Incrementality and effect simulation in the simply typed lambda calculus

Lourdes del Carmen Gonzalez Huesca

UNIVERSITÉ PARIS DIDEROT – PARIS 7
SORBONNE PARIS CITÉ

ÉCOLE DOCTORALE SCIENCES MATHÉMATIQUES DE PARIS CENTRE
LABORATOIRE PREUVES, PROGRAMMES ET SYSTÈMES – ÉQUIPE πr^2

THÈSE DE DOCTORAT
EN INFORMATIQUE

Présentée et soutenue par

Lourdes del Carmen
GONZÁLEZ HUESCA

le 27 novembre 2015

INCREMENTALITY AND EFFECT SIMULATION IN THE SIMPLY TYPED LAMBDA CALCULUS

INCRÉMENTALITÉ ET SIMULATION D'EFFETS DANS LE LAMBDA CALCUL SIMPLEMENT TYPÉ

devant le jury composé de

Mme.	Delia KESNER	:	*Présidente*
M.	Hugo HERBELIN	:	*Directeur*
M.	Yann RÉGIS-GIANAS	:	*Directeur*
Mme.	Catherine DUBOIS	:	*Rapporteuse*
M.	Sylvain CONCHON	:	*Rapporteur*
M.	Daniele VARACCA	:	*Examinateur*
Mme.	Sandrine BLAZY	:	*Examinatrice*

Abstract

Certified programming is a framework in which any program is correct by construction. Proof assistants and dependently typed programming languages are the representatives of this paradigm where the proof and implementation of a program are done at the same time. However, it has some limitations: a program in Type Theory is built only with pure and total functions.

Our objective is to write efficient and certified programs. The contributions of this work are the formalization, in the Simply Typed Lambda Calculus, of two mechanisms to achieve efficiency: to validate *impure* computations and to optimize computations by *incrementality*.

An impure computation, that is a program with effects, and its validation in a functional and total language is done through *a posteriori simulation*. The simulation is performed afterwards on a monadic procedure and is guided by a *prophecy*. An efficient *oracle* is responsible for producing prophecies which is actually, the monadic procedure itself translated into an effectful programming language.

The second contribution is an optimization to perform incremental computations. Incrementality as a way to propagate an input change into a corresponding output change is guided by formal change descriptions over terms and dynamic differentiation of functions. Displaceable types represent data-changes while an extension of the simply typed lambda calculus with differentials and partial derivatives offers a language to reason about incrementality.

Résumé

La programmation certifiée offre un cadre dans lequel tout programme est correct par construction. Les assistants de preuve et les langages de programmation avec types dépendents sont les représentants de ce paradigme, où la prévue et l'implementation d'un programme sont faites au même temps. Toutefois, il existe certaines limitations : un programme écrit en théorie des types est construit seulement avec des fonctions pures et totales.

Notre objectif est d'écrire des programmes efficaces et certifiés. Les contributions de cette thèse sont la formalisation, dans le lambda calcul simplement typé, de deux mécanismes pour améliorer l'efficacité : la validation des calculs impurs et l'optimisation des calculs incrémentaux.

Un calcul impur, c'est-à-dire un programme avec effets, et sa validation dans un langage fonctionnel et total est fait á l'aide d'une *simulation a posteriori*. La simulation est effectuée après, par une procédure monadique et elle est guidée par une *prophétie*. Un *oracle* efficace est responsable de la production des prophéties et lui est en fait, la procédure monadique traduite dans un language de programmation généraliste.

La deuxième contribution est une optimisation pour les calculs incrémentaux. L'incrémentalité consiste à propager des changements des entrées en changements des sorties, elle est guidée par les descriptions formelles du changement des termes et une différenciation dynamique des fonctions. La représentation des changements de données est pris en charge par les types déplaçables et une extension du lambda calcul simplement typé avec dérivées et dérivées partielles offre un language pour raisonner sur l'incrementalité.

Remerciements – Agradecimientos

Mon aventure appelée doctorat *a été marquée par des rencontres avec plusieurs personnes. Ces remerciments sont dédiés à toutes ces personnes, que je vais essayer de toutes nommer* [1].

En premier lieu, je veux remercier Delia Kesner pour ses efforts en faveur des liens a-cademiques. Sans son conseil, je ne serais jamais m'engagée dans le doctorat au sein de l'équipe πr^2.

Aussi, dans ce premier merci, je suis très reconnaissant du travail d'encadrement et d'encouragement de Yann Régis-Gianas. Dès le premier jour, il a été un soutien académique et aussi dans les affaires administratives françaises. Merci pour me rappeler toujours le sens pédagogique et pour me montrer un autre regard de mon travail.

Merci à Hugo Herbelin pour être une figure constante dans l'équipe, pour nous faire confiance à Yann et moi dans cette aventure.

Un grand merci aux rapporteurs qui ont accepté la responsabilité de lire et comprendre mon travail. À Catherine Dubois pour les (nombreuses) remarques et commentaires qui ont amelioré ce manuscrit. À Sylvain Conchon pour ses commentaires très précis.

Et un deuxième grand merci aux examinateurs, Sandrine Blazy et Daniele Varacca, pour prendre part au jury.

Cette aventure a eu que de grandes contributions académiques: toute l'équipe πr^2 *et surtout le laboratoire PPS ont été d'excellents fournisseurs de savoir avec les membres permanents et les personnes invitées, aussi avec les nombreux séminaires et groupes de travail.*

Cette première année à Place d'Italie a facilité le développement de mon premier 'noyau dur', lie à l'apprentissage de la langue et culture française et au côté fraternité COQ*: Pierre, Pierre-Marie, Matthias et Guillaume. Je vous remercie énormément de l'encouragement et de la gentillesse de tous ces jours avec votre patience et soutien.*

Pendant les célèbres journées PPS à Trouville 2012, j'ai découvert le vrai laboratoire et quelques mois plus tard nous nous sommes tous rencontrés dans notre nouveau bâtiment. Depuis cela, le groupe de thésards est devenu plus qu'une bande des potes au labo, mais une bande de gens qui n'a cessé d'ajouter plus de membres.

C'est ici que mon deuxième 'noyau dur' a été crée : Ioana, Kuba, Shahin qui m'ont adopté et m'ont surtout encouragé pendant la rédaction de cette thèse. Merci Ioana d'être une des mamacitas les plus fortes et cools du monde. Merci Shahin pour mettre en perspective mes origines et de vivre sans avoir peur. Merci Kuba pour toujours ajouter des images et phrases et parce que 'eres un hombre muy honrado que le gusta lo mejor...'

1. *Si jamais vous n'êtes pas présent dans cette partie, je tiens à vos présenter mes excuses les plus sincères car mon cerveau est devenu inútil ces dernières semaines, mais certainement vous êtes présente dans mon coeur.*

Je tiens à remercier aux thésards qui m'ont toujours fait sourire: Étienne[2]*, Matthieu, Flavien, Giulio, Ludovic, Jovana, Hadrien, Thibaut, Marie. Et tous les thésards dont la présence a contribué aux journées très agréables au labo.*

Merci aussi aux thésards Liafa, en particulier à Bruno et Jehanne qui ont toujours descendu les escaliers pour venir nous visiter et surtout pour partager les soirées.

Samy, Rafa, Luis, Shadi, merci pour les soirées, cafés et discussions très variées.

Les couloirs de PPS bien silencieux m'ont fait aussi sourire, merci à Juliusz pour toujours avoir une histoire à raconter, merci à Paul-André pour les samedis avec le son du violon et plus récemment, merci pour les bonjours de Daniel. Merci aux stagiaires avec lesquels on a partagé un eté au labo et qui sont restés plus dans le coin: Philipp, Kenji et Béa.

Le boulot administratif a toujours été un succés grâce à deux super responsables administratives, merci Lindsay, merci Odile.

Un côté académique international a influencié ces années, merci à Tim Griffin pour me montrer les origins du COQ*, à Beta pour ses nombreuses visites et discussions sur n'importe quel sujet, à Daniel pour le sourire brésilien, à Pablo et Elisa.*

No puedo dejar de dar las gracias al lugar donde viví por más de la mitad de este tiempo acá y en donde conocí otra parte de mi país: la Maison du Mexique en citéU. Ahí tuve el privilegio de conocer a muchas personas, las cuales han sido mi apoyo más grande todo este tiempo. Gracias por las fêtes, les pique-niques, les dimanches culturels, les petits week-ends, las vacaciones y por la isla. Merci Denis por ser la primera en recibirme con una sonrisa aquel 31 de diciembre. Merci Alberto por las pláticas deportivas y las no tan deportivas... Merci Juan Carlos por las discussions sobre la vida y la música. Merci a todos los que compartimos más que una ciudad de estudiantes: Fer, José, Vero, Ale, Dama, Arturo, Nuria, Daniel, Braulio, Alejandro, Xiomara, Ale, Laura...

También las visitas fueron un gran apoyo: Edgar, Mini, Liliana, Emilio, Daniela, Noé, Merari, Arturo, Nancy, Adriana, Selene, Moni, Ame. Gracias por los juegos, los 'chats' y por las visitas turísticas que me ayudaron a conocer más.

El apoyo a la distancia ha sido constante: Favio, Liliana y Rafa, siempre al pendiente, mil gracias. Gracias Felipe por soñar(me) y contestar siempre, sin importar mis crisis y locuras.

Finalmente, las gracias infinitas van hacia mi familia: a mi mamá por siempre estar ahi, y a Criss y a Carlos, los tres que son todo junto con los dos pilares de mi vida. No puedo dejar de agradecer a quien me inició en la computación. Y a pesar de todo, 'por hacerme el paro' en cualquier circunstancia y por el apoyo incondicional, gracias Joel.

Un dia nos pidieron, como ejercicio, que describieramos el doctorado en tres palabras. A mi mente vinieron las palabras: desafío, introspección, constancia, resistencia y aprendizaje. Mi decisión de iniciar un doctorado fue motivado por la idea de aprender, de dar un paso más en el camino profesional, por la necesidad de compartir el conocimiento, pero no comprendí que este camino está hecho para convertirse en maestro de uno mismo. Después de estos cuatro años, definidos por autoaprendizaje y por ponerse a prueba constantemente, he aprendido más que sólo lo académico, la capacidad de asombro por cosas nuevas y por lo desconocido, pero sobre todo he aprendido el sentido de la ***amistad****.*

2. *Gracias por corregir mi francés!*

Contents

Note: British English is used along this document, but some established nouns in literature come from American English.

Chapter 1

Introduction

One of the principal and practical aspects of Computer Science is to give solutions to problems by computations through algorithms or programs. The computer scientist may see in the solution of abstract problems an opportunity for creativity, where once a solution is found the curious scientist takes the challenge further and seeks to improve it.

Ideally, problem solving should obey a cycle of development and program improvement more or less as the following schema: a specification of the problem to be solved, a sketch of the solution, an implementation and finally, some optimisations to the solution or to the implementation in order to improve the performance.

The optimisations may vary depending on the user and the desired behaviour. We are confident that they should not belong to an isolated phase at the end of the process, they must be included at different levels and stages of program design and implementation.

Also, the development process should be directed by techniques and standards dedicated to obtain better solutions and to ensure that the final program indeed satisfy the specification, that it actually solves the problem. Furthermore, the design and implementation phases are guided not only by the specification but they are commonly influenced by the programming paradigm in which the solution will be carried out.

A higher demand of software within a short developing time and the necessity of trustworthy programs with respect to their specification become more and more ubiquitous. In addition to this, the errors or program failures due to specification mismatch or encoding flaws should not be present in the implementation.

We can argue that the above remarks can be alleviated by adding a phase of program verification to the development cycle, where a formal validation of the implementation should be carried out. In practice, the verification is substituted by a merely code test and is occasionally totally skipped from the development cycle.

In order to ease the program verification, to amend the errors or even prevent them, we claim that any program development must be done in an environment which ensures the correctness of programs from the very process of design and creation.

However, there exist a large number of computing paradigms, many different, some of them share certain qualities while other contrast the abstraction and understanding of programming. The user must choose the best among them, which offers convenient tools in order to perform reliable software developments.

To restrict the possible failures the programmer can make, some researchers and users support the functional paradigm as a *safe* development environment [53]. The confidence in functional programming languages has been largely considered, they offer a modular and high-level environment to create well-structured programs. They are closer to mathematics, and hence the program specifications can be verified rigorously and almost in a direct way.

Furthermore, there is an increasing interest in providing a rigorous verification of programs, *i.e.*, to validate that a program meets a specification through a formal and mechanised proof. This draws the attention to the recently called *certified programming* paradigm, a new concept of programming where the programs are *correct by construction* [29]. As functional programming, certified programming is a strict and elegant framework for abstraction, verification and coding, based on type theory.

Through this introductory chapter, the reader will be immersed into the universe of theoretical aspects of programming languages leading to the certified programming paradigm which is the inspiration of the work and research done during my PhD. We give a brief reminder of the λ-Calculus and its relation with proof assistants, our contributions and some initial notations.

1.1 λ-Calculus and Proof Assistants

λ-Calculus is a model of computation, a formalism where functions are the central object of study. The original theory defined by Church in 1932, is a language to manipulate functions in its most pure and abstract form. Curry includes the notion of type for any lambda-expression in 1934, thus initiating the second most studied functional formalism.

While the two models of calculus, Church and Curry, are sufficient for reasoning about functions in a general setting, there is a practical side where computability becomes a *tangible* task thanks to (abstract) machines.

λ-Calculus inspired the first machines and their programming languages. Since then, there is a double encouragement between formal and practical aspects of the calculus, leading to more and more complex developments which raise to expressive theories and moreover, specialise a variety of programming languages [57, 91].

During the second half of the 20th century, the development of (typed) theories as formal frameworks to model many classes of programming languages, inspired more theories and promote implementations to realise the concept of computation.

In particular, λ-Calculus is considered as the basis of functional programming [106]. This paradigm of computing is a theory also characterised as 'pure and elegant', in the sense that is considered as a high-level language where reasoning is clear and programs are easier to verify given the closeness to mathematics.

As said before, the necessity of programs with fewer errors, developed in less time and with a high level of reliability, demands complex and specialised frameworks for software development. The more a program development is based and constructed following an accurate design, the more we will trust its reliance to the specification.

Thus, the formal approach of type theory in functional programming makes this paradigm a perfect environment to develop correct programs by construction. As a computational model, *per se*, the typed theories enable to reason about languages and their properties, bringing closer a formal verification.

The Curry-Howard correspondence enables a direct construction of programs from a mathematical specification, it considers Type Theory as a language of proofs where solutions to problems are given through demonstrations of logical formulae. It is founded on two notions: a formula describes a calculation that is, a specification is given through a logical statement, and given a formula a proof of it can be *decorated* resulting in a program. Then, the correspondence between *formulae* and *types*, between *proofs* and *programs*, ensures that demonstrating a formula or finding an inhabitant of a type will deliver a program that satisfies its specification.

This correspondence inspires many programming languages and also theorem provers or proof assistants. The latter are computer programs to help the verification of proofs. For an historical and accurate presentation of the *propositions as types* principle we refer the reader to the work of Wadler [112].

Foundations of Theorem Provers

Scott presented in 1969 a formal language which served as an approach to denotational semantics for programming languages, this model was named *Logic of Computable Functions* (LCF). After this, Milner was inspired and proposed in 1972 a *Logic for Computable Functions* (also abbreviated as LCF), a theorem prover. This logic was refined and later in 1979 Milner together with Gordon and Wadsworth presented the *Edinburgh LCF* [1], an interactive theorem prover along with its meta-language ML. The latter was conceived as a language to develop proof tactics in the theorem prover. While it was a functional and polymorphic language of the initial proposal, it evolved as a prominent programming language with imperative and sophisticated features. Two major implementations of the meta-language are SML and OCaml [2].

This was the beginning of the interactive theorem provers era, those systems that help users to make proofs while ensuring the correctness of them. Nowadays, they are just called theorem provers or proof assistants, while a more general name is *proof management systems* [3].

The original implementation of LCF included data-types to prove logic formulae: a type to denote a theorem (`thm`) which is a pair of conclusion and hypotheses, a validation which is a function from a list of theorems that implies the theorem to be demonstrated and a *tactic* is a function to transform goals. They are related as follows:

1. We will always make reference to *Edinburgh LCF* just as LCF.
2. The current implementations can be found in `http://www.smlnj.org/` and `http://ocaml.org/`.
3. Inside the community of proof assistants, there is an indistinct use of *proof assistants* and *theorem provers* to refer to the same computational frameworks. The interactive adjective was added to denote an implicit feature of these systems which is nowadays essential. We continue to use them interchangeably.

```
thm = form * form list

goal = form * simplset * form list

tactic = goal -> (goal list * validation)

validation = thm list -> thm
```

Figure 1.1 – LCF data-types

A tactic is a tool, a heuristic to transform a goal into a sequence of simpler goals using the information available in the context of the proof. It includes a validation which is a rule that mimics an hypothetical judgement, using the schema of premises and conclusion. When a tactic is applied to a goal, it replaces the current goal with the premises of the rule, to generate a list of new sub-goals to be proven together with a validation. The validation or witness of the tactic is maintained, is usually the name of the tactic. In this way, at the end of the proof, the witnesses and the order they were applied form a proof-script which will be checked. This process describes a user interaction with the theorem prover to direct the *construction* of a proof.

A useful feature of the system is provided by a mechanism to combine and compose tactics, named then *tacticals*. Since tactics are functions, their composition through a tactical gives a higher-order tactic to ease the construction of complex proofs. It is also used to control how the current list of goals can be focused and merged.

In the LCF theorem prover, a tactic can only be applied to the focused goal, this means that the global state of the system all along the proof is not tracked but used in each step of the proof. Despite the validations, an interpretation of a proof-script, outside the proof-development is meaningless.

Meanwhile, the spread of the above theories and the influence of the ideas in LCF together with a (philosophical) foundation of Martin-Löf in 1971[4], as a system for intuitionistic or constructive proofs, inspired the modern theorem provers.

The Martin-Löf Type Theory is an expressive typed λ-calculus, with a predicative type system, an infinite sequence of type universes and equality for types. Later, more expressive extensions of λ-calculus as the *System **F*** by Girard and Reynolds in the mid 1970's formalised the polymorphism of ML. Then, in the beginning of 1980's the work of Huet and Coquand set the foundations of a theory of dependent types called *The Calculus of Constructions*. Later, in 1985, the first implementation of the Calculus of Constructions appeared as a type-checker for lambda expressions.

Together with this theory, other efforts of many researchers started pushing on the realisation of systems and languages to program efficient decision procedures and more ambitiously, to contribute to the first versions of a theorem prover based on the Calculus

4. Notes and transcriptions of lectures given by Martin-Löf are classical references: 'On the meanings of the logical constants and the justifications of the logical laws' and *Intuitionistic Type Theory*.

of Constructions[5]. In particular, the work of Paulin-Mohring, added some automation to the tactics. Later, she also extended the formal and practical way to describe data-types with inductive constructions and principles in the Calculus of Constructions. This gave the theoretical base of the modern COQ, the *The Calculus of Inductive Constructions*.

Proof management systems

A theorem prover based on Type Theory is an interactive system for symbolic manipulation guided by the user. Its central activity is to build proofs of theorems and to verify formal developments or what we call *theories*.

Any theorem prover has a specification language and is usually composed of two independent parts:

1. a proof-development environment, to construct proofs interactively controlled by the user, and
2. a proof-checker, to verify the proof-term obtained at the end of a proof.

The user gives definitions and properties, all the information needed by the system in order to support and develop a theory. As described before, while trying to demonstrate a statement or theorem, the proof assistant considers the statement as a *goal*. Then, a tactic breaks the goal into simpler sub-goals which in their turn have to be demonstrated. This process continues until the goals are facts easy to proof and finishes when there are no more goals to prove. Up to here, the proof construction process is managed by the first component of the theorem prover, the so-called *proof engine*. Finally, a *composition* of tactics gives a *proof-script* describing the construction of the proof.

The proof-checker or type-checker is the implementation of the type theory in which is inspired the theorem prover, it is called the *kernel*. Then, constructing a proof is comparable with the inhabitant problem, where given a context and a type, then a proof or term is searched. Therefore, a final proof verification by the kernel is needed to ensure that the proof-term obtained is a well-typed term in the type theory. Any proof is type-checked when it is finished. The process generates a *proof-term* corresponding to the proof-script.

We highlight the following facts of the LCF, as ancestor of the modern proof management or development systems like COQ, ISABELLE, HOL, NUPRL, etc. They are also the result of deep analysis and implementation experiences of several authors [116, 9]:

- Tactics are (deduction) rules associating premises to a conclusion. Some systems implement a backward reasoning, others use the forward reasoning.
- The proof-scripts are just a sequence of tactics and system instructions: they are unstructured, sometimes heavy (depending on the user experience) and at the same time fragile since outside the proof, there is no current information to keep track of the proof and the context where a tactic was applied. This makes the proof sensitive to small changes.

5. A detailed history is available in the *Notes on the prehistory of Coq* http://github.com/coq/coq/blob/beedccef9ddc8633c705d7c5ee2f1bbbb3ec8a47/dev/doc/README-V1-V5.

- The *Poincaré* principle states that the proofs carried out in a proof assistant must be verified. A verification of a proof is done until it is completed, by the proof-checker.
- The kernel must be reliable, since the proof verification depends on it. Hence the it is usually a small, isolated and human-certified program. This is called the *de Bruijn* criterion.
- Some proof management systems allow that parts of proofs are achieved by computations. Then, there is no need of verification of those calculations. This is related with the computational power of the type theory on which is based the theorem prover offering great benefits.

Coq

The COQ proof assistant, defined nowadays as a *formal proof management system*, is a well-established interactive system for developing and checking proofs. It is based on a formal language: the Calculus of Inductive Constructions, a powerful type theory.

It provides a framework where the user defines and elaborates theories, offering a safe, strict and formal environment to reason about them. The system helps the user through proofs while it can also generate automatic proofs because of its features to do proof-automation.

The proof-terms obtained after the demonstration process are certified by the *kernel* of COQ. The kernel is essentially a type checker which is the implementation of the Calculus of Inductive Constructions, it takes a term or proof which is treated as an inhabitant candidate for a type or formula.

Thanks to the Curry-Howard correspondence, COQ is also considered as a functional programming language since the proofs are lambda-terms. Moreover, as we said earlier, the type theory behind COQ is highly expressive and there is always a way to perform computations at any level, that is, the convertibility of terms and types is ensured by the conversion rule which characterises the computation or reduction of them.

