally extends the existing simulation proofs. These techniques seem to apply to any compilation chain as we have systematically used them to reason about the complexity of both imperative and functional programs.

Recently, the need for concrete bound on execution time appeared in a new context: blockchains. In collaboration with Benjamin Canou, we are about to start a Ph.D. thesis that will reuse most of the ideas of the CerCo project.

References

[1] Armelle Bonenfant et al. "Worst-Case Execution Times for a Purely Functional Language". In: *Implementation and Application of Functional Languages, 18th International Symp osium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers*. Ed. by Zoltán Horváth, Viktória Zsók, and Andrew Butterfield. Vol. 4449. Lecture Notes in Computer Science. Springer, 2006, pp. 235–252.

[2] Armaël Guéneau, Arthur Charguéraud, and François Pottier. "A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Amal Ahmed. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 533–560.

[3] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Commun. ACM* 52.7 (2009), pp. 107–115.

[4] Jean-Yves Marion. “Developments in implicit computational complexity”. In: *Inf. Comput.* 241 (2015), pp. 1–2.

[5] Yann Régis-Gianas and François Pottier. “A Hoare Logic for Call-by-Value Functional Programs”. In: *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings.* Ed. by Philippe Audebaud and Christine Paulin-Mohring. Vol. 5133. Lecture Notes in Computer Science. Springer, 2008, pp. 305–335.

[6] David Sands. “Complexity Analysis for a Lazy Higher-Order Language”. In: *ESOP’90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings.* Ed. by Neil D. Jones. Vol. 432. Lecture Notes in Computer Science. Springer, 1990, pp. 361–376.

[7] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. “Fast and Precise WCET Prediction by Separated Cache and Path Analyses”. In: *Real-Time Systems* 18.2/3 (2000), pp. 157–179.

[8] Paolo Tranquilli. “Indexed Labels for Loop Iteration Dependent Costs”. In: *Proceedings 11th International Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2013, Rome, Italy, March 23-24, 2013.* Ed. by Luca Bortolussi and Herbert Wiklicky. Vol. 117. EPTCS. 2013, pp. 19–33.

[9] Philip Wadler. “Comprehending Monads”. In: *Mathematical Structures in Computer Science* 2.4 (1992), pp. 461–493.

[10] Wikipedia. *INTEL MCS-51.* `https://en.wikipedia.org/wiki/Intel_MCS-51`.

[11] Reinhard Wilhelm et al. “The worst-case execution-time problem - overview of methods and survey of tools”. In: *ACM Trans. Embedded Comput. Syst.* 7.3 (2008), 36:1–36:53.

[12] Bohua Zhan and Maximilian P. L. Haslbeck. “Verifying Asymptotic Time Complexity of Imperative Programs in Isabelle”. In: *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings.* Ed. by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Vol. 10900. Lecture Notes in Computer Science. Springer, 2018, pp. 532–548.

CHAPTER 6

Analysis of Posix shell scripts

> In theory, there is no difference between theory and practice. But, in practice, there is.
>
> Jan L. A. van de Snepscheut

Contents

Related publications:

- *Morbig: a static parser for* Posix *shell.*

  SLE (2018) with Nicolas Jeannerod, and Ralf Treinen.

1 Introduction

Context Since the advent of UNIX, Posix shell scripts are ubiquitous in a large majority of computer systems. For instance, in the Debian distribution, shell scripts install, configure and update software packages. As they modify the operating system, they are executed as root, i.e., with the most powerful privileges. These scripts can therefore affect the consistency of the system if they behave badly.

Ralf Treinen is coordinating the efforts of the CoLiS project with the ambitious goal of verifying the maintainer scripts of the Debian distribution. The project aims at checking that each installation script is safe, that its logic is justified by valid assumptions about the file system, and that it follows the requirements expressed by the Debian policy [7]. Similar properties are to be verified about scripts that remove a package from a system or scripts that update a given package. Interactions between scripts are also under the focus of the CoLiS project: for instance, we want to know if the composition of the installation script and the deinstallation script is equivalent to the identity.