Many contributors since then, have consolidated this influential framework for defining and formalising theories including a part of mathematics itself. For more historical details and implementation description of COQ, the online documentation[6] can be consulted as well as the *Coq'Art* book [20].

Proofs in COQ

Theorem provers give confidence in the formalisation and the proofs carried out while developing a theory. Their evolution responds to user demands and each system must incorporate user creativity as (permanent) features. Therefore, expanding the mechanisms for proof development like extensions for tactics, facilities for programming or deduction reasoning are desirable. All the tools or techniques for alleviating the process of proof construction must be all sound.

6. http://coq.inria.fr/documentation/

The focus on providing an efficient environment while constructing and checking proofs has been largely explored. There are several parts that can be improved by the system developers, in order to get a better overall performance on proof management.

As proposed by Milner, there should be a language support for writing new tactics in theorem provers that is, achieving efficiency of development by means of improving the tactic language. In COQ we would like to build interactively proofs with more complex constructions and techniques for proof automation. To achieve this, there is a language for tactics named $\mathcal{L}_{tac}$, proposed by Delahaye [36] to enrich the current tactic combinators. Formally, $\mathcal{L}_{tac}$ is a higher order language to design domain specific tactics. Nowadays, most of the provided tactics are definitions of such combinations of primitive tactics.

However, one of the shortcomings of $\mathcal{L}_{tac}$ is that tactics used to construct proofs are unsafe[7]. This language can be improved by means of types for tactics as is suggested by Delahaye.

Efficiency in theorem provers can also be gained by improving the machinery to guide and verify proofs, by means of improvements in the interaction with the kernel and the tactic engine. Improvements for system development have many contributions: libraries for specialised theories (see the list of users' contributions[8]), tactic languages to enhance the mechanisation of proofs [36, 118], improvements to the user back-end framework for example the asynchronous edition of proofs [17], interaction with other theorem provers to cooperate in a large developments, etc. All of this turns COQ into a sophisticated programming language where the program development arises naturally, for instance the RUSSELL extension [101] to develop programs with dependent types.

1.2 This thesis

We have highlighted that the functional paradigm provides a formal approach to programming and ensure correct programs. Unfortunately, it disregards to be used while solving low level problems or when incorporating imperative techniques to obtain efficient solutions.

Besides, the effects offered by a low-level language, unavoidable when reaching the innermost practical aspects of computing, need to be studied and therefore incorporated in the models of pure lambda calculus to represent and to reason about programming mechanisms.

The focus of this work is on formal techniques that extend the models of the simply typed lambda calculus, inspired by some practical motivations taken from the everyday programming, and with the aim to be applied in a system for formal proof assistance.

Our goal is to reason about two improvements to be carried out in a total functional programming language: to verify the *simulations* of effects *a posteriori* and the optimization by *incrementality* through data-change description.

7. As pointed out in various works, there are disadvantages with the treatment of tactics inherited form LCF, for example the exhaustive comparison made by Harrison [47].

8. https://coq.inria.fr/contribs/

The Paral-ITP project This work was supported mainly by the ANR project Pervasive Paralellism in Highly-Trustable Interactive Theorem Proving Systems. The project involves the interactive theorem provers ISABELLE and COQ and its main goal is to overcome the sequential model of both proof assistants to make the resources of multi-core hardware available in large proof developments[9].

The project proposal stated a PhD student to work towards the conception and study of a *formal repository* based on Logical Frameworks (LF) [46]. The formal repository, also named the prover repository, was planned as an entity which must be highly sensitive to handle the dependencies between different versions of terms. All actions performed over the repository must be logically sound: history management and operations like fine-grained requests and the propagation of (non-sequential) changes must ensure a sound control of formal content.

The above ideas originate the research reported in this thesis where its concrete and future goal is to obtain a certified type-checker for COQ. This is a bigger project which can be divided in two parts, the first to improve or make a new proposal for a tactic language and the second to achieve the formalization of a prover repository. The idea of a formal repository depends on the design of a typed language for tactics, which will provide safe proof-scripts in order to facilitate their use and control within the repository.

A dependent type theory, as LF or the Calculus of Inductive Constructions, could serve as a robust theory for a formalization of the repository and its operations as suggested by the project proposal. While the rigorous frameworks of LF and VeriML [102] can be the basis for the formal repository, we chose another line of investigation where research was conducted in the direction to define a typed language to improve the tactic language of COQ. This results in a proposal of an effectful language to write decision procedures whose formalization will be elaborated in the first part of this thesis.

One of the project's main goals described above seeks to improve COQ by giving a new way of certification of proof-terms to be used in the repository. We chose to start the research towards the improvement of proof-development by setting an incremental framework, which one future application will allow the incremental construction and certification of proofs. The incremental flavour is inspired by a new model of computation where programs automatically respond to changes in their data. A recent paradigm supporting incrementality is the so-called self-adjusting computation which uses several techniques to write self-adjusting programs as normal programs. The second part of this thesis introduces a language where incremental computations are part of the evaluation process.

The ideas of changeable data and incrementality could lead a formal repository, in where fragments or complete proof-terms, which are already certified, are stored in order to improve the type-checking process. As we said, this work is pretended to be done in collaboration of a typed language for COQ's tactics to construct safe proof-terms throughout the proof development process.

The contribution of this thesis states an initial theoretic support to the formal repository, a long term goal to be elaborated in the future. In the following, we briefly describe

9. For more information visit http://www.lri.fr/~wolff/projects/ANR-Paral-ITP/.

the contributions which are focused on providing methods for reasoning within effect simulation and incrementality in the simply typed lambda calculus.

Cybele: lightweight proof by reflection

This work conciliates imperative features in the pure and total theory of COQ by adding effects through monads like in purely functional approaches. This enhancement allows the user to write efficient and trustworthy programs in a certifying environment. Moreover, since in theorem provers the construction of proofs (or programs) can be simplified by performing computations at type level, the enhancement is used in proof development using the reflection technique.

The joint work with Yann Régis-Gianas, Guillaume Claret and Beta Ziliani states a way to describe effectful decision procedures. The main idea is to use an untrusted compiled version of a *monadic* decision procedure written in Type Theory within an effectful language as an efficient oracle for itself. The evaluation of decision procedures is executed with the help of what we call a *prophecy*, acquired by the execution of an instrumented code in an impure programming language (OCAML). The prophecy is a small piece of information to efficiently simulate a converging reduction in Type Theory. This work is the Cybele project [10] which results in a new style of proof by reflection characterised by a lightweight simulation of effectful programs.

The contribution described in this thesis, is to analyse and state the requirements of simulation by studying the relation between two languages, one representing Type Theory without effects and the other representing a general purpose effectful language. The notion of *a posteriori simulation* is the main support on which rests a new style of proof-by-reflection. A formalisation of the lightweight approach to proof by reflection enables an extension of the lambda calculus with monads and an operator to simulate computations.

Incrementality

Among the computational optimizations, an intuitive technique is to avoid repeated computations by data re-use. Incremental computation as a programming paradigm, takes advantage of similar computations in order to reduce the costs. The incrementality can be achieved in several ways and can be present in explicit program transformations or in implicit features of some program developments.

An analysis of many efforts to add incrementality in computations leaded to propose a functional approach by a differential treatment of data and programs, that is a combination of a *change theory* for data dissection and a formal language for program *differentiation*.

The contribution to incremental computation is achieved by dynamic differentiation of functions to take advantage of computed results from old program inputs and function differentials. It is inspired in the differential lambda calculus [38] and in a type-directed change description. A new system meeting these ideas, λ-diff, is defined and of-

10. Visit http://cybele.gforge.inria.fr/.

fers to the user the ability to reason about fine-grained input changes which are reflected into efficient computed results. The computation by means of a gradual and steady sub-computations, led by *smooth* input transformations. The language λ-diff, exposed in this thesis, is a framework for analysis and reasoning about incremental computation and is not intended to be an optimal implementation of incrementality.

Thesis Organisation

Following the above topics, the thesis is organised in two main parts, one for reflection by simulation (Chapters 2 and 3) and one for incrementality (Chapters 4 to 7).

Each part is self-contained, however the foundations of both are commented in the next section for preliminary theoretical concepts.

1.3 Preliminaries and General Notations

In this section, we present some basic notions to level the knowledge of the reader with the concepts used through this work. Other concepts are more common to be included as background, such as the formal system which is the base for the languages in this work, the simply typed λ-calculus[11]. Nevertheless, the chapters in this work are self-contained and in what follows, we restrict ourselves to present a brief description where some statements of properties are given without their proofs.

Relations

Given a set $\mathcal{S}$, a binary relation $\mathcal{R}$ is a collection of pairs of elements in $\mathcal{S}$, we use an infix notation for relations. Given a relation $\mathcal{R}$ we say that $\mathcal{R}^*$ is the reflexive and transitive closure of $\mathcal{R}$.

The notation $\overline{e}$ is used to denote a finite sequence or a vector of elements $e_0\, e_1\, \ldots\, e_n$ where $n \geqslant 0$. The length of a vector is the number of its elements. If a function $\mathcal{F}$ is defined over e then we abusively write $\mathcal{F}\,(\overline{e})$ for the pointwise extension of $\mathcal{F}$ to a sequence of elements.

Simply typed lambda calculus

λ-calculus as an abstract system to represent and perform computations is a model which was first formulated as a pure theory, now distinguished as the *untyped* lambda calculus. It is composed by variables, abstractions or functions, and applications of terms. There are no constants nor any other primitives.

The terms created under these three basic elements are infinitely many and some of them are not adequate for characterising the solution to a given problem, for instance consider an application of a function with the wrong type of arguments.

11. The conventional references are: H. Barendregt [12, 13, 14], B. Pierce [92], R. Harper [45], among others [11] who traditionally recall the pillars of the programming language theory.

We want to get rid of those terms in order to prevent misbehaviours while encoding programs. Therefore, imposing syntactical restrictions to the terms in order to reduce the non-sense terms or wrong constructions, gives an accurate system where *simple types* limit the terms to those which make sense to compute. This idea was coined by Milner in the famous phrase 'Well typed programs do not go wrong'.

Simply typed λ-Calculus, *à la Church*, is the language generated by the constructions in the grammar of Figure 1.2 whose principal syntactic classes distinguish terms and types. The terms are built up from typed variables, lambda abstractions binding a single variable to a body sub-term and applications of a left sub-term to a right sub-term. The types include *abstract* basic types denoted by ι and the functional-type with hypothesis σ and conclusion τ types. The third syntactic class encompasses the class of typing contexts which are collections of all distinct typed variables.

$$\boxed{\text{Syntax}}$$

$$\begin{array}{rcl} t,\, r,\, s & ::= & x \mid \lambda x^{\sigma}.\, s \mid r\, s \\ \tau,\, \sigma & ::= & \iota \mid \sigma \to \tau \end{array} \qquad \Gamma \;\; ::= \;\; \varnothing \mid \Gamma,\, x^{\tau}$$

$$\boxed{\text{Static Semantics}}$$

$$\frac{x^{\tau} \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma,\, x^{\sigma} \vdash s : \tau}{\Gamma \vdash \lambda x^{\sigma}.\, s : \sigma \to \tau}\ \textsc{Abs} \qquad \frac{\Gamma \vdash r : \sigma \to \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash r\, s : \tau}\ \textsc{App}$$

Figure 1.2 – Simply Typed Lambda Calculus

Static semantics

We choose to present first the *static* semantics, that is the rules for type assignation, to give to the reader an attachment to a notion that will be implicit in this work, the treatment of well-typed terms. The static semantics are *typing judgements* relating terms to types under typing contexts. The judgement $\Gamma \vdash t : \tau$ is read as 'the term t has type τ under the context Γ' and the rules are defined inductively on the form of terms. A variable carries her own type and a lambda abstraction has a function-type constructed with the type of the linked variable and the type of the body sub-term. The type of an application results from the types of its sub-terms: if the left sub-term has a function-type and the right sub-term agrees with the type of the hypothesis then, the type of the application is the conclusion type of the function-type.

Notation A lambda abstraction is generalised to multiple binding: $\lambda x_0.\, \lambda x_1.\, \ldots \lambda x_n.\, t = \lambda x_0,\, \ldots,\, x_n.\, t \;=\; \lambda \overline{x}.\, t$. The syntactic equality between elements is denoted by $=$.

A multi-argument function is represented in a *curried* form if its arguments are expected one by one that is, the function is a chain of λ-abstractions. The *uncurried* version of the multi-argument function expects all the arguments in a tuple.

Operations

Operations over terms are essential for reasoning, here we present the operations used in this work. Each of these definitions can be extended according to the new elements of the corresponding system extensions addressed in later chapters.

Definition 1.1 (Free and Bound variables)
The set of free variables of a term $\mathsf{FV}\,(t)$ *is defined inductively:*

$$\begin{aligned} \mathsf{FV}\,(x^{\tau}) &\overset{def}{=} \{x^{\tau}\} \\ \mathsf{FV}\,(\lambda x^{\sigma}.\, s) &\overset{def}{=} \mathsf{FV}\,(s) \setminus \{x^{\sigma}\} \\ \mathsf{FV}\,(r\, s) &\overset{def}{=} \mathsf{FV}\,(r) \bigcup \mathsf{FV}\,(s) \end{aligned}$$

Bound variables are those which are not free.

Definition 1.2 (α-equivalent terms)
Any two terms are α*-equivalent if they are syntactically equal up to the renaming of their bound variables.*

Definition 1.3 (Substitution)
The substitution of a variable by a given term in another term is defined inductively, a renaming of bound variables in λ*-abstractions is considered before the substitution.*

$$\begin{aligned} x\,[x\ :=\ t] &= t \\ y\,[x\ :=\ t] &= y \\ (\lambda y^{\sigma}.\, s)\ [x\ :=\ t] &= \lambda y^{\sigma}.\, s\,[x\ :=\ t] \qquad \textit{where } x \neq y \\ (r\, s)\ [x\ :=\ t] &= r\,[x\ :=\ t]\, s\,[x\ :=\ t] \end{aligned}$$

Dynamic semantics

The concept of computation is realised by means of input processing by applying functions. There are two main ways to describe the process of obtaining outputs: by a precise description of every computation made until an irreducible expression is reached, named a *reduction* relation between two terms; or a description of the *values* obtained at the end of the process, named an *evaluation* of a term.

A value is an object which cannot be reduced any more, it is impossible to apply at least one more reduction step. The collection of values is a *syntactic class*, distinguishing those objects form the rest of terms in the grammar defining the language. In the case of the simply typed lambda calculus, the values are the abstractions, but for instance in a calculus including the natural numbers as primitives, the numbers and constant functions are also values.

A normal form is also a term on which it is not possible to make progress, that is, there does not exist a term t' from which a given t is reduced by a stepping rule. The collection of normal forms is a *semantic* distinction between terms, it is defined by a proposition and not directly in the syntax as the definition of values.

The two notions of dynamic semantics, for a call-by-value strategy, are formalised by small-step semantics and big-step semantics whose respective rules appear in Figure 1.3.

$\boxed{\text{Small-step semantics}}$

$$v \ ::= \ \lambda x^{\sigma}.\, s \qquad\qquad \mathcal{E} \ ::= \ [\,] \mid v\,\mathcal{E} \mid \mathcal{E}\,t$$

$$\begin{array}{c}\beta\text{-RED}\\ (\lambda x^{\sigma}.\, s)\ v \longrightarrow s\,[x \ := \ v]\end{array} \qquad \textsc{Red}\ \frac{t \longrightarrow t'}{\mathcal{E}[t] \longrightarrow \mathcal{E}[t']} \qquad \begin{array}{c}\textsc{Refl}\\ t \longrightarrow^{*} t\end{array}$$

$$\textsc{Trans}\ \frac{t_1 \longrightarrow t_2 \qquad t_2 \longrightarrow^{*} t_3}{t_1 \longrightarrow^{*} t_3}$$

$\boxed{\text{Big-step semantics}}$

$$v \ ::= \ (\lambda x^{\sigma}.\, s)\,[\eta] \qquad\qquad \eta \ ::= \ \bullet \mid \eta\,;\, x \mapsto v$$

$$\frac{}{\eta \vdash \lambda x^{\sigma}.\, s \Downarrow (\lambda x^{\sigma}.\, s)\,[\eta]} \qquad \frac{\eta \vdash r \Downarrow (\lambda x^{\sigma}.\, t)\,[\eta'] \qquad \eta \vdash s \Downarrow u \qquad \eta'\,;\, x \mapsto u \vdash t \Downarrow v}{\eta \vdash r\, s \Downarrow v}$$

Figure 1.3 – Dynamic Semantics for a call-by-value strategy.

The small-step semantics reduces a term by making explicit each computation. The use of *evaluation contexts* η, emphasizes the reduced expression or *redex* in the term. The contexts and reduction rules define the call-by-value strategy of reduction, where each term has a left-to-right order of reduction and the β-reduction makes progress simplifying a function: introduces an input value in the body-term of the function by using the substitution operation.

A full reduction of a term is obtained after applying exhaustively the reduction rules and it is defined by the reflexive and transitive closure of reduction ($\longrightarrow$).

The big-step operational semantics makes use of an environment η, a partial function from variables to values. The values in this calculus are functions that depend on the context that is, they are λ-abstractions all along with its current environment of evaluation $[\eta]$, called *closures*.

We write $\eta \vdash t \Downarrow v$ where $\Downarrow$ relates a term t and a value v under a given environment η whose domain is the set of free variables in t. The relation is defined inductively over terms: a variable is evaluated to its corresponding value in the environment, a lambda-abstraction is evaluated to a closure and the evaluation of an application rests in the evaluation of the function-body of its left sub-term under an extended environment with the value of the right sub-term evaluation.

It is possible to recover from small-step semantics the big-step semantics, and vice versa, that is they are equivalent by means of value-ending chains of reductions using $\longrightarrow^{*}$ in the former semantics and the $\Downarrow$ relation in the latter.

Properties

Simply typed λ-calculus enjoys many properties derived from the semantic relations. From type assignation, the traditional lemmas of inversion of typing and uniqueness of types to ensure well-typed terms. Together with small-step semantics we have the properties of determinism and subject reduction also known as type preservation. Each extension of λ-calculus used in this work contains the corresponding statements and proofs for these properties, they are stated and proved in the appropriate places in future sections.

Logical Relations

Logical relations can be defined as predicates to describe properties of terms. We follow the notation of logical relations used by Ahmed [8] and we make some recalls to conduct the reader toward the work of Reddy et al. [49] who stress the closer relation between logical relations and Reynolds work on parametricity. The last authors consider the logical relations and parametricity as a more general form of abstraction which enables *information hiding* through the structure over which the relation is defined.

One of the applications of logical relations is the proof of meta-level theorems [97]. Most of the applications are adopted to prove theorems of type theory where the (traditional) method of proving by induction over one syntactical structure as terms, types or judgements do not give strong enough induction hypotheses.

There are different choices between languages on which interpret a type theory, referred also as the object language. One useful choice is to take the same type theory as meta-language. Other choices consider the set interpretation of types or a domain interpretation. The case of set theory as meta-theory interprets the type theory as:

$$[\![\iota]\!] = \mathcal{X} \quad \text{for each basic type and a given set } \mathcal{X} \qquad [\![\sigma \to \tau]\!] = [\![\sigma]\!] \Longrightarrow [\![\tau]\!]$$

In order to have more logical relations than just a set and function spaces $\Longrightarrow$, we add some relation operators like the set operators: product $\times$, sum $+$, power set $\mathcal{P}$, predicates $\widehat{\mathcal{Q}}$, among others. On the one hand, logical relations do not require to compose, but in the other hand they could have a *mapping operation* known as *parametric transformation* [49].

The logical relation approach to proofs is a two step procedure. The first step is to define a logical relation adapted to the target type theory and to the property to demonstrate. This is done inductively by defining a type-indexed family of relations. In the case of a theory of the simple typed lambda calculus, we consider that any logical relation at least must detail and extend the following definition:

Definition 1.4 (Logical Relation)
A logical relation is a family of type-indexed n-ary relations $\mathcal{R} = \{\mathcal{R}_\tau\}$ such that the terms of a n-tuple have the same type and

- *if τ is a basic type ι, then $\mathcal{R}_\iota \subseteq [\![\iota]\!]^n$*
- *if τ is a function type, then $\mathcal{R}_{\sigma\to\tau}\,(r_1, \ldots, r_n)$ if and only if for all n-tuples $(s_1, \ldots, s_n)$, if $\mathcal{R}_\sigma\,(s_1, \ldots, s_n)$ then $\mathcal{R}_\tau\,(r_1\, s_1, \ldots, r_n\, s_n)$.*

Logical relations are defined strictly following the structure of types and this structure is reflected and preserved in the relation. In other words, the above definition and extensions of it comprises the well-typing of a term, a condition for a term to has the property of interest and a condition to ensure that the logical relation is preserved by evaluation of elimination forms.

The second step is to show the completeness and soundness of the logical relation: that any well-typed term of the theory belongs to the relation and that any element which belongs to the relation has the desired property.

Lemma 1.1 (Completeness of a Logical Relation)
Consider a well-typed term, $\Gamma \vdash t : \tau$, then $\mathcal{R}_\tau\,(t)$.

Lemma 1.2 (Soundness of a Logical Relation)
Any term such that $\mathcal{R}_\tau\,(t)$ has the property characterised by the relation.

Coq

As commented before, the Curry-Howard correspondence is a principal paradigm on which Coq is based. The constructive logic which used by Coq is the Calculus of Inductive Constructions. This calculus has only one syntactic category for types and terms [12]:

$$\begin{array}{rcll} t,\ r & ::= & s & \textit{sort} \\ & & \mid x \mid c \mid \mathcal{C} \mid \mathcal{I} & \textit{identifiers} \\ & & \mid \forall x^t.t \mid \lambda x^t.r \mid \mathsf{let}\ x = r\ \mathsf{in}\ t \mid t\,r & \\ & & \mid \mathsf{case}\,t\ \mathsf{of}\ r_0 \ldots r_n & \textit{elimination} \\ & & \mid \mathsf{fix}\ x\ \{x^r : \overline{y} := t\} & \textit{recursion} \end{array}$$

There are three sorts in the calculus: Prop, Set and Type, which represent the propositional or logical level of the language and the hierarchical universes of types, respectively. The sort Type has an enumeration according to the type level needed in the theory.

The identifiers give the global definitions of the language and are included in contexts when doing proofs. The identifiers are variables and names for constants, constructors and types. Together with the third syntactic category built the terms of the language. The computational terms include the product $\forall$ which is an upper-level abstraction for function definition while the other terms are the usual constructions for abstraction, term application and local definitions (let). The elimination form case, goes together with the inductive definitions introduced by constants $\mathcal{I}$. Finally, another elimination form is recursion and is performed by a fix-point term.

A context, while constructing a proof, includes hypotheses as variables $x : t$, definitions of constants $c := t : r$ and inductive declarations $\mathcal{I}\,(\Gamma_\mathcal{I} := \Gamma_c)$ where the context $\Gamma_\mathcal{I}$ contains the inductive types and Γ_c the corresponding constructors.

Any term in this calculus has a type and the evaluation always terminates. This condition is continuously verified syntactically by the system through the *positive* requirement

12. We follow the presentation used by Letouzey [61] and the CIC language description in the reference manual: `http://coq.inria.fr/distrib/current/refman/`.

for types and by *well-founded definitions* for elimination forms. The recursive calls must show a clear use of the function over a sub-component of the parameter.

The dynamics of the calculus, the reduction of terms, is done mainly by a *strong* call-by-value strategy. This kind of reduction differs from the one described previously in our presentation of λ-Calculus, since in the Calculus of Inductive Constructions the reductions are carried out pervasively. This means that the body-terms of λ-abstractions are likely to be reduced.

The system provides other reductions as the ι, δ, ζ or η reduction rules, each one to reduce respectively: inductive terms, definitions, let-terms and to permorm the η-expansion of the λ-calculus.

The whole combination of these rules defines the conversion rule stating that two expressions are equivalent $t \equiv s$. The relation $t \equiv s$ is read as t *is convertible with* s and is a reflexive, symmetric and transitive relation over the above strong reduction rules. Then, two terms are convertible or equivalent if they are reduced to identical terms or are convertible up to η-expansion:

$$\textsc{Conv}\ \frac{\Gamma \vdash \mathsf{U} : \sigma \qquad \Gamma \vdash t : \mathsf{T} \qquad \Gamma \vdash \mathsf{T} \equiv_{\beta\delta\iota\zeta\eta} \mathsf{U}}{\Gamma \vdash t : \mathsf{U}}$$

where the relation $\equiv_{\beta\delta\iota\zeta\eta}$ represents the set of reduction rules in the language.

Monads

The purely functional approach reviewed up to this point does not allow a representation of *computational effects* as memory location, perform input/output, error handling or exceptions, non-determinism, etc.

The monadic approach to functional programming proposed by Moggi [76] offers an extension and interpretation of a λ-Calculus to represent computations by abstracting a program logic inside a structure. The data-type constructor $\mathsf{M}\,\tau$ represents computations that will produce a value of type τ. It consists of two operations, one named the unit-operation to encapsulate an expression as a trivial computation and another operation to perform and compose computations named bind. These operations allow to explicitly describe sequential computations.

A single effect is an instance of a monad which needs a specific definition for the two operations of unit and bind. Any monad holds the three monadic laws characterising the equivalences between computations.

In Figure 1.4 is depicted the extension of λ-Calculus with the terms for monads. The dynamic semantics uses the strong reduction where the values include unit terms.

References in the literature of monads include the work of Wadler [111, 113]. The success of this theoretical approach is used in practice in several (functional) programming languages to ease a structured programming, such as in HASKELL [65, 81]. In this language, the operations $\mathsf{unit}\ t$ and $\mathsf{bind}\, r\, s$ are written `return t` and `r >>= s` respectively. Using the type-class abstraction for polymorphism, to regroup types and manage overloading of functions, the `Monad` class describes four monadic operations:

$\boxed{\text{Syntax}}$

$$\begin{array}{rcl} t,\, r,\, s & ::= & \cdots \mid \mathsf{unit}\ t \mid \mathsf{bind}\, r\, s \\ v & ::= & \cdots \mid \mathsf{unit}\ v \\ \tau,\, \sigma & ::= & \cdots \mid \mathsf{M}\,\tau \end{array} \qquad \begin{array}{rcl} \mathcal{E} & ::= & \cdots \mid \mathsf{unit}\ \mathcal{E} \mid \mathsf{bind}\,\mathcal{E}\, s \mid \mathsf{bind}\, v\, \mathcal{E} \end{array}$$

$\boxed{\text{Static Semantics}}$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{unit}\ t : \mathsf{M}\,\tau\ \mathsf{unit}}\ \textsc{Unit} \qquad \frac{\Gamma \vdash r : \mathsf{M}\,\sigma \qquad \Gamma \vdash s : \sigma \to \mathsf{M}\,\tau}{\Gamma \vdash \mathsf{bind}\, r\, s : \mathsf{M}\,\tau}\ \textsc{Bind}$$

$\boxed{\text{Dynamic Semantics}}$

$$\begin{array}{l} \textsc{Comp} \\ \mathsf{bind}\ (\mathsf{unit}\ v)\ (\lambda x^{\sigma}.\, s) \longrightarrow s\,[x\ :=\ v] \end{array}$$

$\boxed{\text{Monadic Laws}}$

$$\begin{array}{rcl} \mathsf{bind}\,(\mathsf{unit}\ t)\, r & = & r\, t \\ \mathsf{bind}\, t\ (\lambda x^{\sigma}.\, \mathsf{unit}\ x) & = & t \\ \mathsf{bind}\ (\mathsf{bind}\, t\, s)\ r & = & \mathsf{bind}\, t\ (\lambda x^{\sigma}.\, \mathsf{bind}\ (s\, x)\ r) \end{array}$$

Figure 1.4 – Monads

```
class Monad m where
 (>>=)  :: m a -> (a -> m b) -> m b
 (>>)   :: m a -> m b -> m b
 return :: a -> m a
 fail   :: String -> m a
```

Any instantiation must declare the definition of the >== operation. Also, the popular do-notation is used to give structure to monadic programs. This is a syntactic sugar to hide the details of bind-applications and to make clear the compositions. For instance the term `bind t1 (\x -> bind (f x) (\y -> return y))` can be rephrased as:

```
do
    x <- t1
    y <- f x
    return y
```

There is a large range of effects that can be represented by monadic definitions. For example, the effect representation where programs have an optional outcome or where functions can return or not meaningful values, is done through the monadic type `Maybe` (in HASKELL) or the `Option`-type (in ML-like languages). The instantiation in HASKELL, of the monadic operators is the following:

```
data Maybe a = Nothing | Just a
```

```
instance  Monad Maybe  where
    (Just x) >>= k    = k x
    Nothing  >>= _    = Nothing
    return            = Just
    fail _            = Nothing
```

The one-effect representation by one monadic type can be generalised into a multiple-effect monad by the combination of effects using *monad transformers*. It is a type constructor that takes a monad and returns a new monad by 'lifting' the monad, it is defined in HASKELL also as a type-class:

```
class MonadTrans t where
    lift :: (Monad m) => m a -> t m a
```

This generalisation is done by taking an existing monad `m`, and encapsulating the type into another monad `t`.

For the `Maybe` type, the monad transformer is defined by encapsulating the optional value inside the monadic argument `m`:

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

instance Monad m => Monad (MaybeT m) where
  return  = MaybeT . return . Just
  x >>= f = MaybeT (do maybe_value <- runMaybeT x
                       case maybe_value of
                            Nothing    -> return Nothing
                            Just value -> runMaybeT (f value))

instance MonadTrans MaybeT where
     lift m = MaybeT (liftM Just m)
```

Lightweight proof by reflection using *a posteriori* simulation of effectful computation

Chapter 2

Proof by reflection

Among the problems a computer scientist faces to, there are those whose solution is achieved by deciding if an object has a property or not, that is where the computations are focused to produce a 'yes' or 'no' answer for a given statement[1]. For instance, consider the following problem:

Congruence problem *Given a set $\mathcal{S}$ of equivalent terms $s_i \sim s_j$, determine whether or not two given elements s and s' are equivalent from set $\mathcal{S}$.*

The word *equivalent* refers to the fact that a pair of objects are considered as equals because they share the same properties or have the same behaviour under a specific scenario, not only because they could have the same form or syntax. This problem does not examine the properties of the objects but just the fact that they are related, in order to *decide* if a given equivalence follows from a set of equivalences.

The meaning of equality between elements arises when we can relate them directly by definition, that is when $s \sim s'$ belongs to $\mathcal{S}$, or because we can *go* from s to s' by passing through some equivalences in $\mathcal{S}$. This is better known as the Leibniz equality, the ability of *substituting* equals by equals. Then, we say that the equivalence $s \sim s'$ follows from $\mathcal{S}$ when a *path* between the elements s and s' is found. The notion of paths is derived from considering the relation $\sim$ as an equivalence relation: any element is equal to itself (reflexivity), if two elements are equal $s_1 \sim s_2$, then they are also related the other way around $s_2 \sim s_1$ (symmetry) and given two equal elements and one of them is equal to a third one, then the first element is equivalent to the third element: $s_1 \sim s_2$ and $s_2 \sim s_3$ then $s_1 \sim s_3$ (transitivity).

A popular and efficient solution to this problem is achieved by computing the equivalence closure of the set of equivalences using the union-find algorithm due to Tarjan et al. [37, 105, 104]. This method computes the equivalence classes of the set $\mathcal{S}$ and then decide if the representatives of the classes of terms s and s' are the same.

It is based on two principal operations: one to merge two equivalence classes, by unifying the representatives of the classes named the union operation, and a second operation named find, to search for the representative of the class of a given element. Hence, this algorithm is called the union-find algorithm.

1. We are not considering decision problems like the Halting Problem, nor even problems in the spirit of Logic Programming.

There are many variations, abstractions and applications inspired on the couple union-find leading to efficient implementations. A range of problems get better run time when solved by computing a closure, and also the so called disjoint-set data structures, to maintain a structure for the elements and their representatives, lead to better data handling ([78] and [35], chapters 22 and 24).

Let us analyse some possible implementations of the intuitive solution of path finding for the equivalence problem. The following is a first and very naive pseudo-code approximation of the algorithm described above.

```
is_equivalent(S, <s, s'>) : Bool
   C := singleton(S)
   while  C != S  do
      A := C
      for <si,sj> in S do union(C,si,sj)
      S := A
   return (find(C,s) == find(C,s'))
```

The function `is_equivalent` expects two arguments, the set of equivalences together with the pair of elements to be tested, and it returns a boolean value. We start by creating an initial closure where each element is its own representative, this is achieved by the function `singleton`[2]. Then we have a `while`-loop to compute the closure of `S`, guarded by the comparison between the actual closure `C` and the previous one `S`. The `union` function computes the representatives and we use it to create a new closure at each iteration. The loop stops when the closure does not change after the `union` operation. Then, the closure is `C` and we can verify if the representatives of the elements `s` and `s'` are the same or not by comparing the results of the `find` function.

To implement this pseudo-code, the programmer may expect a comfortable way to bring an optimal implementation where coexist all the benefits and components of effectful and imperative languages. For instance features as control-flow operators and a global state, where a specialised data-structure is available to maintain the elements of $\mathcal{S}$ together with their representatives.

A 'more elegant' implementation is possible by means of functional program with *effects*, a development admissible using a *monadic* extension of the language, as HASKELL offers. Here is another pseudo-code, similar to the one given above:

```
is_equivalent :: Monad m => a -> a -> UnFndST a m Bool
is_equivalent S (s, s') =
   do x <- iter (\x-> \y -> union x y) S
      u <- find x
      v <- find y
   return (u == v)
```

2. In a formal way, we can denote $|\mathcal{S}|$ as the set of single elements of a set of equivalences $\mathcal{S}$: if $s_i \sim s_j \in \mathcal{S}$ then $s_i \in |\mathcal{S}|$ and $s_j \in |\mathcal{S}|$.

This program shows an imperative encoding using the monadic approach. The data-structure `S` and all the functions are operations of a monad. The do-notation is a syntactic-sugar to ease the use of effectful operators belonging to a specific monad, which in this case is the monad transformer `UnFndST a m Bool` parametrized by the type of the elements in equalities `a`, a monad `m` to keep the state of the structure for the representatives and the boolean type for the returning value [3].

The work of Conchon and Filliâtre [31], proposes a persistent data structure to maintain a class-partition of a given set. This functional approach also incorporates imperative features (effects) in a functional paradigm to provide an efficient solution to union-find problems.

We have described, quickly, two possible implementations of the algorithm. The first one under an imperative setting and a second one that mimics a structural approach but that belongs to a functional setting. These programming approaches could be prejudged to be distant and opposed since the paradigms of imperative and functional programming have been historically confronted, but this not the case in the pseudo-code exposed so far, as the reader can see.

However, we can say that there is a third paradigm which is becoming attractive in the programming community. As introduced earlier in this work, the certified programming approach to problem solving provides a strict but secure development setting where the programmer can trust the correctness of programs. In this spirit, we can use a theorem prover to solve a subset of problems which are matter of deciding properties such as our problem.

The reader can argue that the functional approach is enough in order to ensure that the implemented program meets the specifications of an algorithm, but in the following we will explore another *elegant* [4] implementation using a theorem prover which appeals to the expressive power of a dependent type theory.

In order to solve the congruence problem *via* a theorem prover, the programmer sets up a proper framework with definitions and auxiliary statements, then declares a statement (a theorem or a type) which describes the problem. At the end, a program is constructed as the proof of the statement inside the particular framework is completed. Now, the implementation of a solution is a matter of producing a proof. Let us explore some ways to construct such a proof.

First, we suppose that a framework or theory for equivalences is developed in the theorem prover, in this case Coq [5]. Then, in the traditional way of proof construction, founded by the Logic for Computable Functions, to prove the statement program we use a rewrite tactic thoroughly until the wanted equivalence is found or not. This is carried out by the programmer who repeatedly types rewrite H_i whenever a hypothesis H_i is available to make progress in the proof.

3. There is a package developed by Thomas Schilling with the implementation of the union-find algorithm: `http://hackage.haskell.org/package/union-find`.

4. This time, the elegant qualifier for the certified programming approach is subjective.

5. For example the library Equivalence by Matthieu Sozeau, a Type-class for setoids.

Thanks to the mechanism for combining tactics (tacticals), there is another way to prove the statement. Tacticals factorise the repeated tactics and therefore alleviate the (tedious) interaction between the programmer and the theorem prover.

We recall an important characteristic of Coq discussed in the introduction chapter, the conversion rule which performs reductions on every term of the language and therefore types are likely to be evaluated. Using this feature, another way to construct a proof for our theorem is to perform *computations* under the theory for equivalences, resulting in a proof that would look as follows: simpl; reflexivity. This tactical reduces the goal and the new goal to prove is an equality which can be solve by reflexivity.

This proposal to conduct the proof is well known as *reflection*, because after the transformation of the goal by reduction, that is using a version of the compute tactic, most of the time the proof is usually finished by an application of the reflexivity tactic. The program that corresponds to the above two-tactic-proof seems to be a more natural sketch of proof using a theory of equivalences.

In this chapter we will take a closer look at the technique of proof by reflection, which inspires the work exposed in the first part of this thesis. We start by explaining roughly the concept of reflection to narrow its use in theorem provers, specifically in the Coq system.

2.1 Reflection

As just described, regarding proof construction, most of the work is done by the user whose reasoning can be exploited to guide proofs even under automation. In the case of the Coq theorem prover, several efforts have been made to incorporate techniques to ease the process of proof construction. One of these techniques considers the practical utility of *computational reflection* to write decision procedures [23, 20, 80, 48].

The intricate concept of computational reflection, or simply *reflection*, can be explained *grosso modo* as a self-optimisation technique or as a process of reasoning and acting upon itself [100]. In her thesis [71] and further research [6], Maes elaborates a general concept of computational reflection: 'an activity performed by a computational system when doing computations about its own computation'.

Other authors, specialised in the field of theorem proving, prefer to identify a two-level language abstraction, composed of the so-called object-language and meta-language [7]. For instance, Harrison gives an accurate definition of reflection as a technique that employs a *meta-logic* to analyse and simplify proofs and appeals to specialised decision algorithms [47].

Certainly, the above attempt to find and establish a general definition of computational reflection is entangled and for sure, leaves the reader obfuscated. We hope to unravel this definition in the rest of the chapter.

6. http://www.idi.ntnu.no/grupper/su/courses/dif8901/presentations2001/t05-maes.pdf
7. The meta-language is a language to make statements about statements of the object-language.

As starting point, we go back to the definition of Harrison where logical theories are useful as their own meta-theory to make and prove statements about themselves. The two-level abstraction in theorem provers like Coq relates the Calculus of Inductive Constructions as meta-level and the computational model as object-level. Therefore the reflection is done in between these levels: a property or theorem stated at the object level is proved by reflecting a similar proof of the translation of the property in the meta-language.

As we will see, this approach of proof production differs from the traditional proof style, where each step in a proof is correct thanks to a tactic. In the following we are going to explore two different approaches of proof by reflection.

2.2 Proof by computation

The reflection technique in its form of *proof by computation* is used by Coq since types may embed computation, as we saw in the certified programming approach to the solution of the equivalence problem.

Proving a statement by reflection is carried out by doing computations until the proof has converged into a last goal which is solved by a test of equality, commonly as an instance of the *reflexivity* of equality. Therefore, some proof steps are replaced by computations which reduces the size of proofs and also the time of proof-term type-checking whose verification also requires computation. In this way, a hard theorem or statement is demonstrated only once and then multiple instances can be proven easily.

Consider a problem to be solved in Coq, it can be stated as a property $\mathcal{P}$ at object level. In a general setting, the *ingredients* to perform a proof using reflection are the following:

- a targeted class of problems defined by a type B, for the *reified* property $\mathsf{R}\ (\mathcal{P})$, where the function to reify a term is a translation from the meta-level to the object-level;
- a *boolean* decision procedure for B, say D;
- an interpretation function $\mathsf{I} : \mathsf{B} \to \mathsf{Prop}$
- and a *soundness proof* of the decision procedure.

The theorem stating the soundness of D, let name it sound, has the following type:

$$\forall x : \mathsf{B},\ \mathsf{D}\, x = \mathsf{true} \to \mathsf{I}\, x$$

And a proof of sound applied to a specific instance $\mathsf{b} = \mathsf{R}\ (\mathcal{P})$ results in a *proof-term* for $\mathsf{I}\,\mathsf{b}$ with the form:

$$\mathsf{sound}\ \ \mathsf{b}\ \ (\mathsf{refl\_eq}\,(\mathsf{D}\,\mathsf{b}))$$

The reflection technique guarantees that the proof-term above has type $\mathsf{I}\,\mathsf{b}$ only if $\mathsf{D}\,\mathsf{b}$ is *convertible* to true. In this way, the proof of property $\mathcal{P}$ that we attempt to construct is achieved through the proof of $\mathsf{I}\,\mathsf{R}\,(\mathcal{P})$.

Beside the advantage of performing computations at type level, we have to stress two facts related to decision procedures in this approach:

- for writing the decision procedure D, the programmer is restricted to only use total functions since COQ expects that each procedure in it terminates
- and the proof of soundness of D could require extra work that may be of greater difficulty than the proof of the original goal.

As a result, the implementation of decision procedures is sometimes over simplified to shorten the proof of soundness, which may lead to inefficiencies.

Nevertheless, there is a powerful extension for proof by reflection, SSREFLECT extension, which is actively used in long mathematical proofs, for instance the computer-checked proofs of the 4-colour theorem and the Odd Order theorem.

Small Scale Reflection extension for COQ

The Mathematical Components project leaded Georges Gonthier[8] aims to show the modular formalization of mathematical theories under a computational environment. The project contributes to COQ with the small scale reflection extension or SSREFLECT [42], a set of extensions for a pervasive use of computations in formal proofs.

Its main objective is to provide a methodology to prove complex theorems, where there is an extensive use of computations with symbolic representations. Since the COQ system is interactive, all the functionalities provided by the SSREFLECT extension stay always guided by the user through a strong framework for proof management.

Among the functionalities, the most prominent ones are: an extension of the proof scripting language, a support to perform forward steps in proofs and the improvement of some basic tactics of COQ. All this changes are meant to enhance the experience while doing formal proofs by linking a logical and a symbolic approaches.

However it demands an expert user level of COQ as the decision procedures demonstrated under this extension are hard to prove.

2.3 Certifying proof by Reflection

Another variant of the reflection technique is named *certifying proof by reflection*, which reduces the cost of development of decision procedures as the user can code and run efficient decision procedures in a *general purpose* programming language. The sophisticated decision procedure is used as an *untrusted oracle* by the theorem prover. The oracle generates a certificate which only needs to be validated [44].

In the following we describe the ingredients of this variant:

- a type B which provides a description of a class of problems in COQ;
- an *untrusted* but (hopefully) *efficient* oracle D written in a general purpose language which generates certificates;
- a corresponding interpretation function I from class B to the type Prop;

8. Official site: http://www.msr-inria.fr/projects/mathematical-components-2/

- a certificate checker, check, in Coq whose type is $\forall x : \mathsf{B},\ \mathsf{C} \rightarrow \mathsf{bool}$ where C is the type of the certificates
- and a proof of the soundness of the certificate checker.

In this variant, the type of the soundness theorem $\mathsf{sound}_{\mathsf{check}}$ is

$$\forall x : \mathsf{B},\ \forall y : \mathsf{C},\ \mathsf{check}\, x\, y = \mathsf{true} \rightarrow \mathsf{I}\, x$$

Then, for a specific instance $\mathsf{b} = \mathsf{R}(\mathcal{P})$ of type B, a proof-term for $\mathsf{I}\,\mathsf{b}$ has the form:

$$\mathsf{sound}_{\mathsf{check}}\ \mathsf{b}\ (\mathsf{D}\,\mathsf{b})\ (\mathsf{refl\_eq}\ (\mathsf{check}\,\mathsf{b}\ (\mathsf{D}\,\mathsf{b})))$$

The proof-term corresponding to demonstrate the decision procedure statement must include the certificate and its validation. Thus the corresponding proof-term becomes very heavy. Nevertheless, the definition of the certificate checker is simpler than the decision procedure.

We can notice some features of the certifying proof by reflection:

- the user can implement an efficient oracle which is no longer trusted, since the language in which the implementation is done offers big facilities but is unreliable
- and the decision procedure always returns a certificate that must be checked by the certificate checker which is easier to prove than the decision procedure.

Despite the effort invested to develop an efficient decision procedure as an oracle, its implementation has only weak guarantees, *i.e.* each time we want to use a certificate, the checker will be called to validate it. Only if the certificate is validated, then the property holds for the considered instance. Then the proof of $\mathcal{P}$ corresponds to proof of term $\mathsf{I}\,\mathsf{b}$.

2.4 Discussion

Choosing the better solution to the equivalence problem among the approaches to reflection described above, depends on the user experience while performing proofs.

The two approaches presented in the preceding sections are illustrated in Figure 2.1. Recall that they share almost the same components and the key part in each style consists in writing decision procedures and proving their soundness.

$$\begin{tikzcd}
\mathsf{R}(\mathcal{P}) : \mathsf{B} \arrow[r, "\mathsf{D}", "\textit{in}\ \textsc{Coq}"'] & \mathsf{D}(\mathsf{R}(\mathcal{P})) : \mathsf{bool} \arrow[d, "\mathsf{sound}"] & \mathsf{R}(\mathcal{P}) : \mathsf{B} \arrow[r, Rightarrow, "\mathsf{D}", "\textit{in}\ \textsc{OCaml}"'] & \mathsf{D}(\mathsf{R}(\mathcal{P})) : \mathsf{C} \arrow[d, "\mathsf{sound}_{\mathsf{check}}.\mathsf{check}"] \\
\dfrac{\mathcal{P} : \mathsf{Prop}}{? : \mathcal{P}} \arrow[u, "\mathsf{R}"] \arrow[r, phantom, "\equiv"] & \dfrac{\mathsf{I}\,\mathsf{R}(\mathcal{P}) : \mathsf{Prop}}{\Delta : \mathcal{P}}\ {\scriptstyle \mathrm{CONV}} & \dfrac{\mathcal{P} : \mathsf{Prop}}{? : \mathcal{P}} \arrow[u, "\mathsf{R}"] \arrow[r, phantom, "\equiv"] & \dfrac{\mathsf{I}\,\mathsf{R}(\mathcal{P}) : \mathsf{Prop}}{\Delta : \mathcal{P}}\ {\scriptstyle \mathrm{CONV}}
\end{tikzcd}$$

Figure 2.1 – Two styles of Proof by reflection: original and certified.

The main objective is to find a proof for $\mathcal{P}$ and after the reflection technique, the proof Δ obtained from conversion of term $\mathsf{I}\,\mathsf{R}(\mathcal{P})$ gives the proof.

In the original style of proof by reflection, the language does not have native imperative features, like effects, and usually is a total programming language. Therefore, the decision procedure of the congruence problem and its proof is distant from the solution of the union-find algorithm but this contrasts with the clarity of the statement and its (short) proof performed by reduction. The final proof-term is a proof of equality, which is small and its type-checking is just a convertibility check.

The following is the instance of the above diagram, for the case of proof by computation of our decision problem of equivalences, where τ is the type of the elements in the equivalences, the data-type for pairs represent an equivalence and a list of pairs represents the set of equivalences $\mathcal{S}$:

- the class of problems is described by the type list $(\tau \times \tau) \times (\tau \times \tau)$
- the interpretation function embeds the above type into the meta-level $\mathsf{I}\,\langle \mathcal{S},\, \langle \mathsf{i}, \mathsf{j} \rangle \rangle : \mathsf{Prop}$
- applying the decision procedure gives a value of boolean type $\mathsf{D}\,\langle \mathcal{S},\, \langle \mathsf{i}, \mathsf{j} \rangle \rangle : \mathsf{bool}$
- and the theorem which states the soundness of the decision procedure is $\mathsf{sound} : \forall \mathsf{x} : \mathsf{B}, \mathsf{D}\,\mathsf{x} = \mathsf{true} \rightarrow \mathsf{I}\,\mathsf{x}$.

Then, if we show that $\mathsf{D}\,\langle \mathcal{S},\, \langle \mathsf{i}, \mathsf{j} \rangle \rangle = \mathsf{true}$ then we can conclude that $s_{\mathsf{i}} \sim s_{\mathsf{j}}$.

In a certifying style, the decision procedure is written in a general purpose language, typically an effectful language, and is mostly used as an untrusted but sophisticated oracle by the theorem prover. The final proof-term has to embed the certificate check and therefore it cannot be small as in the proof by computation style, moreover there must be a dedicated checker for the certificate.

In our example, the equivalence decision procedure is implemented under the certifying approach as follows: the two types B and I are the same to the ones in the proof by computation approach, then:

- the untrusted oracle is the decision procedure which returns certificates: $\mathsf{D} : \mathsf{B} \rightarrow \mathsf{C}$
- the theorem to check certificates is: $\mathsf{check} : \forall x : \mathsf{B},\ \mathsf{C} \rightarrow \mathsf{bool}$
- and the soundness statement is: $\mathsf{sound}_{\mathsf{check}} : \forall x : \mathsf{B},\ y : \mathsf{C},\ \mathsf{check}\, x\, y = \mathsf{true} \rightarrow \mathsf{I}\, x$.

In this setting, the programmer has more freedom to give an efficient implementation of the decision procedure.

2.5 Towards another approach to proof-by-reflection

The above analysis of three approaches, seeking an optimal solution for our problem, reveals their advantages. The first, an imperative approach, is an efficient solution in a powerful and enriched language. The second, a functional with an imperative flavour, proposes a solution closer to the previous one in an environment where the verification of the specification is reliable. The third one offers the correctness of the program implicitly in a theorem prover.

Our aim is to exploit the benefits of certified programming to have correct programs by construction. The functional and imperative solutions are used in the two approaches to reflection and exploited their benefits. We can think in carry further their use. Then, in order to improve the use of theorem provers, consider a monadic extension of the Type Theory (on which is based the theorem prover), to add the imperative facilities as offered the monadic approach in functional programming. This will offer the advantages mentioned before: the reflection technique ensures shorter proofs and the monads will allow the user to develop efficient decision procedures, all in the same certified paradigm.

However, a theorem prover like Coq imposes an environment where each program must terminate. It is possible to *describe* terms that diverge but it is not permitted to evaluate them. The monadic extension needs to ensure that each monadic statement written in the enriched language is total, this could be guarantee by a certificate of termination. But which is the best oracle to deliver certificates than a correct program?

The proposal we have in mind is to use a decision procedure as its own oracle. A procedure is constructed and verified in a safe and efficient language (the monadic extension) whereas is not evaluated, it will be *simulated* by means of a guided evaluation (to ensure its termination). The guided evaluation is performed by using a certificate which is the result of a 'real' execution in an effectful language.

We propose a new approach to reflection as just described [30]: the idea is to use *an (untrusted) compiled version of a monadic decision procedure written in Type Theory within an effectful language, as an efficient oracle for itself.*

The next chapter explains the proposal of *reflection by simulation*, a novel lightweight style of proof by reflection together with a formalisation of the requirements for a monadic extension of a type theory which is the first contribution of this thesis.

Chapter 3

Reflection by simulation

This chapter presents the formalisation of the principle of *a posteriori* simulation in the simply typed λ-calculus, in order to study the correctness of this new approach and the characteristics of monads to be simulable.

The decision procedures involved in this novel style are written in a total language based on Type Theory, which is extended with monads as we have suggested earlier. The monadic extension is inspired in the approach to add effects in a purely functional framework, as commonly found in HASKELL programs. It also uses oracle certificates to ensure the efficiency of decision procedures. In this way, programmers have a set of effects at their hands (references, exceptions, non-termination) bringing the amenities of a general purpose programming language, together with dependent types to enforce (partial) correctness.

The choice of a dependent Type Theory framework, in the final implementation, imposes some constraints. The strict and total environment of COQ does not allow the user to write and execute a total function which at the end, is a value encapsulated by a monad. Specifically, there is no total function of type $\mathsf{M}\,\tau \to \tau$ for a given arbitrary monad $\mathsf{M}\,\tau$. This restriction is minimised by means of what we characterise as a *simulable monad*, that will be presented and developed later on.

***A posteriori* simulation** The evaluation of decision procedures, in *monadic style*, is designed to be executed or simulated with the help of a certificate that we call a *prophecy* and depends on the simulable monad.

After the decision procedure is programmed, it is *compiled* into an impure programming language (OCAML) with an efficient computational model which performs all the effectful computations. The extraction procedure in COQ plays the role of compilation and is possible to be adjusted as needed [63]. The compilation maps the monad operators to effectful terms and it also *instruments the code* to compute a small piece of information that will serve as prophecy to efficiently simulate a converging reduction.

Finally, a relation of *a posteriori* simulation stands between the compiled monadic decision procedure and the original monadic procedure through the collected information as a prophecy. The lightweight approach is depicted in Figure 3.1 for a decision procedure $\mathsf{D}\,\mathsf{b}$.

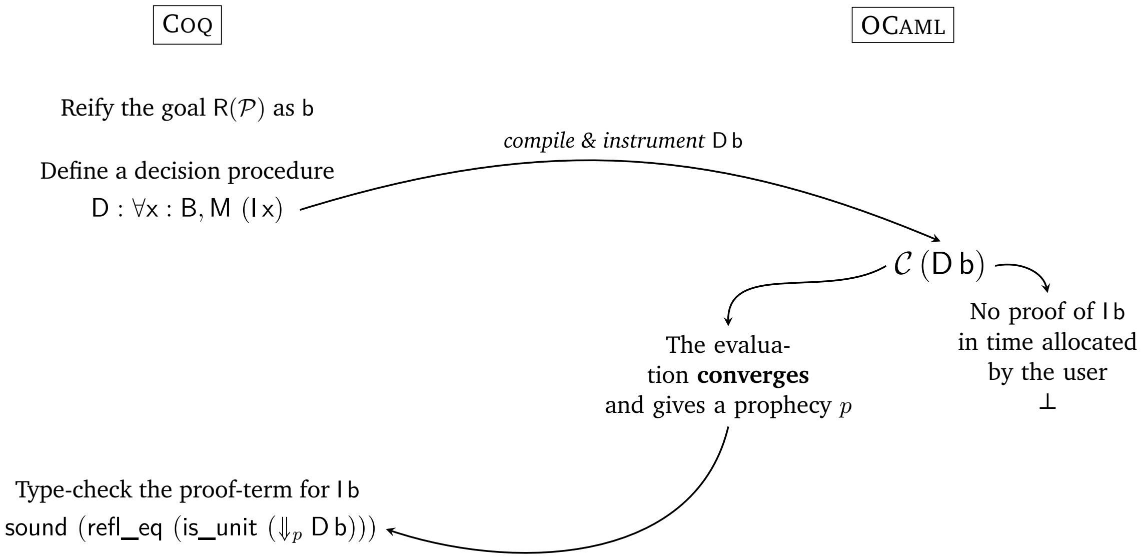
Coq
OCaml
Reify the goal $\mathsf{R}(\mathcal{P})$ as b
Define a decision procedure
$\mathsf{D} : \forall \mathsf{x} : \mathsf{B}, \mathsf{M}\ (\mathsf{I}\,\mathsf{x})$
compile & instrument $\mathsf{D}\,\mathsf{b}$
$\mathcal{C}\,(\mathsf{D}\,\mathsf{b})$
No proof of $\mathsf{I}\,\mathsf{b}$ in time allocated by the user $\perp$
The evaluation **converges** and gives a prophecy p
Type-check the proof-term for $\mathsf{I}\,\mathsf{b}$
$\mathsf{sound}\ (\mathsf{refl\_eq}\ (\mathsf{is\_unit}\ (\Downarrow_p \mathsf{D}\,\mathsf{b})))$

Figure 3.1 – Lightweight proof by reflection

As the reflection technique requires, the property $\mathcal{P}$ is reified into a term b of type B. Then, a decision procedure D for terms of type B is given and its return value has a monadic type for interpreted terms $\mathsf{I}\,\mathsf{x}$. The compilation of the instance of the decision procedure $\mathsf{D}\,\mathsf{b}$, is instrumented to collect information during its execution in the effectful language. This resulting piece of information will be the prophecy to simulate back the decision procedure in the Coq proof assistant.

Once inside the effectful language, the compiled term is executed: either the execution takes a long time or it does not finish, or the evaluation converges. In the first case, the user can stop the execution and hence no prophecy is generated, meaning that the simulation process will not be finished. The second case, where the evaluation converges, the compiled term gives a prophecy p which is used as a certificate for the guided evaluation back in the theorem prover. The proof of the soundness theorem is the proof of the property $\mathcal{P}$: $\mathsf{sound}\ (\mathsf{refl\_eq}\ (\mathsf{is\_unit}\ (\Downarrow_p \mathsf{D}\,\mathsf{b})))$.

We are interested in the formalisation of the above process and moreover the requirements a monad needs to achieve a simulation. The formalisation proposed uses two languages, one to represent the type theory with a monadic extension and one for the impure language where the prophecies are generated. We make this formalisation over the simply typed λ-calculus but we think that the results presented are extensible to full Type Theory and OCaml.