Unfortunately, it is a euphemism to say that the Posix shell language has not been designed with verification in mind. This language suffers from many irregularities and weirdnesses that make reasoning about scripts a very hazardous activity. Despite the relatively recent efforts of the OpenGroup to specify Posix shell syntax and semantics, the prior developments of multiple major shell implementations make the process of building a common specification extremely difficult. As a consequence, the specification of Posix shell is still largely informal and ambiguous. With no precise semantics of Posix shell, the design of a verification framework is a delicate matter because we must always be able to roll back a misinterpretation of the standard. In addition to that, the “organic” growth of the language induced a lot of *adhoc* and unconventional design choices that hardly fit in the standard implementation techniques described in the literature.

Writing a Posix shell static analyzer is, therefore, a programming challenge as the readability and the modularity of its source code must be sufficient to sustain the reviews of experts, and the

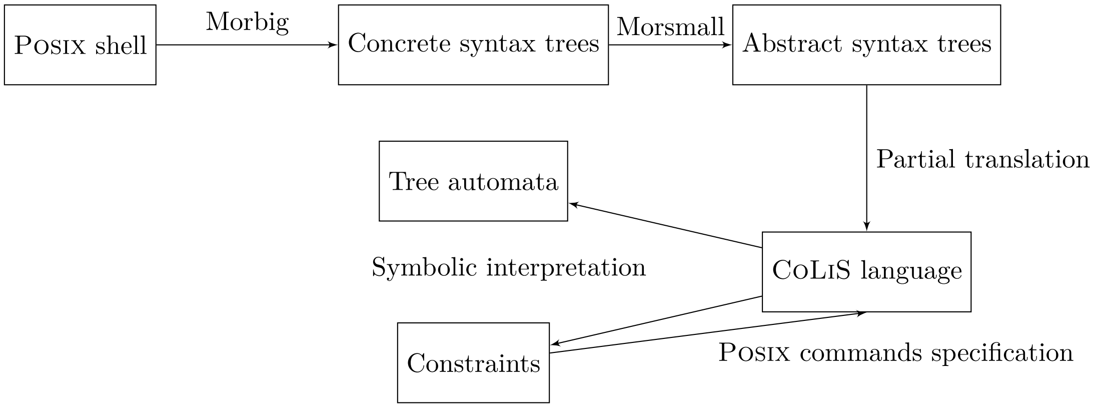


Figure 6.1: The architecture of the CoLiS static analyzer.

consequences of these reviews in terms of source code evolutions. The CoLiS project is still ongoing, but it has committed itself to a proper decomposition of the shell script verification problem since its earliest stages. The resulting architecture is picturized in Figure 6.1. As usual, parsing comes first and it consists in a static parser called Morbig which produces canonical concrete syntax trees, followed by a translation called Morsmall of these concrete syntax trees to specific abstract syntax trees. We translate these abstract syntax trees to a well-specified subset of Posix shell [10] with a partial translation. Finally, a generic symbolic interpreter turns the script into a logically exploitable form. For the moment, scripts are translated to constraints over feature trees with update [11]. We use these constraints as a specification language to represent the effects of Posix commands on the file system. The symbolic interpreter uses them when such command is encountered in a script.

Contributions This may be surprising but a significant part of the CoLiS project first year has been dedicated to the development of Morbig, a static parser for Posix shell. This is our main contribution in CoLiS. We will explain in Section 2 why parsing shell scripts is difficult and in Section 3, why a purely functional approach to this problem helps us to design and to implement Morbig.

2 The difficulties of Posix shell parsing

The syntax of Posix shell is unconventional. In this section, we give a glimpse of shell parsing pitfalls. Many others are described in our paper [17].

Non standard lexical conventions Consider the following script:

```
1   BAR='foo'"ba"r
2   X=0 echo x$BAR" "$(echo $(date))
```

In most programming languages, the first line would be composed of five tokens. By contrast, a lexical analyzer of Posix shell must recognize only one token on that input. Indeed, while standard lexical conventions are typically defined by regular expressions equipped with a longest match strategy, the tokens of Posix shell are only defined by the way they are delimited inside the input. In other words, a Posix shell lexer only recognizes token delimiters, and they should be sufficient to split the inputs into tokens…

However, Line 2 shows another aspect of this problem: the notion of delimiters is contextual. The space character is not always a delimiter: in that example, the first two occurrences are delimiters, whereas the last two are not. The double quotes and the nesting of subshells with the syntax `$(...)` indeed introduces a contextual information which changes how the lexer must recognize token delimiters.