On the one hand, we define λ_{M} a purely functional and strongly normalizing programming language parametrized by a monad M. On the other hand, we define $\lambda_{v,\perp}$ an impure functional and non-terminating programming language. The parameter monad M of λ_{M} is abstractly specified by a set of requirements. Accordingly, $\lambda_{v,\perp}$ offers impure operators that match the effectful primitives of monad M. Through this chapter, we develop both languages offering the constructions needed to formalise the lightweight style.

Roughly speaking, given a computation[1] t of a monadic type $\mathsf{M}\,\tau$, we determine on which conditions there exists some information p such that the evaluation of the computation using such information can witness an inhabitant of type τ. This forces a close relation between the two languages, the soundness of the simulation shows this relation: if a compiled computation $\mathcal{C}(t)$ converges to a value while *recording* some information p in $\lambda_{v,\perp}$, then the same evaluation can be simulated *a posteriori* in λ_{M} using that piece of information as a prophecy. This prophecy *completes* the computation to get a reduced computation $\Downarrow_p t$ convertible to a monadic value $\mathsf{unit}\ t'$, where t' is the inhabitant of the proof we are looking for.

The two following sections describe the languages λ_{M} and $\lambda_{v,\perp}$. Then we link both languages by the principal theorem of soundness of the *a posteriori* simulation. Finally we discuss some examples of simulable monads and in the chapter conclusions we give some closing remarks and comment the available plug-in for Coq, Cybele[2] for a lightweight proof by reflection.

3.1 λ_{M} a purely functional monadic language

The language λ_{M} is an extension of the simply typed λ-calculus *à la Curry* with evaluation contexts to characterise a call-by-value small-step semantics. The language, its dynamic semantics and the typing rules are depicted in Figure 3.2.

The terms of language λ_{M} include variables, lambda abstractions, term applications and a set of constant symbols c. Constants include the usual monadic combinators for effects [111]: unit lifts a term of type τ as a computation of type $\mathsf{M}\,\tau$, and bind composes two given computations.

An additional constant is $\Downarrow$ whose role is to perform *a posteriori* simulation using a prophecy value p of type P. When applied, instead of writing $\Downarrow t\,p$, we write $\Downarrow_p t$[3] which is read as 'the computation of t reduced using the prophecy p'.

Prophecies are distinguished constants that will guide and complete the reduction of a monadic term. The characterisation of prophecies will be explained later by a set of requirements shared between languages λ_{M} and $\lambda_{v,\perp}$. We emphasise that each prophecy is produced in language $\lambda_{v,\perp}$ by a compiled computation, this will be established and formalised in the next section where the compilation of terms of the monadic language is presented.

Constants are kept abstract by regrouping them into the syntactic category ∇. They include constants of basic types and other specific effectful primitives $\mathbf{c}$, for example recursion or state operators.

Reduction of terms is directed by *evaluation contexts* which are described by the syntactic category $\mathcal{E}$. They are just terms of the language with one 'hole' (represented by $[\,]$), containing an emphasised *redex* which is the focus of the call-by-value reduction.

1. The decision procedures are programs or monadic terms, from now on we call them computations.
2. `http://cybele.gforge.inria.fr`
3. This convention is used along the chapter.

$\boxed{\text{Syntax}}$

$$
\begin{array}{rcl}
t,\ r,\ s & ::= & x \mid \lambda x.\, s \mid r\, s \mid c \\
c & ::= & \mathsf{unit} \mid \mathsf{bind} \mid \Downarrow \mid \nabla \\
u,\ v,\ w & ::= & x \mid \lambda x.\, s \mid \mathsf{unit}\ r \\
\tau,\ \sigma & ::= & \sigma \to \tau \mid \mathsf{M}\,\tau \mid \mathsf{P}
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{E} & ::= & [\,] \mid v\,\mathcal{E} \mid \mathcal{E}\, t \mid \mathsf{bind}\,\mathcal{E}\, t \mid \Downarrow_p \mathcal{E} \\
\Gamma & ::= & \varnothing \mid \Gamma,\, x^{\tau}
\end{array}
$$

$\boxed{\text{Dynamic Semantics}}$

$$
\begin{array}{l}\beta\text{-}\textsc{Red} \\ (\lambda x.\, s)\ v \longrightarrow s\,[x\ :=\ v]\end{array}
\qquad
\begin{array}{l}\textsc{Comp-Red} \\ \mathsf{bind}\ (\mathsf{unit}\ r)\ s \longrightarrow s\, r\end{array}
\qquad
\textsc{Red}\ \frac{t \longrightarrow t'}{\mathcal{E}[t] \longrightarrow \mathcal{E}[t']}
$$

$\boxed{\text{Static Semantics}}$

$$
\frac{x^{\tau} \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var}
\qquad
\frac{\Gamma,\, x^{\sigma} \vdash s : \tau}{\Gamma \vdash \lambda x.\, s : \sigma \to \tau}\ \textsc{Abs}
\qquad
\frac{\Gamma \vdash r : \sigma \to \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash r\, s : \tau}\ \textsc{App}
$$

$$
\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{unit}\ t : \mathsf{M}\,\tau}\ \textsc{Unit}
\qquad
\frac{\Gamma \vdash r : \mathsf{M}\,\sigma \qquad \Gamma \vdash s : \sigma \to \mathsf{M}\,\tau}{\Gamma \vdash \mathsf{bind}\, r\, s : \mathsf{M}\,\tau}\ \textsc{Bind}
$$

$$
\frac{\Gamma \vdash t : \mathsf{M}\,\tau \qquad \Gamma \vdash p : \mathsf{P}}{\Gamma \vdash \Downarrow_p t : \mathsf{M}\,\tau}\ \textsc{Eval}
$$

Figure 3.2 – The language λ_{M}

In order to accomplish a call-by-value reduction strategy we also distinguish the syntactic category of values which include variables, λ-abstractions and terms whose head symbol is the unit constructor. The reduction process is performed in the traditional way by a step-by-step rewriting of the emphasised redex *via* some axioms and the context rule RED.

There are three reduction axioms: the usual β-reduction for lambda terms (β-RED), a reduction of the composition of computations (COMP-RED) and a set of δ-reductions for unfolding constant definitions (δ_{c}-RED) which is left abstract until basic types and a monad are fixed.

Each constant **c** in ∇ is a term with a non-negative arity, in particular when a fully applied constant is reduced, then the corresponding reduction axiom will be triggered. For example, numerical constants and primitive operations, like addition, are basic constants and only the full-applied operations can be reduced.

The intended reduction behaviour of constant $\Downarrow$ is similar to the monadic primitive run. The action of the later is to take a monadic term, *unwrap* the term from the monad and reduce it retrieving a result of a possibly non-monadic type. Here, the simulation constant is used to represent the fact that a monadic term must be reduced using a prophecy returning a computation.

Routine operations over terms that accompany any lambda calculus (see the preliminaries) are also assumed for language λ_{M}, such as the set of free and bounded variables, α-conversion, substitution, etc. The substitution is defined as a mapping between variables and closed terms.

The full reduction of terms is done by the reflexive and transitive closure of the small-step semantics. We are interested in reasoning on $\beta\delta$-convertibility between terms, then we write $t_1 = t_2$ when t_1 is $\beta\delta$-convertible into t_2, that is when both terms fully-reduce to syntactically equal terms.

With respect to the static semantics, types in λ_{M} include functional types and type constructor applications, which are assumed to be well-formed. M and P are the type constructors for monad and prophecy, respectively.

The type assignment uses *typing environments* or contexts, which are sets of variables including the free variables of the term to be typed. Typing environments Γ, are defined inductively by the empty context and the binding operation to add a variable with its type to a given context. We say that a variable is in the *last position* in a context when that variable appears in the rightmost place in it.

The judgements for type assignment are the last set of rules in Figure 3.2 relating contexts, terms and types: lambda terms, abstractions and applications, have the traditional rules inherited from the simply typed λ-calculus. For the monadic combinators, the typing rules are as expected: the UNIT and BIND rules assign a monadic type to the corresponding terms unit and bind. Since we are reasoning under an abstract monad M we left undefined the typing rules for the monad primitives in ∇. Finally, the applied constant $\Downarrow$ has a monadic type which is captured by rule EVAL. As we mentioned before, only the full application of constants to terms can be reduced and the typing rules described enforce this[4].

In the following we state the requirements for prophecies and monads needed to support our *a posteriori* simulation approach to reflection. The remaining of this section is dedicated to some definitions and state the lemmas and theorems to prove properties about the language.

3.1.1 Simulation

Up to now, we stated the basis of a simple language with a monadic extension where computations can be expressed. We set the features addressed to *simulate* computations in the system λ_{M} through a set of requirements where the standard notion of monad is extended with a mechanism of simulation directed by a prophecy.

Definition 3.1 (Simulable monad)
A type constructor M *is a simulable monad if it is equipped with constants* unit, bind, $\Downarrow$ *and an associated type for prophecies* P*, such that the Requirements 0, 1, 2, 3 and 4 are fulfilled by well-typed closed terms.*

4. The reader can always deduce the type of a non-fully applied term by the rules given in Figure 3.2, since the judgement hypotheses appear in the same order as the sub-terms are applied. For example, consider $\Downarrow$ whose type is $\mathsf{M}\,\tau \to \mathsf{P} \to \mathsf{M}\,\tau$.

Requirement 0 (About prophecies)
We require the existence of a total order $\leqslant$ over values of type P *with a minimal element denoted as* $\perp$.

Definition 3.2 (Convergence)
Consider the minimal element of any collection of prophecies, we write $\star t$ *for* $\Downarrow_\perp$ $(\mathsf{unit}\ t)$ *and we say that a computation has* converged *if there exist a prophecy* p *and a term* r *such that* $\Downarrow_p t$ *is convertible to* $\star r$*. If term* t *is closed, then the reduced term* r *is a value.*

Requirement 1 (Standard monadic laws)

$$\begin{array}{rcl}
\mathsf{bind}\,(\mathsf{unit}\ t)\,f & = & f\,t \\
\mathsf{bind}\,t\,(\lambda x.\,\mathsf{unit}\ x) & = & t \\
\mathsf{bind}\,(\mathsf{bind}\,t_1\,t_2)\,t_3 & = & \mathsf{bind}\,t_1\,(\lambda x.\,\mathsf{bind}\,(t_2\,x)\,t_3)
\end{array}$$

Requirement 2 (Evaluation)

$$\begin{array}{c}
\forall\ t,p,\ \ \Downarrow_p \mathsf{unit}\ t = \mathsf{unit}\ t. \\
\forall\ t,s,p_1,p_2,\ \ p_1 \leqslant p_2\ \ \textit{and}\ \ \Downarrow_{p_1} t = \star s\ \ \textit{implies that}\ \ \Downarrow_{p_2} t = \star s. \\
\forall\ p,\ \ \Downarrow_p \mathsf{bind}\,t_1\,t_2 = \Downarrow_p \mathsf{bind}\,(\Downarrow_p t_1)\,t_2
\end{array}$$

Definition 3.1 states five requirements for a monad to be simulable. The requirement zero ensures the existence of an order over prophecies while the first and second requirements are the conditions to conduct the simulable reduction of computations in system λ_{M}. Requirements 3 and 4 are the conditions for a successful information tracking from the compiled version of a computation in order to simulate it back. They are described in Section 3.2, which presents the *oracle* language, a lambda calculus with an operational semantics devoted to collect information to be used as prophecy.

3.1.2 Properties

The language exposed so far has the properties of the simply typed λ-calculus: uniqueness of types, closed and well-typed terms are either values or can make a reduction step (progress property), dynamic semantics preserves types (type preservation), and any well-typed term is normalizing, that is, the reduction yields a normal form. We give the proof of these properties as well as some auxiliary definitions and additional properties, starting by some properties related to type assignment, following a well known presentation of type systems (Pierce et al. [92]).

Lemma 3.1 (Permutation of contexts)
If $\Gamma \vdash t : \tau$ *and* Γ' *is a permutation of* Γ *then* $\Gamma' \vdash t : \tau$.

Proof. Induction on the type derivation $\Gamma \vdash t : \tau$ **(1)**.

- Case $\Gamma \vdash x : \tau$.
  Consider an environment Γ where variable x belongs to. Then, any permutation Γ' of Γ has variable x and therefore $\Gamma' \vdash x : \tau$ holds.

The induction hypotheses are the properties instantiated for the corresponding premises of the typing rule (1).

- Case $\Gamma \vdash \lambda x.\, s : \sigma \to \tau$.
  The premise in the typing judgement is: $\Gamma,\, x^{\sigma} \vdash s : \tau$ **(2)**.
  The induction hypothesis (2) gives a typing assignment for the sub-term s under a permutation of context $\Gamma,\, x^{\sigma}$ which we call $\Gamma'' = \Gamma',\, x^{\sigma}$, that is a permutation in which the abstracted variable is in the last position of the environment and Γ' is a permutation of Γ. Then, we can apply rule LAM with the above information and conclude that term $\lambda x.\, s$ has type $\sigma \to \tau$ using context Γ''.

- Case $\Gamma \vdash r\, s : \tau$.
  In this case, the induction hypotheses are the statements of permutation and weakening for $\Gamma \vdash r : \sigma \to \tau$ **(3)** and $\Gamma \vdash s : \sigma$ **(4)**.
  Take the same permutation of environment Γ, say Γ' for premises (3) and (4). Then by rule APP we can assign the type τ to the term application $r\, s$ under Γ'.

- Case $\Gamma \vdash \mathsf{unit}\ t' : \mathsf{M}\,\tau$.
  The induction hypothesis considers a permutation of context Γ such that $\Gamma' \vdash t' : \tau$. Then, by rule UNIT, it is possible to assign type $\mathsf{M}\,\tau$ to term $\mathsf{unit}\ t'$ using the above judgement.

- Case $\Gamma \vdash \mathsf{bind}\, r\, s : \mathsf{M}\,\tau$.
  This case takes two induction hypotheses, one for $\Gamma \vdash r : \mathsf{M}\,\sigma$ and the second one for $\Gamma \vdash s : \sigma \to \mathsf{M}\,\tau$.
  Consider the same permutation in the instantiations of the induction hypotheses. Then, we can use the typing rule BIND to give type $\mathsf{M}\,\tau$ to t.

- Case $\Gamma \vdash \Downarrow_p t' : \mathsf{M}\,\tau$.
  From judgement $\Gamma \vdash t' : \mathsf{M}\,\tau$, the induction hypothesis states that $\Gamma,\, y^{\tau'} \vdash t' : \mathsf{M}\,\tau$.
  The induction hypothesis ensures the typing assignment $\mathsf{M}\,\tau$ for term t' under a permutation Γ' of context Γ. Then, taking the same prophecy p we can assign the monadic type $\mathsf{M}\,\tau$ to $\Downarrow_p t'$ under Γ'. $\square$

Lemma 3.2 (Weakening of contexts)
If $\Gamma \vdash t : \tau$ *and given a variable* y *of type* τ' *such that* $y \notin \Gamma$*, then* $\Gamma,\, y^{\tau'} \vdash t : \tau$.

Proof. The proof is performed by induction on the type derivation $\Gamma \vdash t : \tau$ **(1)**.

- Case $\Gamma \vdash x : \tau$.
  Take any environment Γ in which the variable x of type τ belongs to. Adding a fresh variable y of type τ' to the typing context does not modify the type of variable x following rule VAR.

The induction hypotheses are the properties instantiated for the corresponding premises of the typing rule (1).

- Case $\Gamma \vdash \lambda x.\, s : \sigma \to \tau$.
  The premise in the typing judgement is: $\Gamma,\, x^{\sigma} \vdash s : \tau$ **(2)**. We want to prove that, under an extension of Γ, we keep the same arrow type for $\lambda x.\, s$. From the induction

hypothesis (2), we can extend the context into $\Gamma, x^{\sigma}, y^{\sigma'}$ (**3**) and preserve the type of s. Take a permutation of the above context (3), where in its last position appears the variable x. Then we can use rule LAM and conclude.

- Case $\Gamma \vdash r\, s : \tau$.
  In this case, the induction hypotheses are the statements of permutation and weakening for $\Gamma \vdash r : \sigma \rightarrow \tau$ (**4**) and $\Gamma \vdash s : \sigma$ (**5**). The extension of Γ with y of type τ' in judgements (4) and (5) generate judgements: $\Gamma, y^{\tau'} \vdash r : \sigma \rightarrow \tau$ and $\Gamma, y^{\tau'} \vdash s : \sigma$. They can be used as premises in rule APP to conclude that $\Gamma, y^{\tau'} \vdash r\, s : \tau$.
- Case $\Gamma \vdash \mathsf{unit}\ t' : \mathsf{M}\,\tau$.
  The induction hypothesis $\Gamma, y^{\tau'} \vdash t' : \tau$ as premise of rule UNIT allows to prove that term $\mathsf{unit}\ t'$ has the same type under the above context extension.
- Case $\Gamma \vdash \mathsf{bind}\, r\, s : \mathsf{M}\,\tau$.
  This case takes two induction hypotheses, one for $\Gamma \vdash r : \mathsf{M}\,\sigma$ and the second one for $\Gamma \vdash s : \sigma \rightarrow \mathsf{M}\,\tau$
  Take the extension of context Γ used in the induction hypothesis $\Gamma, y^{\tau'}$. By applying the rule BIND to judgements $\Gamma, y^{\tau'} \vdash r : \mathsf{M}\,\sigma$ and $\Gamma, y^{\tau'} \vdash s : \sigma \rightarrow \mathsf{M}\,\tau$ we can assign the type $\mathsf{M}\,\tau$ to term $\mathsf{bind}\, r\, s$.
- Case $\Gamma \vdash \Downarrow_p t' : \mathsf{M}\,\tau$.
  From judgement $\Gamma \vdash t' : \mathsf{M}\,\tau$, the induction hypothesis states that $\Gamma, y^{\tau'} \vdash t' : \mathsf{M}\,\tau$. Then by applying rule EVAL to the last judgement, we can conclude that the type of $\Downarrow_p t'$ is $\mathsf{M}\,\tau$ under the context extension. □

When a term with a specific form has a given type, we can deduce the form of the type from the typing rules in Figure 3.2, this is the inversion lemma:

Lemma 3.3 (Inversion of typing in λ_{M})

- *If* $\Gamma \vdash x : \tau$ *then* $x^{\tau} \in \Gamma$.
- *If* $\Gamma \vdash \lambda x.\, s : \tau$ *then there exist* σ *and* τ' *such that* $\tau = \sigma \rightarrow \tau'$ *and* $\Gamma, x^{\sigma} \vdash s : \tau'$.
- *If* $\Gamma \vdash r\, s : \tau$ *then there exists* σ *such that* $\Gamma \vdash r : \sigma \rightarrow \tau$ *and* $\Gamma \vdash s : \sigma$.
- *If* $\Gamma \vdash \mathsf{unit}\ t : \tau$ *then there exists* τ' *such that* $\tau = \mathsf{M}\,\tau'$ *and* $\Gamma \vdash t : \tau'$.
- *If* $\Gamma \vdash \mathsf{bind}\, r\, s : \tau$ *then there exist* σ *and* τ' *such that* $\tau = \mathsf{M}\,\tau'$, $\Gamma \vdash r : \mathsf{M}\,\sigma$ *and* $\Gamma \vdash s : \sigma \rightarrow \mathsf{M}\,\tau'$.
- *If* $\Gamma \vdash \Downarrow_p t : \tau$ *then there exist* P *and* τ' *such that* $\tau = \mathsf{M}\,\tau'$, $\Gamma \vdash p : \mathsf{P}$ *and* $\Gamma \vdash t : \mathsf{M}\,\tau'$.

Proof. The proof is direct from analysis of the last rule used in the typing derivation of the hypothesis. In each case there is always a single choice leading to a unique shape for type τ. □

Using the typing rules of λ_{M}, we ensure the assignment of a unique type to any term in the language.

Theorem 3.4 (Uniqueness of types)
Consider a typing context Γ *and a term* t. *If* t *is typeable under* Γ *then it has a unique type and there is one derivation to build it.*

Proof. The proof is to show that for a given term, the type derivation is unique. This is achieved because the static semantics of the system is syntax directed and at each step of the derivation there is a single choice of the rule to be used. □

In the definition of language λ_{M}, we gave a syntactic condition to distinguish values as irreducible terms. However, we can also decide the form of a closed value by knowing its type. They are the canonical forms and they are useful in later proofs.

Lemma 3.5 (Canonical forms of λ_{M})

- *If $\varnothing \vdash v : \sigma \to \tau$ then there exist x and s such that $v = \lambda x.\, s$.*
- *If $\varnothing \vdash v : \mathsf{M}\,\tau$ then there exists r such that $v = \mathsf{unit}\ r$.*

Proof. Recall that among the values defined in Figure 3.2, there are lambda abstractions and computations under the unit constructor. This minimises the potential cases of closed values to be analysed in the proof.
The first type assignment gives a function type to value v. This is possible through rule ABS and from the two available values, only the function abstraction can be used in that rule. The type assignment $\varnothing \vdash v : \mathsf{M}\,\tau$ can not be used to type an abstraction as stated in the inversion lemma 3.3. Therefore unit r for some term r is the value that fits best and the applied rule is UNIT. □

The semantics of language λ_{M} ensures the progress of any chain of reduction, and therefore, for closed and well-typed terms we avoid stuck reductions.

Theorem 3.6 (Progress)
If $\varnothing \vdash t : \tau$ then either t is a value or there exists t' such that $t \longrightarrow t'$.

Proof. Induction on the derivation of $\varnothing \vdash t : \tau$ and case analysis of the evaluation contexts used in reduction.

- Case $\varnothing \vdash \lambda x.\, s : \tau$.
  The theorem holds for any lambda abstraction since it is considered a value.
- Case $\varnothing \vdash r\, s : \tau$.
  The term application is well typed and by inversion 3.3 we know that the sub-terms r and s have type $\sigma \to \tau$ and σ respectively. Consider the induction hypothesis instantiated for them, we analyse the following cases:
  - If r is a value v of function type then it is a canonical form, *i.e.* an abstraction as stated in Lemma 3.5. Then, we proceed to inspect term s: if it is a value then we can apply the axiom β-RED to reduce and therefore there exists a term t'.
    If it is not a value, then by induction hypothesis, there exists a reduction $s \longrightarrow s'$ which can be applied as premise in rule RED for context $v\,\mathcal{E}$.
  - If r is not a value, we can ensure a reduction of term $r\,s$ by means of the reduction of r in rule RED over context $\mathcal{E}\,s$.
- Case $\varnothing \vdash \mathsf{unit}\ r : \tau$.
  A term of the form unit r is already a value.

- Case $\varnothing \vdash \mathsf{bind}\, r\, s : \tau$.
  Consider the two instantiations of the induction hypothesis for r and s of type $\mathsf{M}\,\sigma$ and $\sigma \rightarrow \mathsf{M}\,\sigma'$ respectively.
  - If r is a value, then by Lemma 3.5 it is a canonical form: $\mathsf{unit}\ r'$. Then, we can apply the reduction axiom Comp-Red to reduce computations.
  - If r is not a value, then we can perform a reduction on it and therefore, exists term t' which is the reduction of context $\mathsf{bind}\, \mathcal{E}\, s$.
- Case $\varnothing \vdash \Downarrow_p t' : \tau$.
  The induction hypothesis for this case is the statement for term t' which by inversion has type $\mathsf{M}\,\tau'$ for some τ'.
  - If t' is a value then, following Lemma 3.5, it is a canonical form say $\mathsf{unit}\ r$. The proof is to show that $\Downarrow_p \mathsf{unit}\ r$ **(1)** is a value or it can make progress. By Requirement 2, we ensure the progress of term (1) (through the corresponding δ-rule of $\Downarrow$).
  - If t' is not a value then a reduction can be done inside the context $\Downarrow_p \mathcal{E}$. □

Another property relating the static and dynamic semantics, is the theorem which shows that after a reduction step of a well-typed term, the type of the reduced term remains the same. To achieve this proof, we must ensure the preservation of the type after a substitution which will be useful in the case of type preservation of applications, we prove this property and then the theorem for type preservation.

Lemma 3.7 (Type preservation under substitution)
If $\Gamma, x^\sigma \vdash t : \tau$ *and* $\Gamma \vdash v : \sigma$ *then* $\Gamma \vdash t\,[x\ := \ v] : \tau$.

Proof. Suppose that $\Gamma \vdash v : \sigma$ **(1)**. The proof is carried out by induction on the derivation $\Gamma' \vdash t : \tau$ where $\Gamma' = \Gamma, x^\sigma$. We analyse the cases of the last rule used in the derivation.

- Case $\Gamma, x^\sigma \vdash y : \tau$.
  If $y = x$ then $\tau = \sigma$ and the variable preserves the type after the substitution. If the variables are different, then the substitution does not affect variable y which already belongs to Γ and keeps its type τ.
- Case $\Gamma, x^\sigma \vdash \lambda y.\, s : \tau' \rightarrow \tau$.
  Choose the variables x and y to be different. By inversion Lemma 3.3, we ensure that $\Gamma'' \vdash s : \tau$ where $\Gamma'' = \Gamma, x^\sigma, y^{\tau'}$. Following Lemma 3.1, take a permutation of environment Γ'' where the right-most variable is x. Then the induction hypothesis preserves the type of s after applying the substitution: $\Gamma, y^{\tau'} \vdash s\,[x\ := \ v] : \tau$.
  Finally we can assign type $\tau' \rightarrow \tau$ to the abstraction $\lambda y.\, s\,[x\ := \ v]$ under environment Γ using the typing rule Lam.
- Case $\Gamma, x^\sigma \vdash r\, s : \tau$.
  This case has two induction hypotheses for $\Gamma' \vdash r : \sigma \rightarrow \tau$ and $\Gamma' \vdash s : \sigma$, the terms preserve their types after the substitution. Then we can derive the type τ for the application $r\, s$ using rule App with context Γ.

- Case $\Gamma,\, x^\sigma \vdash \mathsf{unit}\ t' : \mathsf{M}\,\tau$.
  By inversion, the type of term t' is τ and by induction hypothesis, the type of $t'\,[x\ := \ v]$ is preserved. Then, using rule UNIT the last term has type $\mathsf{M}\,\tau$ and therefore the type is preserved.
- Case $\Gamma,\, x^\sigma \vdash \mathsf{bind}\, r\, s : \mathsf{M}\,\tau$.
  The inversion Lemma 3.3 gives the types for terms r and s, $\mathsf{M}\,\sigma$ and $\sigma \to \mathsf{M}\,\tau$ respectively using the type context Γ'. Then, by induction hypothesis both terms preserve their type after substitution. To prove that term $(\mathsf{bind}\, r\, s)\ [x\ := \ v] = \mathsf{bind}\, r\,[x\ := \ v]\, s\,[x\ := \ v]$ has also type $\mathsf{M}\,\tau$, it is sufficient to apply the typing rule BIND with the induction hypotheses as premises.
- Case $\Gamma,\, x^\sigma \vdash \Downarrow_p t' : \mathsf{M}\,\tau$.
  The induction hypothesis for this case is the type preservation of the corresponding type $\mathsf{M}\,\tau$ of term t' after substitution $[x\ := \ v]$. Then, the application of rule EVAL with the induction hypotheses $\Gamma,\, x^\sigma \vdash t' : \mathsf{M}\,\tau$ and $\Gamma,\, x^\sigma \vdash p : \mathsf{P}$ shows the preservation of type $\mathsf{M}\,\tau$ after the substitution $(\Downarrow_p t')\ [x\ := \ v] = \Downarrow_p t'\,[x\ := \ v]$. □

Theorem 3.8 (Type preservation)
If $\Gamma \vdash t : \tau$ *and* $t \longrightarrow t'$ *then* $\Gamma \vdash t' : \tau$.

Proof. Induction on the last rule used in derivation $\Gamma \vdash t : \tau$ with analysis of reduction of term t.

- Cases $\Gamma \vdash x : \tau$, $\Gamma \vdash \lambda x.\, s : \tau$ and $\Gamma \vdash \mathsf{unit}\ r : \tau$.
  A variable, a lambda abstraction or a unit-term cannot be reduced, so we discard this cases.
- Case $\Gamma \vdash r\, s : \tau$.
  By inversion Lemma we know that there exists σ such that $\Gamma \vdash r : \sigma \to \tau$ **(1)** and $\Gamma \vdash s : \sigma$ **(2)**. We proceed by analysis of the reduction of the application.
  - β-RED where $(\lambda x.\, r')\ v \longrightarrow r'\,[x\ := \ v]$.
    In order to assign a type to $r'\,[x\ := \ v]$ we use Lemma 3.7 with $\Gamma,\, x^\sigma \vdash r' : \tau$ obtained by inversion of (1) where $r = \lambda x.\, r'$.
  - RED where $\mathcal{E}[s] = v\, s$ and $s \longrightarrow s'$.
    By induction hypothesis, term s' preserves the type σ. Then, we use this hypothesis together with (1) in rule APP to assign type τ to term $v\, s'$.
  - RED where $\mathcal{E}[r] = r\, s$ and $r \longrightarrow r'$.
    This sub-case uses the induction hypothesis of typing of r' together with hypothesis (2) in rule APP. This gives the type τ to the application $r'\, s$.
- Case $\Gamma \vdash \mathsf{bind}\, r\, s : \tau$.
  We ensure by the inversion Lemma that there exist types σ and τ' such that $\tau = \mathsf{M}\,\tau'$, $\Gamma \vdash r : \mathsf{M}\,\sigma$ **(3)** and $\Gamma \vdash s : \sigma \to \mathsf{M}\,\tau'$ **(4)**. We analyse the ways of reducing the term $\mathsf{bind}\, r\, s$.
  - COMP-RED where bind $(\mathsf{unit}\ r')\ s \longrightarrow s\,[x\ := \ t']$.
    The terms $\mathsf{unit}\ r'$ and s are well typed and by inversion we ensure that r' and s have type σ and (4) respectively. Then we can apply Lemma 3.7 with these typing statements to ensure that the substitution $s\,[x\ := \ r']$ has type $\mathsf{M}\,\tau'$.

    - RED where $\mathcal{E}[r] = \mathsf{bind}\, r\, s$ and $r \longrightarrow r'$.
      The rule BIND assign type $\mathsf{M}\,\tau'$ to term $\mathsf{bind}\, r'\, s$ using $\Gamma \vdash r' : \mathsf{M}\,\sigma$, which holds by induction hypothesis, and (4).
- Case $\Gamma \vdash \Downarrow_p t' : \tau$.
  We know, by inversion, that $\tau = \mathsf{M}\,\tau'$ for some τ' and that $\Gamma \vdash t' : \mathsf{M}\,\tau'$ **(5)** and $\Gamma \vdash p : \mathsf{P}$ **(6)**.
    - RED where $\mathcal{E}[t'] = \Downarrow_p t'$.
      Consider the induction hypothesis for $t' \longrightarrow t''$ which preserves type $\mathsf{M}\,\tau'$. Then, to prove that $\Gamma \vdash \Downarrow_p t'' : \tau'$ we use the above hypothesis together with (6) in rule RED.
    - A δ-reduction of $\Downarrow$.
      Since the monadic type $\mathsf{M}\,\tau$ is abstract, then the typing preservation must be ensured by the definition of rule for reduction of $\Downarrow$. □

In Theorem 3.4 we showed that the typing assignment is deterministic. We can ensure the same property for the reduction relation as stated in the following lemma:

Lemma 3.9 (Determinism of dynamic semantics)
The small-step reduction $\rightarrow$ is deterministic.

Proof. The reduction strategy imposed by the dynamic semantics through the evaluation contexts (call-by-value with left to right) gives a unique derivation. □

Normalization

The normal forms are terms on which it is not possible to make progress, that is, terms that do not contain any redexes. They include the values of system λ_{M} and other irreducible terms as defined by the grammar in Figure 3.3.

$$\begin{array}{rcl} \hat{t} & ::= & \lambda x.\, s \mid \mathsf{unit}\ \hat{t} \mid m \mid \hat{t}\,\hat{t} \\ m & ::= & x \mid \mathbf{c} \mid m\,\hat{t} \mid \mathsf{bind}\, m\,\hat{t} \end{array}$$

Figure 3.3 – Normal forms of λ_{M}

In order to prove that any well-typed term has a normal form, that is for any term t there exists a normal form $\hat{t}$ such that $t \longrightarrow^* \hat{t}$, we follow the approach of Joachimski and Matthes [56] where two logical relations $\mathcal{W}^{\Gamma}_{\tau}$ and $\mathcal{WN}^{\Gamma}_{\tau}$ are defined to characterise the normal forms in λ_{M} defined above.

The relation $\mathcal{WN}^{\Gamma}_{\tau}$, defined in Figure 3.4, has four cases for terms with a specific shape which verify recursively the normal forms of their sub-terms. Another case for ground cases, characterised by relation $\mathcal{W}^{\Gamma}_{\tau}$. And a final case to make emphasis that reduction lead to normal forms.

Prophecies in λ_{M} are considered as constants and the membership of constants in relation $\mathcal{WN}$ is ensured by the cases of the ground relation $\mathcal{W}$, they are abstract and will be defined after establishing the basic types.

$$
\begin{array}{rl}
\mathcal{W}^{\Gamma}_{\tau}(x) & \textit{if } x^{\tau} \in \Gamma. \\
\mathcal{W}^{\Gamma}_{\tau}(\mathbf{c}) & \textit{if } \Gamma \vdash \mathbf{c} : \tau. \\
\mathcal{W}^{\Gamma}_{\tau}(r\, s) & \textit{if } \Gamma \vdash r\, s : \tau,\ \mathcal{W}^{\Gamma}_{\sigma\to\tau}(r) \textit{ and } \mathcal{WN}^{\Gamma}_{\sigma}(s). \\
\\
\mathcal{WN}^{\Gamma}_{\sigma\to\tau}(\lambda x.\, s) & \textit{if } \Gamma \vdash \lambda x.\, s : \sigma \to \tau \textit{ and } \mathcal{WN}^{\Gamma, x^{\sigma}}_{\tau}(s). \\
\mathcal{WN}^{\Gamma}_{\tau}(r\, s) & \textit{if } \Gamma \vdash r\, s : \tau,\ \mathcal{WN}^{\Gamma}_{\sigma\to\tau}(r) \textit{ and } \mathcal{WN}^{\Gamma}_{\sigma}(s). \\
\mathcal{WN}^{\Gamma}_{\mathsf{M}\,\tau}(\mathsf{unit}\ t) & \textit{if } \Gamma \vdash \mathsf{unit}\ t : \mathsf{M}\,\tau \textit{ and } \mathcal{WN}^{\Gamma}_{\tau}(t). \\
\mathcal{WN}^{\Gamma}_{\mathsf{M}\,\tau}(\mathsf{bind}\, r\, s) & \textit{if } \Gamma \vdash \mathsf{bind}\, r\, s : \mathsf{M}\,\tau,\ \mathcal{WN}^{\Gamma}_{\mathsf{M}\,\sigma}(r) \textit{ and } \mathcal{WN}^{\Gamma}_{\sigma\to\mathsf{M}\,\tau}(s). \\
\mathcal{WN}^{\Gamma}_{\tau}(t) & \textit{if } \mathcal{W}^{\Gamma}_{\tau}(t). \\
\mathcal{WN}^{\Gamma}_{\tau}(t) & \textit{if } \Gamma \vdash t : \tau,\ t \longrightarrow t' \textit{ and } \mathcal{WN}^{\Gamma}_{\tau}(t').
\end{array}
$$

Figure 3.4 – Logical relation for weak normalisation.

To complete the definition of the logical relations, the completeness and soundness of $\mathcal{WN}^{\Gamma}_{\tau}$ are proved below. An auxiliary lemma of substitution of normal forms is also demonstrated.

Lemma 3.10 (Substitution in $\mathcal{WN}$)
Consider $\mathcal{WN}^{\Gamma}_{\sigma}(t')$. If $\mathcal{WN}^{\Gamma'}_{\tau}(t)$ where $\Gamma' = \Gamma, z^{\sigma}$ then $\mathcal{WN}^{\Gamma}_{\tau}(t\,[z := t'])$.

Proof. Suppose that $\mathcal{WN}^{\Gamma}_{\sigma}(s)$ **(1)**. To prove that $\mathcal{WN}^{\Gamma}_{\tau}(t\,[x := s])$ we proceed by induction over $\mathcal{WN}^{\Gamma'}_{\tau}(t)$.

- Case $\mathcal{WN}^{\Gamma'}_{\sigma\to\tau}(\lambda x.\, s)$.
  The substitution gives the term $\lambda x.\, s\,[z := t']$. In order to show that this term belongs to relation $\mathcal{WN}$ we must prove that the body-term also belongs to the relation. This holds by the induction hypothesis $\mathcal{WN}^{\Gamma', x^{\sigma}}_{\tau}(s)$.
- Case $\mathcal{WN}^{\Gamma'}_{\tau}(r\, s)$.
  By induction hypothesis, we know that $\mathcal{WN}^{\Gamma}_{\sigma\to\tau}(r\,[z := t'])$ and $\mathcal{WN}^{\Gamma}_{\sigma}(s\,[z := t'])$. Following the definition of the logical relation we can conclude that
  $\mathcal{W}^{\Gamma}_{\tau}(r\,[z := t']\, s\,[z := t'])$.
- Case $\mathcal{WN}^{\Gamma'}_{\mathsf{M}\,\tau}(\mathsf{unit}\ r)$.
  The term $r\,[z := t']$ belongs to relation $\mathcal{WN}^{\Gamma}_{\tau}$ by induction hypothesis and therefore, by definition, the term $\mathsf{unit}\ r\,[z := t']$ belongs to the relation $\mathcal{WN}^{\Gamma}_{\mathsf{M}\,\tau}$.
- Case $\mathcal{WN}^{\Gamma'}_{\mathsf{M}\,\tau}(\mathsf{bind}\, r\, s)$.
  The induction hypotheses $\mathcal{WN}^{\Gamma}_{\mathsf{M}\,\sigma}(r\,[z := t'])$ and $\mathcal{WN}^{\Gamma}_{\sigma\to\mathsf{M}\,\tau}(s\,[z := t'])$ allow to conclude that $(\mathsf{bind}\, r\, s)\ [z := t'] = \mathsf{bind}\ (r\,[z := t'])\ (r\,[z := t'])$ belongs to $\mathcal{WN}^{\Gamma}_{\mathsf{M}\,\tau}$.
- Case $\mathcal{W}^{\Gamma'}_{\tau}(t)$.
  The relation $\mathcal{W}$ has three cases:
  - $t = x$ with two sub-cases
    If $x = z$, then $\mathcal{W}^{\Gamma'}_{\tau}(z)$ holds since after substitution of term t', the term belongs to relation $\mathcal{WN}$ by hypothesis (1).
    If $x \neq z$ then $\mathcal{WN}^{\tau}_{\Gamma}(x)$ already holds.

    - $t = \mathbf{c}$ which is not affected by the substitution.
    - $t = r\,s$ which holds by induction hypotheses of its sub-terms.

- Case $\mathcal{WN}^{\Gamma'}_{\tau}(r)$ where $r \longrightarrow r'$.
  From induction hypothesis, $\mathcal{WN}^{\Gamma}_{\tau}(r'[z := t'])$ and following the last case of the definition of relation $\mathcal{WN}$, then $\mathcal{WN}^{\Gamma}_{\tau}(r[z := t'])$ holds. □

Lemma 3.11 (Completeness of $\mathcal{WN}^{\Gamma}_{\tau}$)
If $\Gamma \vdash t : \tau$ then $\mathcal{WN}^{\Gamma}_{\tau}(t)$.

Proof. Induction over $\Gamma \vdash t : \tau$. We use the the inversion Lemma 3.3 in the inductive cases.

- Case $\Gamma \vdash x : \tau$
  The variable x belongs to the typing context Γ, then it is in relation $\mathcal{W}^{\Gamma}_{\tau}$ and therefore it also belongs to $\mathcal{WN}^{\Gamma}_{\tau}$.
- Case $\Gamma \vdash \lambda x.\, s : \sigma \to \tau$
  The inversion Lemma assigns type τ to the body-term s under Γ, x^{σ}. Then we can instantiate the induction hypothesis: $\mathcal{WN}^{\Gamma, x^{\sigma}}_{\tau}(s)$ This hypothesis ensures that the abstraction belongs to relation $\mathcal{WN}^{\Gamma}_{\sigma\to\tau}$.
- Case $\Gamma \vdash r\,s : \tau$
  By inversion of typing, we ensure that there exists a type σ such that $\Gamma \vdash r : \sigma \to \tau$ and $\Gamma \vdash s : \sigma$, then we can suppose the induction hypotheses: $\mathcal{WN}^{\Gamma}_{\sigma\to\tau}(r)$ and $\mathcal{WN}^{\Gamma}_{\sigma}(s)$. We can conclude that the application $r\,s$ belongs to relation $\mathcal{WN}^{\Gamma}_{\tau}$ since each sub-term also belongs to the relation.
- Case $\Gamma \vdash \mathsf{unit}\ t : \mathsf{M}\,\tau$
  The induction hypothesis state that $\mathcal{WN}^{\Gamma}_{\tau}(t)$ since by inversion $\Gamma \vdash t : \tau$. Therefore, $\mathsf{unit}\ t$ belongs to $\mathcal{WN}^{\Gamma}_{\mathsf{M}\,\tau}$.
- Case $\Gamma \vdash \mathsf{bind}\, r\, s : \mathsf{M}\,\tau$
  The inversion ensures that there exists a type σ such that $\Gamma \vdash r : \mathsf{M}\,\sigma$ and $\Gamma \vdash s : \sigma \to \mathsf{M}\,\tau$. Then, we obtain two induction hypotheses: $\mathcal{WN}^{\Gamma}_{\mathsf{M}\,\sigma}(r)$ and $\mathcal{WN}^{\Gamma}_{\sigma\to\mathsf{M}\,\tau}(s)$. They allow us to conclude that $\mathsf{bind}\, r\, s$ belongs to relation $\mathcal{WN}^{\Gamma}_{\mathsf{M}\,\tau}$. □

Lemma 3.12 (Soundness of $\mathcal{WN}^{\Gamma}_{\tau}$)
For any term such that $\mathcal{WN}^{\Gamma}_{\tau}(t)$, then it has a normal form.

Proof. Induction over $\mathcal{WN}^{\Gamma}_{\tau}(t)$. We will show that term t is reducible until reaching a normal form $\hat{t}$.

- Case $\mathcal{WN}^{\Gamma}_{\sigma\to\tau}(\lambda x.\, s)$
  A lambda abstraction does not reduce, it is a normal form.
- Case $\mathcal{WN}^{\Gamma}_{\tau}(r\,s)$
  By definition we have the following hypotheses: $\mathcal{WN}^{\Gamma}_{\sigma\to\tau}(r)$ and $\mathcal{WN}^{\Gamma}_{\sigma}(s)$. Then, by induction over these hypotheses there exist the normal forms $\hat{r}$ and $\hat{s}$ for terms r and s respectively. And therefore the application reaches the term $\hat{r}\,\hat{s}$.

The above term could have no more reductions and in that case it is a normal form.
If we can go forward in one more reduction step, it is necessarily when the normal forms have the following forms: $\hat{r} = \lambda x.\, t'$ for a variable x and a term t', and $\hat{s} = v$ for a value v. Both normal forms belong to the corresponding type-indexed relation $\mathcal{WN}$. A reduction is possible, by rule β-RED, into term $t'\,[x \;:=\; v]$. In order to prove that this term has a normal form we proceed by arguing two facts.
First, that the sub-term t' has a normal form, since by definition $\mathcal{WN}^{\Gamma}_{\tau}\,(\lambda x.\, t')$.
And second, that the substitution of v gives a normal form as proved in Lemma 3.10.
Therefore $t'\,[x \;:=\; v]$ has a normal form which is the normal form of $r\,s$.

- Case $\mathcal{WN}^{\Gamma}_{\mathsf{M}\,\tau}\,(\mathsf{unit}\ t')$
  Following the definition of relation $\mathcal{WN}^{\Gamma}_{\mathsf{M}\,\tau}$, the sub-term t' also belongs to the relation. Then, by induction hypothesis the term t' has a normal form say $\hat{t'}$ and therefore the reduction of the term $\mathsf{unit}\ t'$ has the normal form $\mathsf{unit}\ \hat{t'}$.
- Case $\mathcal{WN}^{\Gamma}_{\mathsf{M}\,\tau}\,(\mathsf{bind}\,r\,s)$
  By induction hypotheses we ensure the normal forms $\hat{r}$ and $\hat{s}$ exist since by definition $\mathcal{WN}^{\Gamma}_{\mathsf{M}\,\sigma}\,(r)$ and $\mathcal{WN}^{\Gamma}_{\sigma\to\mathsf{M}\,\tau}\,(s)$.
  Then, to show that $\mathsf{bind}\,\hat{r}\,\hat{s}$ has a normal form we analyse the term itself. If one of the normal forms is a variable, then the term is a normal form.
  If both sub-terms are not variables and have the following forms: $\hat{r} = \mathsf{unit}\ r$ and $\hat{s} = \lambda x.\, t'$, then we can perform one step reduction to $t'\,[x \;:=\; r]$. This term has a normal form following definition $\mathcal{WN}$ and Lemma 3.10.
- Case $\mathcal{W}^{\Gamma}_{\tau}\,(t)$
  This case has two main sub-cases:
  - $t = x$ or $t = \mathbf{c}$. A variable or a constant are normal forms.
  - $t = r\,s$. By definition $\mathcal{W}^{\Gamma}_{\sigma\to\tau}\,(r)$ and $\mathcal{WN}^{\Gamma}_{\sigma}\,(s)$. Then by induction hypothesis, both terms have a normal form and an application of normal forms is a normal form. We cannot do a reduction step since the terms in relation $\mathcal{W}$ include only variables and constants.
- Case $\mathcal{WN}^{\Gamma}_{\tau}\,(t')$ where $t \longrightarrow t'$.
  By definition, $\mathcal{WN}^{\Gamma}_{\tau}\,(t')$ holds and therefore there exists a normal form of term t', say $\hat{t'}$ which is also the normal form of t since there is only one reduction from t to t'. □

Monadic normal forms

The normal forms are useful to distinguish a subset of computations in the language λ_{M}. Moreover, we want to use them to characterise the reduced computations that need a prophecy in order to complete their reduction. This is achieved by the *monadic normal forms*, a subset of the normal forms defined in Figure 3.3 whose returning type is always monadic and may have as many arguments as wanted. These types are defined by the following grammar:

$$\rho ::= \tau \to \rho \mid \mathsf{M}\,\tau$$

Definition 3.3 (Monadic normal forms)
For all t such that $\Gamma \vdash t : \rho$, there exists a monadic normal form $\hat{t}$ and t is $\beta\delta$-convertible to $\hat{t}$.

We ensure that each well-typed computation in λ_{M}, is likely to be reduced into a monadic normal form. We do not include the terms $\Downarrow_p t$ as computations, since this construction serves only as a mechanism to perform simulations. As we mentioned before, the terms $\Downarrow_p t$ can make progress only by making progress on the inner computation t and later on we will show that they can be reduced with the assistance of a prophecy.

The language λ_{M} includes the operators for a simulable monad where the prophecies are given terms, which as we mentioned before, are values computed in the effectful language. The next section presents a language with effectful primitives and the last requirements to complete the *a posteriori simulation* framework.

3.2 $\lambda_{v,\perp}$ a call-by-value impure functional language

The impure functional and non-terminating language $\lambda_{v,\perp}$ has the same syntax as λ_{M} except that for this language, the set of constants only consists of effectful primitive operators. The language is also equipped with an *instrumented* big-step operational semantics for a call-by-value reduction strategy. We chose a big-step semantics since we only focus on converging executions of compiled terms. The executions are carried out under environments η assigning closed values to variables. Closed values comprise full applications of effectful constants to values and function closures:

$$\eta ::= \bullet \mid \eta\,;\, x \mapsto v \qquad\qquad v ::= \boldsymbol{c}\,\overline{v} \mid (\lambda x.\, s)\,[\eta]$$

The judgement for the instrumented semantics is $\eta \vdash s \Downarrow_{p \to p'} v$, which is read as 'the execution of a term s under the environment η converges to a value v and computes a prophecy p' from an initial prophecy p'. The reader may notice that a standard big-step operational semantics can be recovered by erasing the annotation $p \longrightarrow p'$ in each judgement of Figure 3.5, meaning that the evaluation of terms does not depend on the computation of prophecies. That is, the rules just carry the prophecies while the rules for effectful constants, which are abstract for now, are responsible for increasing the prophecies. They will be characterised later by the Requirement 4.

$$\textsc{Eval-Var}\ \frac{}{\eta \vdash x \Downarrow_{p \to p} \eta\ (x)} \qquad\qquad \textsc{Eval-Lam}\ \frac{}{\eta \vdash \lambda x.\, s \Downarrow_{p \to p} (\lambda x.\, s)\,[\eta]}$$

$$\textsc{Eval-App}\ \frac{\eta \vdash s_1 \Downarrow_{p \to p_1} (\lambda x.\, s)\,[\eta'] \qquad \eta \vdash s_2 \Downarrow_{p_1 \to p_2} v_1 \qquad \eta'; x \mapsto v_1 \vdash s \Downarrow_{p_2 \to p'} v}{\eta \vdash s_1\, s_2 \Downarrow_{p \to p'} v}$$

Figure 3.5 – Instrumented semantics of $\lambda_{v,\perp}$

The purpose of the instrumented semantics is to *monotonically* refine the prophecy at each step of the computation in order to recover enough information for a successful simulation. The third requirement ensures this property:

Requirement 3 (Monotonicity of prophecy computation)

$$\forall\, p,\ p',\ \eta \vdash s \Downarrow_{p \to p'} v \ \textit{ implies } \ p \leqslant p'.$$

Compilation

We recall one of the key features in our lightweight approach to proof by reflection: the connection between the monadic and the effectful languages to obtain the prophecies needed to simulate computations. In order to connect both languages, as we mentioned in the introduction, we use a compilation function from the monadic language to the effectful language.

The compilation is defined in Figure 3.6, it replaces the monadic constructs unit and bind with their respective definitions in the identity monad and converts each effectful primitive $\mathbf{c} \in \nabla$ into the corresponding impure primitive construction of $\lambda_{v,\perp}$ [5].

The translation of $\Downarrow_p$ is explicitly *undefined* as a consequence of the fact that this operator cannot appear in any well-typed *user-written* monadic term because it is only useful for a simulation of computations in language λ_{M}.

TERM COMPILATION

$$\begin{array}{rcl} \mathcal{C}(x) & = & x \\ \mathcal{C}(\lambda x.\, s) & = & \lambda x.\, \mathcal{C}(s) \\ \mathcal{C}(r\, s) & = & \mathcal{C}(r)\, \mathcal{C}(s) \end{array} \qquad \begin{array}{rcl} \mathcal{C}(\mathsf{unit}) & = & \lambda x.\, x \\ \mathcal{C}(\mathsf{bind}) & = & \lambda x, y.\, y\, x \\ \mathcal{C}(\Downarrow_p) & = & \text{undefined} \end{array}$$

TYPE COMPILATION

$$\begin{array}{rcl} \mathcal{C}(\sigma \to \tau) & = & \mathcal{C}(\sigma) \to \mathcal{C}(\tau) \\ \mathcal{C}(\mathsf{M}\, \tau) & = & \mathcal{C}(\tau) \end{array} \qquad \begin{array}{rcl} \mathcal{C}(\varnothing) & = & \varnothing \\ \mathcal{C}(\Gamma,\, x^{\tau}) & = & \mathcal{C}(\Gamma),\, x^{\mathcal{C}(\tau)} \end{array}$$

TYPING CONTEXT COMPILATION

Figure 3.6 – Compilation of terms, types and typing contexts from λ_{M} to $\lambda_{v,\perp}$.

Prophecies are common to both languages but they have different purposes. In $\lambda_{v,\perp}$, the type for prophecies is kept fully abstract to the programmer, it is not available since only the instrumented compiled code is allowed to generate prophecies. While in language λ_{M} the prophecies are treated as given elements which are valuable for reduction.

The following lemmas collect the properties ensuring the well behaviour of compilation.

5. Recall that there is no specific monad M as parameter and our monadic constants remain abstract.

Lemma 3.13 (Compilation and substitution)
Compilation commutes with substitution: $\mathcal{C}(t\,[z\ :=\ t']) = \mathcal{C}(t)[z\ :=\ \mathcal{C}(t')]$.

Proof. Induction on t, using definition in Figure 3.6.

- Case $t = y$.
  Suppose that $y = z$, then the compilation of the substitution gives $\mathcal{C}(t')$ which is the same result when applying the substitution to the compilation of variable z.
  If $y \neq z$ then the substitution does not affect neither y nor $\mathcal{C}(y)$ and therefore both sides are equal.
- Case $t = \lambda y.\, s$.
  The substitution applied to the λ-abstraction internalises it and therefore the compilation of term $\lambda y.\, s\,[z\ :=\ t']$ is $\lambda y.\, \mathcal{C}(s\,[z\ :=\ t'])$.
  On the other side, the application of the substitution to the compilation of $\lambda y.\, s$ gives $\lambda y.\, \mathcal{C}(s)[z\ :=\ \mathcal{C}(t')]$.
  Then we can conclude that both terms are equal since the body-terms are equal by induction hypothesis.
- Case $t = r\, s$.
  The induction hypotheses are the following: $\mathcal{C}(r\,[z\ :=\ t']) = \mathcal{C}(r)\,[z\ :=\ \mathcal{C}(t')]$ and $\mathcal{C}(s\,[z\ :=\ t']) = \mathcal{C}(s)\,[z\ :=\ \mathcal{C}(t')]$.
  Then $\mathcal{C}(r\,[z\ :=\ t'])\,\mathcal{C}(s\,[z\ :=\ t'])$ and $\mathcal{C}(r)\,[z\ :=\ \mathcal{C}(t')]\,\mathcal{C}(s)\,[z\ :=\ \mathcal{C}(t')]$ are equal.
- Case $t = \mathsf{unit}\ r$.
  On one side, we have the compilation of term $(\mathsf{unit}\ r)\ [z\ :=\ t']$, which is the application $(\lambda x.\, x)\ (\mathcal{C}(r\,[z\ :=\ t']))$ since the substitution passes through the constructor unit .
  On the other side, the compilation of term $\mathsf{unit}\ r$ is the application $(\lambda x.\, x)\ \mathcal{C}(r)$, which after applying the substitution is $(\lambda x.\, x)\ [z\ :=\ t']\,\mathcal{C}(r)\,[z\ :=\ \mathcal{C}(t')]$. The first substitution does not affect the term $\lambda x.\, x$ and the induction hypothesis $\mathcal{C}(r[z\ :=\ t']) = \mathcal{C}(r)[z\ :=\ \mathcal{C}(t')]$ leads to conclude that both sides are equal.
- Case $t = \mathsf{bind}\, r\, s$.
  The compilation of a bind term is the application of the abstraction $\lambda x, y.\, y\, x$ to the compilation of its sub-terms, $\mathcal{C}(r)$ and $\mathcal{C}(s)$.
  Then, on one side we have the compilation of term $(\mathsf{bind}\, r\, s)\ [z\ :=\ \mathcal{C}(t')]$ which gives the term $(\lambda x, y.\, y\, x)\ \mathcal{C}(r\,[z\ :=\ t'])\,\mathcal{C}(s\,[z\ :=\ t'])$
  On the other side, the application of the substitution to the compilation gives the term: $((\lambda x, y.\, y\, x)\ \mathcal{C}(r)\,\mathcal{C}(s))\ [z\ :=\ t']$.
  We can conclude that both sides are equal by internalising the substitution in the last term above and by applying the induction hypotheses for terms r and s. □

Lemma 3.14 (Compilation and typing)
If $\Gamma \vdash t : \tau$ in λ_{M}, then $\mathcal{C}(\Gamma) \vdash \mathcal{C}(t) : \mathcal{C}(\tau)$ is a valid typing judgement in $\lambda_{v,\perp}$.

Proof. Induction over $\Gamma \vdash t : \tau$. We use the same typing rules as in Figure 3.2 to assign a type to variables, λ-abstractions and applications.

- Case $\Gamma \vdash x : \tau$.
  The variable x belongs to the type environment Γ. Then the compilation of the environment also has the variable x whose type is $\mathcal{C}(\tau)$. Therefore the judgement $\mathcal{C}(\Gamma) \vdash x : \mathcal{C}(\tau)$ is well formed.
- Case $\Gamma \vdash \lambda y.\, s : \sigma \to \tau$.
  The compilation of a λ-abstraction is also an abstraction but this time over the compiled body-term, $\lambda y.\, \mathcal{C}(s)$. In order to verify that the judgement $\mathcal{C}(\Gamma) \vdash \lambda y.\, \mathcal{C}(s) : \mathcal{C}(\sigma) \to \mathcal{C}(\tau)$ is well-formed we use the induction hypothesis: $\mathcal{C}(\Gamma), y^{\mathcal{C}(\sigma)} \vdash \mathcal{C}(s) : \mathcal{C}(\tau)$.
- Case $\Gamma \vdash r\, s : \tau$.
  An application is compiled by compiling its sub-terms. Then, by the induction hypotheses for terms r and s the following judgement holds: $\mathcal{C}(\Gamma) \vdash \mathcal{C}(r)\, \mathcal{C}(s) : \mathcal{C}(\sigma) \to \mathcal{C}(\tau)$.
- Case $\Gamma \vdash \mathsf{unit}\ t : \mathsf{M}\,\tau$.
  The compilation of term $\mathsf{unit}\ t$ is an application: $(\lambda x.\, x)\ \mathcal{C}(t)$.