Similarly, the newline character, which is vastly considered as a delimiter by most language designers, has four different interpretations in Posix shell.

```
1  $ for i in 0 1
2  > # Some interesting numbers
3  > do echo $i \
4  > + $i
5  > done
```

On Line 1, the newline character has a syntactic meaning because it acts as a marker for the end of the sequence over which the **for**-loop is iterating. On Line 2, the newline character at the end of the comment must not be ignored but is merged with the newline character of the previous line. On Line 3, the newline character is preceded by a backslash. This sequence of characters is interpreted as a line-continuation, which must be handled at the lexing level. That is, in this case the newline is actually interpreted as layout. On Lines 4 and 5, each of the final newlines terminates a command. Most of the newline characters must therefore be interpreted as tokens because they have a meaning in the grammar, but not all of them!

Parsing-dependent lexical analysis Consider now the following two lines of shell:

```
1  for i in a b; do echo $i; done
2  ls for i in a b
```

On Line 1, the words `for`, `in`, `do`, `done` are recognized as reserved words. On Line 2, they are not recognized as such since they appear in position of command arguments for the command `ls`.

As this example illustrates, lexical analysis depends on parsing. Hence, we cannot apply the modular architecture taught in any compiler course, i.e., the decomposition of the compiler first passes as a lexer producing a stream of tokens for a parser.

Evaluation-dependent lexical analysis Strictly speaking, static parsing of Posix shell is undecidable, as suggested by the following script:

```
if ./foo; then
  alias x="for"
else
  alias x=""
fi
x i in a b; do echo $i; done
```

The `alias` command defines a macro which must be expanded before lexical analysis. That example is therefore syntactically correct if and only if `./foo` succeeds, and we cannot even decide if it terminates.

Non conventional grammar specification To finish this quick tour of Posix shell syntactic issues, let us consider the excerpt of the grammar of the standard given in Figure 6.2. At first glance, this grammar looks as if it was written using a standard BNF format, typically following the input format of Yacc [12]. It is not.

The problem comes from the grammar annotations of the form "`Apply rule 6`". They refer to nine rules informally explained in the text body of the specification. They are actually the place where the parsing-dependent lexical conventions are explained. By lack of space, we only focus on the Rule 4 to give the idea. This is an excerpt from the standard describing this rule:

> *[Case statement termination]*
> *When the* **TOKEN** *is exactly the reserved word* **esac***, the token identifier for* **esac** *shall result. Otherwise, the token* **WORD** *shall be returned.*

The grammar refers to that rule in the following case:

```
pattern:
 WORD /* Apply rule 4 */
| pattern '|' WORD /* Do not apply rule 4 */;
```

Roughly speaking, this annotation says that when the parser is recognizing a `pattern` and when the next token is the specific `WORD` **esac**, then the next token is actually not a `WORD` but the token `Esac`. In that situation, one can imagine that an LR parser must pop up its stack to a state where it is recognizing the non terminal `case_clause` defined as follows:

```
program:
  linebreak complete_commands linebreak | linebreak;
complete_commands:
  complete_commands newline_list complete_command
| complete_command;
complete_command:
  list separator_op | list;
list:
  list separator_op and_or | and_or;
and_or:
  pipeline
| and_or AND_IF linebreak pipeline
| and_or OR_IF  linebreak pipeline;
pipeline:
  pipe_sequence | Bang pipe_sequence;
pipe_sequence:
  command | pipe_sequence '|' linebreak command;
command:
  simple_command | compound_command
| compound_command redirect_list | function_definition;
compound_command:
  brace_group | subshell | for_clause | case_clause
| if_clause | while_clause | until_clause;
subshell:
 '(' compound_list ')';
compound_list:
  linebreak term | linebreak term separator;
term:
  term separator and_or | and_or;
while_clause:
  While compound_list do_group;
```

```
do_group:
  Do compound_list Done /* Apply rule 6 */;
simple_command:
  cmd_prefix cmd_word cmd_suffix
| cmd_prefix cmd_word
| cmd_prefix
| cmd_name cmd_suffix
| cmd_name;
cmd_name:
  WORD /* Apply rule 7a */;
cmd_word:
  WORD /* Apply rule 7b */;
newline_list:
  NEWLINE | newline_list NEWLINE;
linebreak:
  newline_list | /* empty */;
separator_op:
  '&' | ';';
separator:
  separator_op linebreak | newline_list;
sequential_sep:
  ';' linebreak | newline_list;

// The rules for the following nonterminals are elided:
// for_clause, name, in, wordlist, case_clause,
// case_list_ns, case_list, case_item_ns, case_item,
// pattern if_clause, else_part, until_clause,
// function_definition, function_body, fname,
// brace_group, cmd_prefix, cmd_suffix, redirect_list,
// io_redirect, io_file, filename, io_here and here_end.
```

Figure 6.2: A fragment of the official grammar for the shell language.

```
case_clause:
Case WORD linebreak in linebreak case_list Esac
| Case WORD linebreak in linebreak case_list_ns Esac
| Case WORD linebreak in linebreak Esac
```

to conclude the recognition of the current `case_list`.

3 Morbig, a static parser for Posix shell

Parser requirements As the previous section explained, the syntax of Posix shell does not fit the standard practice of parser construction. There are a lot more issues that we cannot describe by lack of space, but the Figure 6.3 sums up the consequences of this issues as a list of requirements on the parser implementation.

Besides these technical requirements, there is an extra methodological one: the mapping between the Posix specification and the source code must be as direct as possible. Indeed, we should ideally be able to describe our understanding of each paragraph of the Posix specification as a well-delimited piece of code. More generally, as our parser is part of a static analyzer, it must be trustworthy: we must improve its quality by all possible means. One good practice consists in using code generators, like Lex and Yacc, to write high-level code that can be more easily related to the specification than low-level manually written parsers.

The tight interaction between the lexer and the parser prevents us from writing our syntactic analyzer following the traditional design found in most textbooks [3], that is a pipeline of a lexer followed by a parser. Hence, we cannot use either the standard interfaces of code generated by Lex and Yacc, because these interfaces have been designed to fit this traditional design. There exists alternative parsing technologies, e.g. scannerless generalized LR parsers or topdown general parsing combinators, that could have offered elegant answers to many of the requirements enumerated previously, but as we will explain in Section 4, we believe that none of them fulfill the entire list of these requirements.

In this situation, one could give up using code generators and fall back to the implementation of a hand-written character-level parser. This is done in Dash for instance: the parser of Dash 0.5.7 is made of 1569 hand-crafted lines of C code. This parser is hard to understand because it is implemented by low-level mechanisms that are difficult to relate to the high-level specification of

(i) Lexical analysis must be aware of the parsing context and of some contextual information like the nesting of double quotes and subshell invocations.

(ii) Lexical analysis must be defined in terms of token delimitations, not in terms of token (regular) languages recognition.

(iii) The syntactic analysis must be able to return the longest syntactically valid prefix of the input.

(iv) The parser must be reentrant ;

(v) The parser must forbid certain specific applications of the grammar production rules.

(vi) The parser must be able to switch between the token recognition process and the here-document scanner.

Figure 6.3: Requirements on the parser implementation.

the POSIX standard: for example, lexing functions are implemented by means of `goto`s and complex character-level manipulations; the parsing state is encoded using activation and deactivation of bit fields in one global variable; some speculative parsing is done by allowing the parser to read the input tokens several times, etc.

Other implementations, like the parser of BASH, are based on a YACC grammar extended with some code to work around the specificities of shell parsing. We follow the same approach except on two important points. First, we are stricter than BASH with respect to the POSIX standard: while BASH is using an entirely different grammar from the standard, we literally cut-and-paste the grammar rules of the standard into our implementation to prevent any change in the recognized language. Second, in BASH, the amount of hand-written code that is accompanying the YACC grammar is far from being negligible. Indeed, we counted approximately 5000 extra lines of C to handle the shell syntactic peculiarities. In comparison, our implementation only needed approximately 1000[1] lines of OCAML to deal with them.