  Consider the induction hypothesis of term t, $\mathcal{C}(\Gamma) \vdash \mathcal{C}(t) : \mathcal{C}(\tau)$ **(1)**. We can also give the specific type $\mathcal{C}(\tau) \to \mathcal{C}(\tau)$ to the term $\lambda x.\, x$ **(2)** under the same typing environment $\mathcal{C}(\Gamma)$. Therefore, using the rule of application App with (1) and (2) we conclude that the following judgement is well-formed: $\mathcal{C}(\Gamma) \vdash (\lambda x.\, x)\ \mathcal{C}(t) : \mathcal{C}(\tau)$ where $\mathcal{C}(\mathsf{M}\,\tau) = \mathcal{C}(\tau)$.
- Case $\Gamma \vdash \mathsf{bind}\, r\, s : \mathsf{M}\,\tau$.
  For the bind operator, the corresponding compilation is also an application:
  $(\lambda x, y.\, y\, x)\ \mathcal{C}(r)\, \mathcal{C}(s)$.
  To assign a type to the above term, we use the induction hypotheses: $\mathcal{C}(\Gamma) \vdash \mathcal{C}(r) : \mathcal{C}(\mathsf{M}\,\sigma)$ and $\mathcal{C}(\Gamma) \vdash \mathcal{C}(s) : \mathcal{C}(\sigma) \to \mathcal{C}(\mathsf{M}\,\tau)$, and the typing judgement for the abstraction $\lambda x, y.\, y\, x$ whose type is obtained by taking the types of the induction hypotheses: $\mathcal{C}(\mathsf{M}\,\sigma) \to (\mathcal{C}(\sigma) \to \mathcal{C}(\mathsf{M}\,\tau)) \to \mathcal{C}(\mathsf{M}\,\tau)$. □

Prophecies

The evaluation of the translation of an effectful constant $\mathbf{c}$ in ∇, must extend the prophecy in a sufficient way to make the simulation converge. This property is the last requirement of a monad to be simulable, it details the *collection* of information to obtain a prophecy.

Requirement 4 (Adequate instrumented compilation)

$$\forall p_0, \ldots, p_{n+1}, p,\ \ \textit{if} \left\{ \begin{array}{ll} \forall i, & \eta \vdash \mathcal{C}(t_i) \Downarrow_{p_i \to p_{i+1}} v_i \\ & \eta \vdash \mathcal{C}(\mathbf{c}\ (t_0, \ldots, t_n)) \Downarrow_{p_0 \to p} v \end{array} \right. \quad \textit{then} \ \ \exists\, u, \Downarrow_p \mathbf{c}\ (t_0, \ldots, t_n) = \star u$$

The evaluation of the compilation of a constant increases the prophecy by augmenting it at each sub-term evaluation, that is, if the constant has sub-terms, the prophecy depends on the prophecies of the evaluation of them. If the constant has a zero-arity, then the increment in the prophecy must be ensured by the definition of the evaluation rule for it. In this way, the semantics of $\lambda_{v,\perp}$ ensures a proper growth of the initial prophecy.

The above requirement ends Definition 3.1 characterising what a simulable monad is: a standard monad (Requirement 1) with an operator to *simulate* computations with the help of prophecies (Requirements 2 and 0), which are obtained after the evaluation of the compilation (Requirements 3 and 4).

In Section 3.4 we will discuss some examples to made more reliable this definition, while in the next section we demonstrate that the simulation of effects in λ_{M} is correct.

3.3 *A posteriori* simulation of effects

Up to now, we have described two languages, λ_{M} and $\lambda_{v,\perp}$, to formalize the lightweight approach to proof by reflection, the goal of this chapter. These languages are presented as two separate entities which have in common some syntax, along with the corresponding typing rules, whose operational semantics only differ on the approach of small-step or big-step reduction of terms and the most remarkable feature, the capability to write and simulate or evaluate imperative terms or computations.

The aim of this section is to exhibit a close relationship between these systems. Through the main theorem 3.16, we claim that the witness obtained in $\lambda_{v,\perp}$ is used as a valid prophecy to perform a simulation back in λ_{M} to get a computational term under the unit constructor. Specifically, it states that given a computation t, if the evaluation of $\mathcal{C}(t)$ converges, then the prophecy p obtained in the evaluation process is enough to simulate t back in λ_{M}.

Dynamic Semantics and other remarks Even if the language λ_{M} has monadic terms[6], the translation or compilation into language $\lambda_{v,\perp}$ maps every monadic term into a term in the simply typed lambda calculus with effectful primitives. There, we consider only the terms which converge using the big-step semantics.

We also recall that the instrumented semantics of $\lambda_{v,\perp}$ does not influence the reduction process, but just collect information to have prophecies. Moreover, while evaluating a term in $\lambda_{v,\perp}$, it does not matter on which prophecy we start an evaluation, as long as the evaluation converges, then it will generate a *sufficient* prophecy as ensured by the dynamic semantics: the evaluation of values does not change the starting prophecy and the prophecy growth is carry out by rule EVAL-APP and ensured by Requirements 3 and 4.

In the following, we show that the simulation of computations is correct by using an auxiliary lemma to prove the simulation of monadic normal forms[7].

6. The simulation operator $\Downarrow$ is not taken into account since it does not have a translation as discussed in Section 3.2.

7. Recall that a monadic normal form has a type of the form $\rho ::= \tau \to \rho \mid \mathsf{M}\,\tau$.

Theorem 3.15 (*A posteriori* simulation of normal forms)
Let $\hat{t}$ be a monadic normal form, $\varnothing \vdash \hat{t} : \tau_0 \to \cdots \to \tau_n \to \mathsf{M}\,\tau$. Then, for all terms s_i such that $\varnothing \vdash s_i : \tau_i$, if the compilation of $\mathcal{C}(\hat{t}\,s_0 \ldots s_n)$ converges, $\bullet \vdash \mathcal{C}(\hat{t}\,s_0 \ldots s_n) \Downarrow_{p \to p'} v$, then there exists a term r such that $\Downarrow_{p'} \left(\hat{t}\,s_0 \ldots s_n\right)$ reduces to $\mathsf{unit}\ r$.

Proof. Induction on $\hat{t}$, defined in Figure 3.3.

- Case $\hat{t} = \mathsf{unit}\ \hat{s}$.
  We want to prove that there exits r such that $\Downarrow_{p'} \mathsf{unit}\ \hat{s} = \mathsf{unit}\ r$. Requirement 2 allows to conclude that $r = \hat{s}$.
- Case $\hat{t} = \mathsf{bind}\,m\,\hat{s}$.
  Let us analyse the hypotheses, the typing and the compilation of $\hat{t}$.
  By inversion Lemma 3.3, there exists a type σ such that the type of m is monadic, $\mathsf{M}\,\sigma$ **(1)**, and the term $\hat{s}$ has function type $\sigma \to \mathsf{M}\,\tau$ **(2)**.
  The derivation of evaluation $\bullet \vdash \mathcal{C}(\mathsf{bind}\,m\,\hat{s}) \Downarrow_{p \to p'} v$ where the compilation of term $\mathsf{bind}\,m\,\hat{s}$ is the application $(\lambda x, y.\, y\, x)\ \mathcal{C}(m)\,\mathcal{C}(\hat{s})$, has two significant sub-derivations: $\bullet \vdash \mathcal{C}(m) \Downarrow_{p \to p_1} v_1$ **(3)** and $\bullet \vdash \mathcal{C}(\hat{s}) \Downarrow_{p_1 \to p_2} v_2$ **(4)**.
  Now, we proceed to prove that the simulation of $\mathsf{bind}\,m\,\hat{s}$ reduces to $\mathsf{unit}\ r$ for a term r using prophecy p'. Requirement 2 allows to internalise the simulation and then the term to be reduced is $\Downarrow_p \mathsf{bind}\ (\Downarrow_p m)\ \hat{s}$.
  The induction hypothesis is available for term m, since it is a sub-term of $\hat{t}$ and the hypotheses (1) and (3) hold. Therefore there exists a term r' such that $\Downarrow_{p'} m = \mathsf{unit}\ r'$ and we can continue the reduction of $\Downarrow_p \mathsf{bind}\ (\mathsf{unit}\ r')\ \hat{s}$ using rule COMP-RED. The evaluation process goes to $\Downarrow_p \hat{s}\,r'$. In order to prove that this last term reduces to $\mathsf{unit}\ r$ we apply the induction hypothesis supported by (2): for any argument applied to term $\hat{s}$, the compilation $\bullet \vdash \mathcal{C}(\hat{s}\,r') \Downarrow_{p_2 \to p'} w$ converges and therefore there exists a term r such that $\Downarrow_{p'} \hat{s}\,r' = \mathsf{unit}\ r$.
- Case $t = \hat{r}\,\hat{s}$.
  By inversion of the typing hypothesis $\varnothing \vdash \hat{r}\,\hat{s} : \tau_0 \to \cdots \to \tau_n \to \mathsf{M}\,\tau$, there exists a type σ such that $\varnothing \vdash \hat{s} : \sigma$ and $\varnothing \vdash \hat{r} : \sigma \to \tau_0 \to \cdots \to \tau_n \to \mathsf{M}\,\tau$.
  The evaluation $\bullet \vdash \mathcal{C}(\hat{r}\,\hat{s}\,s_0 \ldots s_n) \Downarrow_{p \to p'} v$ holds as hypothesis and then the proof is to show that $\Downarrow_{p'} \hat{r}\,\hat{s}\,s_0 \ldots s_n = \mathsf{unit}\ r$ for some term r.
  Note that the induction hypothesis holds for term $\hat{r}$ since it is a sub-term of t. Then we can conclude that there exists r such that $\Downarrow_p \hat{r}\,\hat{s}$ is equivalent to $\mathsf{unit}\ r$, since we already know that term $\hat{r}$ with its arguments $s_0 \ldots s_n$ reduces to $\mathsf{unit}\ r$ using the prophecy p'.
- Case $t = \mathbf{c}\,\bar{t}$.
  We claim that each effectful constant, when the monad M is instantiated, is reduced by the corresponding δ-rule of the dynamic semantics. This reduction together with the adequate prophecy gives a unit term, as ensured by the Requirement 4. ☐

Theorem 3.16 (*A posteriori* simulation)
Let $\varnothing \vdash t : \mathsf{M}\,\tau$ be a computation whose compilation converges, that is $\bullet \vdash \mathcal{C}(t) \Downarrow_{p \to p'} u$. Then, there exists a term r such that $\Downarrow_{p'} t = \mathsf{unit}\ r$.

Proof. Suppose that $\bullet \vdash \mathcal{C}(t) \Downarrow_{p \to p'} u$ for a closed and well-typed computation t.
Consider the normal form $\hat{t}$ of the term t that is, reducing $\Downarrow_{p'} t$ through rule RED where the evaluation context is $\Downarrow_{p'} \mathcal{E}$. The reduction of term t before requiring a prophecy is ensured by Lemma 3.3 in section 3.1.2.
We conclude by Lemma 3.15, there exists a term r such that $\Downarrow_{p'} t$ is equivalent to unit r.□

3.4 Examples of simulable monads

We discuss the non-termination and partiality effects as a simulable monad. For ease of reading and writing of terms we use the abbreviation $r \diamond s$ for: $\mathsf{bind}\, r\ (\lambda x.\ \mathsf{bind}\, s\ (\lambda y.\ (x\, y)))$.

The "trace" prophecy. First, notice that, given a monad M with an underlying effectful computation model specified by a reduction relation, there is always a prophecy to simulate a converging effectful reduction: the reduction chain itself. Indeed, it suffices to define the type of prophecy as the abstract syntax trees of an impure programming language that implements the effectful computational model. The operator $\Downarrow$ can then be implemented as an interpreter for these abstract syntax trees defined by induction over the length of the converging reduction chain. However, while this prophecy is very informative, it is such a naive implementation of prophecies which is not efficient because of the interpretation overhead.

Non-termination and partiality. The type $\mathsf{M} = \mathsf{nat} \to \mathsf{Option}\, \tau$ defines an adequate monad to represent non-terminating computations of type τ [8]. A general fix-point operator is defined by induction over a natural number as input representing the number of the steps of recursion. If this number is large enough then the computation *terminates* and produces a result t. In case of termination, the result of a computation is denoted by $\mathsf{Some}\, t$, the non-termination is denoted by None. For this monad, the expected type for prophecies is nat and the instrumentation only has to compute an *over-approximation* of the number of iterations of all the fix-points of the program. We present the corresponding extensions to the languages λ_{M} and $\lambda_{v,\perp}$.

In system λ_{M}, the extension is guided by the introduction of two type constructors: nat for natural numbers and $\mathsf{Option}\, \tau$ for partial computations of type τ. This extension is defined in Figure 3.7.

The constants for natural numbers include the (infinite) numerals and the binary operations of addition and multiplication. The Option type, also known as the Maybe type, is used when referring results may fail. This type encapsulates meaningful results under the Some constructor and the no-result value is specified by the empty constructor None. We also provide case analysis for both types: the recn destructor allows a possible primitive recursion over natural numbers and the $\mathsf{matchOpt}$ supplying the analysis over partiality terms.

8. This approach follows the standard non-termination monad, where termination is gained when a computation is a function taking an *approximation level* n as argument to compute an optimal result.

$\boxed{\text{Syntax}}$

$$\begin{array}{rcl} c & ::= & \cdots \mid \widehat{n} \mid + \mid \times \mid \mathsf{recn} \mid \mathsf{None} \mid \mathsf{Some} \mid \mathsf{matchOpt} \\ v & ::= & \cdots \mid \widehat{n} \mid \mathsf{None} \mid \mathsf{Some}\, v \\ \tau, \sigma & ::= & \cdots \mid \mathsf{nat} \mid \mathsf{Option} \\ \mathcal{E} & ::= & \cdots \mid \mathsf{recn}\, \mathcal{E}\, r\, s \mid \mathsf{recn}\, v\, \mathcal{E}\, s \mid \mathsf{recn}\, v\, u\, \mathcal{E} \\ & & \mid \mathsf{Some}\, \mathcal{E} \mid \mathsf{matchOpt}\, \mathcal{E}\, r\, s \mid \mathsf{matchOpt}\, v\, \mathcal{E}\, s \mid \mathsf{matchOpt}\, v\, u\, \mathcal{E} \end{array}$$

$\boxed{\text{Dynamic Semantics}}$

$$\mathsf{recn}\ \widehat{0}\, v\, u \xrightarrow{\delta_{\mathsf{recn}}} v \qquad \mathsf{recn}\ (\widehat{n+1})\, v_1\, v_2 \xrightarrow{\delta_{\mathsf{recn}}} v_2\ (\mathsf{recn}\ \widehat{n}\, v_1\, v_2)$$

$$\mathsf{matchOpt}\, \mathsf{None}\, v_1\, v_2 \xrightarrow{\delta_{\mathsf{matchOpt}}} v_1 \qquad \mathsf{matchOpt}\ (\mathsf{Some}\, v)\ v_1\, v_2 \xrightarrow{\delta_{\mathsf{matchOpt}}} v_2\, v$$

$\boxed{\text{Static Semantics}}$

$$\frac{}{\Gamma \vdash \widehat{n} : \mathsf{nat}}\ \textsc{Num} \qquad \frac{\Gamma \vdash t_i : \mathsf{nat}}{\Gamma \vdash +\, t_1\, t_2 : \mathsf{nat}}\ \textsc{Add} \qquad \frac{\Gamma \vdash t_i : \mathsf{nat}}{\Gamma \vdash *\, t_1\, t_2 : \mathsf{nat}}\ \textsc{Mult}$$

$$\frac{\Gamma \vdash t : \mathsf{nat} \qquad \Gamma \vdash r : \tau \qquad \Gamma \vdash s : \tau \to (\mathsf{nat} \to \tau)}{\Gamma \vdash \mathsf{recn}\, t\, r\, s : \tau}\ \textsc{Recn}$$

$$\frac{}{\Gamma \vdash \mathsf{None} : \mathsf{Option}\, \tau}\ \textsc{None} \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{Some}\, t : \mathsf{Option}\, \tau}\ \textsc{Some}$$

$$\frac{\Gamma \vdash t : \mathsf{Option}\, \sigma \qquad \Gamma \vdash r : \tau \qquad \Gamma \vdash s : \sigma \to \tau}{\Gamma \vdash \mathsf{matchOpt}\, t\, r\, s : \tau}\ \textsc{MatchOpt}$$

Figure 3.7 – Non-termination and partiality.

The basic combinators of the partiality monad are defined as:

$$\begin{array}{rcl} \mathsf{unit}\ t & \stackrel{\text{def}}{=} & \lambda x.\, \mathsf{Some}\, t \\ \mathsf{bind}\, t_1\, t_2 & \stackrel{\text{def}}{=} & \lambda x.\, \mathsf{matchOpt}\ (t_1\, x)\ \mathsf{None}\ (\lambda y.\ (t_2\, y)\ x) \end{array}$$

While the type of prophecies P is nat, the simulation operator is defined by case analysis on the reduction of the application of a natural number to a monadic term:

$$\Downarrow_p t \stackrel{\text{def}}{=} \mathsf{matchOpt}\ (t\, p)\ (\lambda m.\, t\, m)\ (\lambda x.\, \mathsf{unit}\ x)$$

- If value $t\,p$ is None it means that the computation failed or diverged. We choose to *reset* the computation as another computation that *waits* for an extra natural number m to complete the evaluation. This choice expects that the oracle can produce a prophecy adequate to simulate it back.
- If there exists v such that u is $\mathsf{Some}\, v$, then we return v as a computation.

Remark that the evaluation contexts defined in Figure 3.7, impose an evaluation order from left to right in sub-terms of the matchOpt and recn constructors, while the dynamic semantics reduces terms when all sub-terms are values.

In λ_{M}, the effectful primitive rec $\in \nabla$ is a defined combinator denoting a general fix-point:

$$\begin{array}{rcl} \mathsf{rec}\, r & \overset{\text{def}}{=} & \lambda y, x.\, \mathsf{recn}\ x\ \ (\lambda z.\, \mathsf{None})\ \ t_1 \\ \text{where}\ \ t_1 & = & \lambda z.\ ((\Downarrow_x \mathsf{unit}\ r) \diamond (\Downarrow_x \mathsf{unit}\ z))\ y \end{array}$$

Recall that $\mathsf{M}\,\tau$ is defined as $\mathsf{nat} \to \mathsf{Option}\,\tau$, and therefore the derived typing rule for rec is the following:

$$\frac{\Gamma \vdash r : (\tau_1 \to \mathsf{M}\,\tau_2) \to \mathsf{M}\ (\tau_1 \to \mathsf{M}\,\tau_2)}{\Gamma \vdash \mathsf{rec}\, r : \mathsf{M}\ (\tau_1 \to \mathsf{M}\,\tau_2)}\ \textsc{Rec}$$

In $\lambda_{v,\perp}$, the constants are the same as the extension given in Figure 3.7, only the recursion constant is replaced by the fix_f operator which allows the computation of arbitrary fix-points:

$$\textsc{Eval-Fix}\ \frac{\eta\, ;\ f \mapsto \mathsf{fix}_f\, t \vdash t \Downarrow_{(p+1)\to p'} v}{\eta \vdash \mathsf{fix}_f\, t \Downarrow_{p\to p'} v}$$

We propose as compilation of $\mathcal{C}(\mathsf{rec})$ the term $\mathsf{fix}\, f$ and when evaluating the compilations in language $\lambda_{v,\perp}$, we propose as starting prophecy the value 1 to ensure an over approximation in the number of recursive calls needed to simulate the computation $\mathsf{fix}_f\, t$.

The above definitions verify requirements of Definition 3.1:

Requirement 0
We use the standard order over natural numbers whose minimal element $\perp$ is equal to 0.

Requirement 1
The monadic laws are verified by straight $\beta\delta$-reduction.

Requirement 2
The prophecies behave well while reducing computations:

- The reduction of $\mathsf{unit}\ r$ ignores any prophecy, that is when evaluating $\Downarrow_p \mathsf{unit}\ r$, the term matchOpt in the definition takes the second option since the application $(\mathsf{unit}\ r)\ p$ gives $\mathsf{Some}\, r$ and therefore the whole evaluation reduces to $\mathsf{unit}\ r$. Then $\Downarrow_p \mathsf{unit}\ t = \mathsf{unit}\ t$.
- When a computation has converged, that is when $\Downarrow_p t$ is equivalent to $\star r$ for some term r, the computation has been successfully evaluated in a fixed number of recursive-steps of recn, defined by the number p. Then the evaluation of the same term t with a natural number greater than p will also converge.
- In order to prove that $\Downarrow_p \mathsf{bind}\, t_1\, t_2 = \Downarrow_p \mathsf{bind}\,(\Downarrow_p\ t_1)\, t_2$ we proceed by replacing the corresponding definitions of the terms, performing reductions when permitted and by analysis of the values of terms $t_1\, p$ and $\lambda y.\ (t_2\, y)\ p$.

To complete this proof, it is needed to extend our definition of term equivalence from $\beta\delta$-equivalence, which at the end is a syntactical equivalence, to a *contextual equivalence* where the behaviours of two terms are compared observationally.

Requirements 3 and 4
The evaluation rule EVAL-FIX always increases the prophecy, therefore at the end of any evaluation of fix we will obtain a prophecy adequate to simulate the term.

Example (Factorial). *Given a natural number n, the following function computes the factorial number $n!$*

$$fact\ n \overset{def}{=} \textit{if } (n = 0) \textit{ then } 1 \textit{ else } (fact\ n - 1) \times n$$

Then, we can define the factorial function in λ_{M} as:

$$fact\ n \overset{def}{=} \mathsf{rec}\, r\, \widehat{n} \qquad \text{where} \;\; r = \lambda y.\, \mathsf{recn}\; y\, \widehat{1}\; (\lambda z.\, y \times z)$$

The computed prophecy p, by $\mathcal{C}(fact\ n)$, is obtained from evaluation $\bullet \vdash \mathsf{fix}_{fact}\, r\, \widehat{n} \Downarrow_{\widehat{1} \to p} v$, where the final prophecy p is equal to $\widehat{n+1}$ since at each step performed by rule [EVAL-FIX] *the new prophecy is increased one.*

3.5 Chapter conclusions

Proof by reflection is a powerful technique to simplify proofs in theorem provers, we claim that the lightweight approach presented and formalised in this chapter takes the best of two worlds, using type theory to write correct decision procedures and enjoying more facilities from effectful programming.

Type theory as a total language is a robust tool for developing proofs, extending it with general purpose programming primitives intensifies its expressive power. But programming with dependent types and partial functions add a challenge, the constraint of showing that any program terminates. We have shown that the simulation of effectful computations converges thanks to the prophecies obtained during execution of their compiled counterpart. That is, the new approach ensures efficient programs to be used as their own oracles and whose certificates or prophecies help to mimic them back in type theory. In this way, the challenge of proving termination of computations is ensured and is not managed explicitly by the programmer.

The formalisation of the underlying systems presented in this work are straightforward extensions of the simply typed lambda calculus.

The core language is system λ_{M} which is parametrized by a *simulable monad* M to have one specific effect. The constant $\Downarrow$ and the type constructor P are unusual with respect to the traditional monadic extensions of the lambda calculus. They are inspired in the notions of compilation and certificates [22, 24]. Any proposal of simulable monad M must include the definition of the compilation of its effectful primitives and show the fulfilment of the requirements.

The second language $\lambda_{v,\perp}$ is an extension whose constants are effectful operations.

The main theorem relates small-step semantics of λ_{M} and big-step semantics of $\lambda_{v,\perp}$: *let* $\cdot \vdash t : \mathsf{M}\,\tau$ *a computation which compilation converges to a value and produces a prophecy* p*, then there exists a term* r *such that* $\Downarrow_p t = \star r$ *i.e.* the prophecy simulates a computation.

The simulable monad given in Section 3.4 corresponds to the usual instances of monads for effects, where a monad models one particular and single effect. Of course, setting-up an effectful framework to write decision procedures must ensure a wider sort of effects than the general formalisation given so far. This can be achieved by giving an *ad-hoc* monad for having all the desired effects, that is a combination of monads as the well-known monad-transformers [81].

The lightweight approach to reflection with a monad including a combination of effects, is available through a prototype plug-in explained below.

Cybele plugin

The use of Coq as a framework to develop the new style of reflection has the benefits of programming with dependent types. This extension, due to Claret[9] and reported together with Régis-Gianas and Ziliani [30], is used to develop proofs using the method we described.

There is only one 'monolithic' monad used in Cybele to define the effectful operations of partiality, non-termination, state and printing, all together. The monad-type M is a combination of these effects and its definition is parametrized by a signature to type the memory operations:

$$\mathsf{M}\,\Sigma\,\alpha = \mathsf{State.t}\ \Sigma \to (\alpha + \mathsf{string}) \times \mathsf{State.t}\ \Sigma$$

The non-determinism can be programmed on top of the monad, and when simulating general recursion, one of the basic, most used and desired effects, the combination of $\Downarrow$ and recn perform a kind of delimited recursion. This implementation uses the extraction mechanism of Coq to obtain an OCaml program to be the oracle. The extraction definition follows the translation in Figure 3.6.

When using the plug-in, the user can witness what we claim as lightweight approach: we keep the power of the dependently-typed system of Coq despite the fact that we are working in a monad and we also get a great performance gained for type-checking, leaving to the user just the job of designing an efficient decision procedure whose proof of correctness will be a matter of a simple evaluation.

The formalisation of the lightweight proof by reflection is done entirely in the system λ_{M}. As the reader perceived all along this chapter, most of the proofs of lemmas and theorems related to the simulation process are accomplished by the reduction of terms where the use of prophecies is needed.

In the real implementation, the plug-in does the simulation also through reduction, addressed to obtain a weak head normal form of a formula or theorem. The reduction is completed in order to check if the final term is indeed a term under the unit constructor:

$$\mathsf{unit\_witness} : \forall x : \mathsf{M}\,\tau,\ \mathsf{P} \to \mathsf{is\_\ unit}\,x = \mathsf{true} \to \tau$$

9. http://cybele.gforge.inria.fr

This test achieves the reflection process where the decision procedure is the argument of monadic type in is_ unit : M $\tau \rightarrow$ bool. As established by the certifying approach to reflection, a verification of the prophecy p is always done before the reduction of the decision procedure, in our approach it is just a matter of type checking the prophecy.

Congruence problem Our example of Chapter 2 is implemented in Cybele following Corbineau, who in his master thesis [34] gives a reflexive version of the algorithm, which is purely functional and proved correct. A large part of the code is devoted to prove termination and implementing functional arrays.

We show the `Find` function implemented using Cybele using the partiality monad to avoid proving termination. Notice its dependent type and the hash-table to keep track of the representatives of elements, which is a mutable structure with a `read` and a `write` operations. The code incorporates the invariants to prove termination. The `dependentfix` operator in the monad allows non-termination. The type `S` is the signature to type the memory operations.