A modular architecture thanks to purely functional programming Our main design choice is not to give up on modularity. As shown in Figure 6.4, the architecture of our syntactic analyzer is similar to the common architecture found in textbooks as we clearly separate the lexing phase and the parsing phase in two distinct modules with clear interfaces. Let us now describe the original aspects of this architecture.

As suggested by the illustration, we decompose lexing into two distinct subphases. The first phase called "prelexing" is implementing the "token recognition" process of the POSIX standard. As said earlier, this parsing-independent step classifies the input characters into three categories of "pretokens": operators, words and potentially significant layout characters (newline characters and end-of-input markers). This module is implemented using OCAMLLEX, a lexer generator distributed with the OCAML language.

The second phase of lexing is parsing-dependent. As a consequence, a bidirectional communication between the lexer and the parser is needed. On one side, the parser is waiting for a stream of tokens to reconstruct a parse tree. On the other side, the lexer needs some parsing context to promote words to keywords or assignment words, to switch to the lexing mode for here-documents, and to discriminate between the four interpretations of the newline character. We manage to implement all these *ad hoc* behaviors using speculative parsing, which is easily implemented thanks to the incremental and purely functional interface produced by the parser generator MENHIR [15].

In that new setting, the caller of a parser must manually provide the input information needed by this parser for its next step of execution and the parser gives back the control to its caller after the execution of this single step. Hence, the caller can implement a specific communication protocol between the lexer and the parser. In particular, the state of the parser can be transmitted to the lexer. This protocol between the incremental parser generated by MENHIR and the parsing engine is specified by a single type definition:

[1] The total number of lines of code is 2141, including type definitions, utilities and infrastructure.

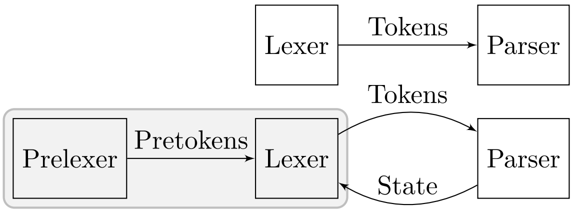


Figure 6.4: Architectures of syntactic analyzers: at the top of the figure, the standard pipeline commonly found in compilers and interpreters; at the bottom of the figure, the architecture of our parser in which there is a bidirectional communication between the lexer and the parser.

```
type 'a checkpoint = private
  | InputNeeded of 'a env
  | Shifting of 'a env * 'a env * bool
  | AboutToReduce of 'a env * production
  | HandlingError of 'a env
  | Accepted of 'a
  | Rejected
```

A value of type `'a checkpoint` represents the entire *immutable* state of the parser generated by Menhir. The type parameter `'a` is the type of semantic values produced by a successful parsing. The type `'a env` is the internal state of the parser which roughly speaking contains the stack and the current state of the generated LR pushdown automaton. As specified by this sum type, there are six situations where the incremental parser generated by Menhir interrupts itself to give the control back to the parsing engine: (i)`InputNeeded` means that the parser is waiting for the next token. By giving back the control to the parsing engine and by exposing a parsing state of type `'a env`, the lexer has the opportunity to inspect this parsing state and decide which token to transmit. This is the property we exploit to implement the parsing-dependent lexical analysis. (ii) `Shifting` is returned by the generated parser just before a *shift* action. We do not exploit this particular checkpoint. (iii) `AboutToReduce` is returned just before a *reduce* action. We exploit this checkpoint to implement the treatment of reserved words. (iv) `HandlingError` is returned when a syntax error has just been detected. We do not exploit this checkpoint. (v) `Accepted` is returned when a complete command has been recognized. In that case, if we are not at the end of the input file, we reiterate the parsing process on the remaining input. (vi) `Rejected` is returned when a syntax error has not been recovered by any handler. This parsing process stops on an error message.

Now that the lexer has access to the state of the parser, how can it exploit this state? Must it go into the internals of LR parsing to decipher the meaning of the stack of the pushdown automaton?

Actually, a far simpler answer can be implemented most of the time: the lexer can simply perform some speculative parsing to observationally deduce information about the parsing state. In other words, to determine if a token is compatible with the current parsing state, the lexer just executes the parser with the considered token to check whether it produces a syntax error, or not. If a syntax error is raised, the lexer backtracks to the parsing state that was just before the speculative parsing execution.

If the parsing engine of Menhir were imperative, then the backtracking process required to implement speculative parsing would necessitate some machinery to undo parsing side-effects. Since the parsing engine of Menhir is purely functional we do not need such a machinery: the state of the parser is an explicit immutable value passed to the parsing engine which returns in exchange a fresh new parsing state without modifying the input state. The API to interact with the generated parser is restricted to only two functions:

```
val offer: 'a checkpoint -> token * position * position -> 'a checkpoint
val resume: 'a checkpoint -> 'a checkpoint
```

The function `offer` is used when the `checkpoint` is exactly of the form `InputNeeded`. In that specific case, the argument is a triple of type `token * position * position` passed to the generated parser.

The function `resume` is used for the other cases to give the control back to the generated parser without transmitting any new input token.

Dash Morbig	All	Accepted	Rejected
All	7,436,215 (100%)	5,981,054 (80%)	1,455,161 (20%)
Accepted	5,609,366 (75%)	5,607,331 (75%)	2,035 (<1%)
Rejected	1,826,849 (25%)	373,723 (5%)	1,453,126 (20%)

Table 6.1: Comparison of Morbig and Dash on the whole corpus from Software Heritage. The percentages are in function of the total number of scripts.

From the programming point of view, backtracking is as cheap as declaring a variable to hold the state to recover it if a speculative parsing goes wrong. From the computational point of view, thanks to sharing, the overhead in terms of space is negligible and the overhead in terms of time is reasonable since we never transmit more than one input token to the parser when we perform such speculative parsing.

Another essential advantage of immutable parsing states is the fact that the parsers generated by Menhir are *reentrant* by construction. As a consequence, multiple instances of our parser can be running at the same time. This property is needed because the prelexer can trigger new instances of the parser to deal with subshell invocations.

Evaluation We have analyzed the 31.582 maintainer scripts present in the Debian *unstable* distribution for the `amd64` architecture, in the areas `main`, `contrib`, and `non-free`, as of 29 Nov 2016. 238 of these are `bash` scripts, 13 are `perl` scripts, one is an ELF executable[2], and hence out of scope for us. Our parser succeeds on 31.484, that is 99.88% of the remaining 31.330 Posix shell scripts.

To disambiguate several paragraphs of the standard, we have checked that the behavior of Morbig coincides with the behavior of shell implementations which are believed to be Posix-compliant, typically Dash and Bash (in Posix mode). We ran both Morbig and Dash on all the files detected as shell scripts by `libmagic`[3] in the Software Heritage [1] archive. This archive contains all the scripts in GitHub, and more, for a total of 7,436,215 files. Table 6.1 shows general numbers about what both parsers accept or reject in this archive. On most scripts (95%), Morbig and Dash do agree. It is interesting to consider the cases where they disagree, because this is where one can find bugs in one parser or the other.

Out of all the 373,723 scripts accepted by Dash and rejected by Morbig the majority (350,259, *i.e.*, 94% and 4.7% of the total) contains Bash-specific constructs in words. Dash, in parse-only mode, separates words but does not look into them. Morbig, on the other hand, does parse words and rejects such scripts. This is not a bug in Dash as the Posix standard does not specify whether such invalid words must be rejected during parsing or during execution.

This means that, in total, the number of scripts on which Morbig and Dash truly disagree is less than 0.4% of the whole archive. These scripts feature unspecified behaviors interpreted differently and a few bugs in both tools on very specific corner cases of the Posix standard.

4 Related work

General parsing frameworks Menhir[15] is based on a conservative extension of LR(1)[13], inspired by Pager's algorithm[14]: it produces pushdown automata almost as compact as LALR(1) automata without the risk of introducing LALR(1) conflicts. As a consequence, the resulting parsers are both efficient (word recognition has a linear complexity) and reasonable in terms of space usage.

However, the set of LR(1) languages is a strict subset of the set of context-free languages. For context-free languages which are not LR(1), there exist well-known algorithms like Earley's [8, 4], GLR[20], GLL[18] or general parser combinators [9]. These algorithms can base their decision on an arbitrary number of look-ahead tokens, can cope with ambiguous grammars by generating parse forests instead of parse trees, and generally have a cubic complexity. There also exist parsing algorithms and

[2] We let the reader find out which package cannot have a `preinst` script written in shell.

[3] `libmagic` is a standard library to detect file formats with heuristics.

specifications that go beyond context-free grammars, e.g. reflective grammars [19] or data-dependent grammars [2].

Since the grammar of Posix shell is ambiguous, one may wonder why we stick to an LR(1) parser instead of choosing a more general parsing framework like the ones cited above. First, as explained in Section 2, the Posix specification embeds a Yacc grammar specification which is annotated by rules that change the semantics of this specification, but only locally by restricting the application of some of the grammar rules. Hence, this leads us to think that the authors of the Posix specification actually have a subset of an LR(1) grammar in mind. Being able to use an LR(1) parser generator to parse the Posix shell language is in our opinion an indication that this belief is true. Second, even though we need to implement some form of speculative parsing to efficiently decide if a word can be promoted to a reserved word, the level of non-determinism required to implement this mechanism is quite light. Indeed, it suffices to exploit the purely functional state of our parser to implement a backtracking point just before looking at one or two new tokens to decide if the context is valid for the promotion, or not. This machinery is immediately available with the interruptible and purely functional LR(1) parsers produced by Menhir. In our opinion, the inherent complexity of generalized parsing frameworks is not justified in that context.

Scannerless parsing Many legacy languages (e.g. PL/1, COBOL, FORTRAN, R, ...) enjoy a syntax which is incompatible with the traditional separation between lexical analysis and syntactic analysis. Indeed, when lexical conventions (typically the recognition of reserved words) interact in a nontrivial way with the parsing context, the distinction between lexing and parsing fades away. For this reason, it can perfectly make sense to implement the lexical conventions in terms of context-free grammar rules and to mix them with the language grammar. With some adjustments of the GLR parsing algorithm to include a longest-match strategy and with the introduction of specification mechanisms to declare layout conventions efficiently, the ASF+SDF project[6] has been able to offer a declarative language to specify modular scannerless grammar[21] specifications for many legacy languages with parsing-dependent lexical conventions.

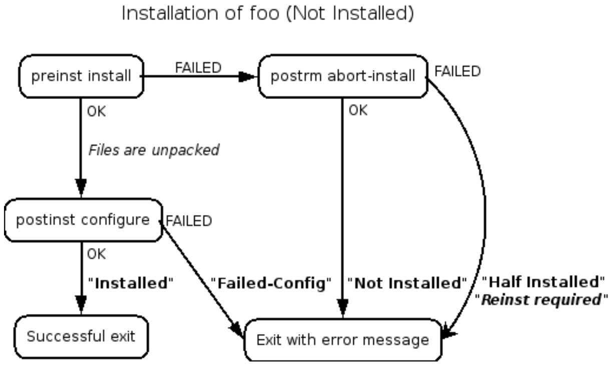
Unfortunately, as said in Section 2, the lexical conventions of Posix shell are not only parsing-dependent but also specified in a “negative way”: Posix defines token recognition by characterizing how tokens are delimited, not how they are recognized. Besides, as also shown in Section 2, the layout conventions of Posix shell, especially the handling of newline characters, are unconventional, hence they hardly match the use cases of existing scannerless tools. Finally, lexical conventions depend not only on the parsing context but also on the nesting context. For all these reasons, we are unable to determine how these unconventional lexical rules could be expressed following the scannerless approach. More generally, it is unclear to us if the expressivity of ASF+SDF specifications is sufficient to handle the Posix shell language without any extra code written in a general purpose programming language.

Schrödinger’s tokens Schrödinger’s tokens[5] is a technique to handle parsing-dependent lexical conventions by means of a superposition of several states on a single lexeme produced by the lexical analysis. This superposition allows to delay to parsing time the actual interpretation of an input string while preserving the separation between the scanner and the parser. This technique only requires minimal modification to parsing engines. Morbig’s promotion of words to reserved words follows a similar path: the prelexer produces pretokens which are similar to Schrödinger’s tokens since they enjoy several potential interpretations at parsing time. The actual decision about which is the right interpretation of these pretokens as valid grammar tokens is deferred to the lexer and obtained by speculative parsing. No modification of Menhir’s parsing engine was required thanks to the incremental interface of the parsers produced by Menhir: the promotion code can be written on top of this interface.

5 Conclusion and Future work

Statically parsing shell scripts is notoriously difficult, due to the fact that the shell language was not designed with static analysis in mind. Thanks to functional programming techniques, we have written a parser that maintains a high level of modularity, despite the fact that the syntactic analysis of shell scripts requires an interaction between lexing and parsing that defies traditional approach.

The next step of the CoLiS project is the actual verification of the approximately 32,000 maintainer scripts of Debian. For each package, we want to make sure that diagrams of the following form [16]:



are actually executable with the maintainer scripts under any reasonable configuration of the file system.

References

[1] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. "Building the universal archive of source code". In: *Communications of the ACM* 61.10 (2018), pp. 29–31.

[2] Ali Afroozeh and Anastasia Izmaylova. "Iguana: a practical data-dependent parsing framework". In: *Proceedings of the 25th International Conference on Compiler Construction.* ACM. 2016, pp. 267–268.

[3] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[4] John Aycock and R Nigel Horspool. "Practical earley parsing". In: *The Computer Journal* 45.6 (2002), pp. 620–630.

[5] John Aycock and R. Nigel Horspool. "Schrödinger's token". In: *Softw., Pract. Exper.* 31 (2001), pp. 803–814.

[6] Mark GJ van den Brand et al. "The asf+sdf meta-environment: A component-based language development environment". In: *International Conference on Compiler Construction.* Springer. 2001, pp. 365–370.

[7] *Debian Policy Manual.*

[8] Jay Earley. "An efficient context-free parsing algorithm". In: *Communications of the ACM* 13.2 (1970), pp. 94–102.

[9] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. "Practical, general parser combinators". In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation.* ACM. 2016, pp. 1–12.

[10] Nicolas Jeannerod, Claude Marché, and Ralf Treinen. "A Formally Verified Interpreter for a Shell-Like Programming Language". In: *Verified Software. Theories, Tools, and Experiments - 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers.* Ed. by Andrei Paskevich and Thomas Wies. Vol. 10712. Lecture Notes in Computer Science. Springer, 2017, pp. 1–18.

[11] Nicolas Jeannerod and Ralf Treinen. "Deciding the First-Order Theory of an Algebra of Feature Trees with Updates". In: *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings.* Ed. by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Vol. 10900. Lecture Notes in Computer Science. Springer, 2018, pp. 439–454.

[12] Steven C. Johnson. "Yacc: Yet Another Compiler Compiler". In: *UNIX Programmer's Manual.* Vol. 2. AT&T Bell Laboratories Technical Report July 31, 1978. 1979, pp. 353–387.

[13] Donald E Knuth. "On the translation of languages from left to right". In: *Information and control* 8.6 (1965), pp. 607–639.

[14] David Pager. "A practical general method for constructing LR (k) parsers". In: *Acta Informatica* 7.3 (1977), pp. 249–268.

[15] François Pottier and Yann Régis-Gianas. "The Menhir parser generator". In: *See: http://gallium. inria. fr/fpottier/menhir* ().

[16] Debian project. *Maintainer script flowcharts*. https://www.debian.org/doc/debian-policy/ap-flowcharts.html.

[17] Yann Régis-Gianas, Nicolas Jeannerod, and Ralf Treinen. "Morbig: a static parser for POSIX shell". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*. Ed. by David Pearce, Tanja Mayerhofer, and Friedrich Steimann. ACM, 2018, pp. 29–41.

[18] Elizabeth Scott and Adrian Johnstone. "GLL parsing". In: *Electronic Notes in Theoretical Computer Science* 253.7 (2010), pp. 177–189.

[19] Paul Stansifer and Mitchell Wand. "Parsing reflective grammars". In: *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*. ACM. 2011, p. 10.

[20] Masaru Tomita and See-Kiong Ng. "The generalized LR parsing algorithm". In: *Generalized LR parsing*. Springer, 1991, pp. 1–16.

[21] Eelco Visser et al. *Scannerless generalized-LR parsing*. Universiteit van Amsterdam. Programming Research Group, 1997.