```
Program Definition Find hash u : M S {u': Index.t | u == u'} :=
 dependentfix (fun i => {j: Index.t | i == j}) (fun find i =>
   let! eq_proof := MHash.Read hash i in
   let (i', j, Hij) := eq_proof in
   if i == i' then (* case i = i': should always be the case *)
     if i == j then (* case i = j: we find it *) return (exist _ j Hij)
     else (* case i <> j: we have to continue from j *)
       let! r := find j in
       let (k, Hjk) := r in
       do! MHash.Write hash i (EqProof.Make (i := i) (j := k) _) in
       return (exist _ k _)
   else (* case i <> i': unexpected *) error "Find: i <> i'")
 u.
```

Figure 3.8 – Find function for congruence problem in Cybele.

The result of this function is the representative term u', equal to the input term u. The proof term is generated in the monad, so the invariants are checked dynamically. For example the comparison of i and i', that is the proof that invariant $i = i'$ holds, does not have to be statically proven. The result is used to coerce a proof of $i' = j$ to $i = j$ (done automatically by the `Program` command in our example). If the invariant check fails, we raise an exception handled by the partiality monad.

In this way we can partially specify programs. Notice that we are not forced to use partial programs, we can also use pure Coq functions leading to stronger static guarantees. This flexibility is not available in functional languages like OCaml.

To summarise, the differences between the reflection approaches discussed along this chapter reveals the pragmatic way of doing reflection under the new approach:

- Original proof by reflection approach: first state and prove a correct decision procedure and then use it.
- Certified proof by reflection approach: develop an untrusted decision procedure then proof that each certificate checks and also that the checker is correct, then finally use the procedure.
- Lightweight proof by reflection approach: provide a monadic decision procedure and use it from the very beginning, then if it works we can check the prophecy and simulate it back.

Therefore, the lightweight proof by reflection *via* the *a posteriori* simulation of effectful computations promotes a simple, easy and strong variant of the proof by reflection technique inside the Coq theorem prover.

Incrementality

Chapter 4

Optimization via data-differences

As mentioned in the introduction, it is common knowledge that an ideal cycle for program development may have a phase of optimization to impact positively either the design or the implementation of programs. Identifying the places where an optimization can be done requires a deep analysis of the problem and the phases of the development. These analyses may conduct to a more suitable abstraction of the problem and therefore a more precise specification, an enhanced solution or an implementation with better performance.

There are different techniques for optimization, some of them consider that an algorithm or a pseudo-code may have syntactic transformations leading to a better run-time, for instance the compilation optimizations. These transformations could be user-driven, be a feature of the programming paradigm or of the implementation language.

This chapter reviews some approaches to optimization, with a focus in those at computation level and development framework. We start by going over the subject of incremental computation to set up the concept of *incrementality* and then reviewing the approaches to change description towards our proposal to optimization.

4.1 Incremental computation, a state of the art

Repeating actions or tasks is an inherent behaviour of computing. The most primitive computer operations, at low-level, perform repeatedly small calculations. At high-level, to solve abstract problems, we appeal to algorithms where performing repeated computations leads to a solution. By instinct, we are always pushing the limits of computations and therefore a re-computation seems not to be helpful to improve performance.

It often happens that we cannot avoid re-computations because some parts of a program are necessarily used several times under the same or similar inputs. A very handy example showing this behaviour is the factorial function. To compute the factorial for number n, following the mathematical definition, we need to *call* the factorial function with input $n-1$ which recursively invokes the factorial function with decreased inputs until the input is 0. While computing the factorial of another number, say m, the recursive calls will at one point be the same to some of the already computed factorials obtained in the process for n.

This behaviour suggests that it is possible to take advantage of certain computations already done. One of the approaches to reuse previous results is the *memoization* technique [74, 79]. The computed outputs are stored in a (memo-)table together with the corresponding input and whenever a computation is required, the program (or a programming language facility) will search for that input in the table and retrieve the output to be reused. This approach claims that the cost of execution is reduced since a complete computation is replaced by a look-up operation over the memo-table. While the storage space will increase, the memoization invests more in optimizations for table-access and updates than in the storage cost of a possible (enormous) memo-table.

The memoization technique takes into account the reuse of results just because an input is the same. This kind of optimizations is based on modifying the functions or the given code in order to implicitly recover and reuse values. As we will see through this chapter, there are other alternatives for optimizations that are focused in the *difference* between data to take advantage of the sub-computations and partial results. Moreover, there are other optimizations whose enhancements are consequences of the features offered by the framework of development.

Considering that any change in the input will reflect a change in the output, we turn our attention to study the relationships of input changes in order to give a framework where the optimization of programs is systematic.

We are more interested in considering the optimization via *incremental computation* which roughly speaking, is a process that takes advantage of small changes in inputs to gradually compute a result, a process to avoid re-computations and reuse outputs. Then, an important issue is the way to decide if an output is reusable or not. As we will see, this decision depends on the program, its input and the output. That is, regarding how a function acts over different inputs and the obtained outputs, we can try to establish a relationship between the input differences and the output differences.

The first approaches to incremental computation were inspired in practical instances of computing and some early uses date back to the 1980's. In a bibliographic collection about incremental computation, Ramalingam and Reps [98] define the goal of incremental computation as '... to make use of the solution to one problem instance to find the solution to a *nearby* problem instance.' They organized and classified the existing approaches in a large range of computational contexts. This article is an accurate summary of what we consider a first wave of computational optimization by incrementality, in the end of 20th century. We take this classification to revisit quickly some approaches to draw a road towards our approach to incrementality.

Finite differentiating

An incipient approach to incrementality is proposed by Paige, in his Ph.D dissertation [83], and then in a joint work with Koenig [84].

The former idea is to analyse FORTRAN programs and apply formal differentiation to some parts or functions in the program. The analysis of the programs focuses on improving expensive parts which are used repeatedly. The hypotheses of this analysis states that a function, which is used many times inside a region of the program, could be pre-computed outside that region and reuse its value as many times as needed. But there is a constraint, the function cannot be pre-computed in isolation since its arguments depend on the changing context of the region. Then, an optimization suggested by a technique named 'iterator inversion' uses old values, the inputs and their differences or changes to compute a new result.

The program is transformed into an incremental version to keep track of the changes of the function inputs inside the region. In this way, a pre-computation of the function can be done outside the region and the function redefined inside the region reuse the pre-computed value and computes the small changes in order to obtain subsequent partial values. The optimization is achieved by avoiding computations of the function with an entirely new input at each call.

In subsequent work, the authors use the term *finite differencing* to name the program transformation described above. This method performs a program parsing which identifies the expensive computations and then makes a transformation to replace these parts by cheap partial sub-computations. The optimization by differentiation is achieved by the transformation: the sub-computations obtain a value gradually, starting with a given input and calculating intermediate values by adding a small increment to that input. The authors claim that they apply the technique of *strength reduction*, a compiler transformation, which is a general approach to optimization and therefore it can be applied to any programming language.

This work has served as a background for other compilation and program transformations, which are closer to incrementality (some of them are explained later).

Dependency graphs

Another early approach to incrementality for program optimisation is proposed by Reps, Teitelbaum and Demers [99]. They use attribute grammars to describe language-based editors and the files created with these editors. An attribute grammar is a formal grammar where the productions have associated values. A computation of values is done in the abstract syntax trees of the corresponding language expressions.

The authors represent each file by an attribute tree and whenever it is modified, the tree is modified by a corresponding tree-operation. The *dependency graphs* are used to represent the functional dependencies between the attributes in the tree. A full attribute tree gives a program, and then modifying a program is also carried out by modifying the tree.

The iterative process of tree evaluation and modification leads to unnecessary re-computations in the parts of the tree that do not have changed. This scenario suggests that re-computation can be performed only on the modified sub-trees and then propagate these modifications through the rest of the tree.

The graph representation enables a *dissection* of the programs where the changes under context are exposed and then the incremental computation arises naturally.

Incremental compiler

Years after, Yellin and Strom define the incremental computation as a general approach to treat problems that recompute slightly different data [117]. They took the finite differentiating approach and the idea of change propagation of the dependency graphs, commented before.

They propose a language called INC, for incremental computation, which has a compiler that takes a program and transforms it into an efficient and incremental version automatically. The compiler considers a program as a data-flow graph to express the dependencies between computations. Then, the compiler uses the finite differentiating technique to generate a new program with extra functions to recover old outputs and inputs to be reused when necessary. The authors claim that this new approach to incrementality ensures lower cost computations performed using the compiled program than the ones done with the program without the transformation.

One of the motivations and possible applications of this work is to offer an incremental programming environment.

The above efforts to deliver efficient programs belong to compiler transformations and programming tools implementing incremental solutions. While some of them point to general applications, they are conceived and used in a particular language or field. The next scope is directed to perform incremental computations in a theoretical framework.

Incremental reduction

In the beginning of the 1990's, a functional approach to incrementality was proposed by Field and Teitelbaum. They give an algorithm to perform incremental reduction in the untyped lambda calculus [39, 40].

Given a set of lambda-terms which are slightly different between them, the algorithm recognizes the common sub-terms of these terms by computing their weak-head normal forms. The terms are represented by closures to highlight the redexes in a term and a representation by abstract syntax trees is used to depict terms. A closure is a pair consisting of the body term of an abstraction and an environment which maps a variable to the term to be substituted in the reduction.

This representation of terms identifies the shared redexes and performs non-overlapping reductions. In addition to the tree representation with closures, the authors include another kind of nodes, the *fork* node denoted Δ, in the spirit of Wadsworth [114]. Then, a single tree can represent multiple terms which are similar, by means of factorizing the common sub-terms by a fork node. The incremental approach is carried out by reduction over the graph as the tree representation allows a propagation of the performed reductions that are the same and will be potentially reused.

This approach to incrementality is achieved as a feature of the framework without doing modifications to the original terms, in contrast with the above work of program transformation. Let us continue with another development offering to the user an environment capable of producing incremental programs.

Incrementalization for efficiency

The work of Liu in the beginning of the 21st century is one of the more developed and fruitful concerning incremental computation. As a starting point, she gives an accurate classification of incremental computation into three categories [68].

The first one encompasses the on-purpose *incremental algorithms* where the optimization is achieved from the conception of the solution and therefore the whole program will be efficient by design. For instance, the recursive nature of the factorial function makes implicit the division in sub-parts which can be reused.

The incremental algorithms are classified under the dynamic programming paradigm in which there is a technique to solve problems with the motto of 'divide and conquer', that is to split the original problem into sub-problems whose aim is to reduce the number of computations by reusing them. A well known technique for this is memoization, mentioned before, which stores intermediate results in order to reuse them in future computations and hence reduce the cost of computation.

The second category is called *incremental execution frameworks*, where some methods are applied to problems that can have an incremental solution, but not for deriving incremental algorithms. The incremental frameworks must provide a language for describing programs and their inputs, to include a definition of the various classes of input changes that the framework can handle and maybe a particular incremental algorithm to handle the input changes. Then, each input change is mapped to a change that the framework can manage and the incremental computation is achieved through the change management.

The third and more general category includes the *incremental-program derivation approaches* where a program is transformed to become incremental by a systematic process. This is done by program analysis and transformations, where a relation between inputs is established in order to obtain results efficiently and using some properties of the original program.

The first approaches exposed so far in this chapter, belong to the third category as well as the work of Liu and her collaborators [93, 70] which is classified by themselves within this category.

The subsequent work done by these authors focus separately into three techniques to obtain incremental programs with respect to input change operations:

- the store and reuse of returned values [66];
- the reuse of some intermediate results in combination with the returned values [67]
- and the use of auxiliary information, which is retrieved thanks to an invariant analysis of the program [69].

Later collaborations of the mentioned authors do not pursue into more sophisticated frameworks or incremental developments, but they inspired the next computational paradigm.

Self-Adjusting Computation

In the contributions mentioned up to this point, the idea of input-output and program dependencies, have been taken to a limit by a framework baptised *Self-Adjusting Computation* by Acar, after his Ph.D dissertation in 2005 [4]. He and his collaborators propose algorithms and techniques under a framework that allows writing self-adjusting programs as normal programs. This framework is available [1] as extensions of the ML and C programming languages and is applied to practical areas of computer science.

In his thesis, Acar proposes a refinement of the approaches briefly discussed above by the introduction of the self-adjusting computation paradigm as a model of computing. This paradigm is an optimization where programs automatically respond to changes in their data. He considers that 'a computation is a first-class mathematical and computational object that can be remembered, re-used and adapted to changes in the environment.'

The key idea is the connection between computations and data. As long as an input has changed it triggers a change propagation all over the computation leading to faster outputs. This idea is carefully addressed by a typed-directed discrimination of expressions, between changeable and stable data.

The components of the self-adjusting framework are: a (modal) type system where the expressions are tagged accordingly to their modifiable nature and a detailed evaluation schema using dependency graphs to ensure an improved memoization and change propagation.

As we said before, the original memoization technique uses a memo-table to store the computed outputs where the corresponding input serves as *key* to index the table. The improved memoization, named *selective memoization* [5] modifies the table indexation by using *branches* which are detailed lists of events to realize the link between data and control. Each branch records the actions performed by an input until an output is computed and then stored in the memo-table.

The framework for selective memoization is strongly directed by types which are responsible for creating the branches for indexing. Any expression whose value can change has a modal type $!\tau$, this means that in the evaluation process, this value may change and therefore leads to new outputs. For instance, the factorial function written in this framework has function type from $!\mathsf{nat}$ to nat. The input will be *explored* to keep track of itself inside the program.

An efficient representation of the computations is carried out by *traces*, which are dependency graphs created and maintained during the executions to bind parts of the program and the inputs that can modify the result. Then, the incrementality in the self-adjusting computation paradigm is achieved by the memoized traces. The function-calls and memory locations are linked by the control dependencies between the program and the inputs. Then, incrementality is completed by the change propagation over the graph that is, the edition of traces by means of enhanced algorithms for adaptivity and memoization.

1. http://www.umut-acar.org/self-adjusting-computation

A last ingredient to boost this approach to incrementality is the dynamization of algorithms for the traces or dependency graphs [6]. The dynamic dependency graph ensures an optimal representation of code and data for re-computation after input changes, that is the change propagation.

This model of computing has been largely developed, on the one side it has been used to enhance specific problems to show its applicability, for example problems in computer graphics as rendering [107, 82]. On the other side, a significant contribution to reasoning about self-adjusting programs is presented in the thesis of Ruy Ley Wild [64, 7]. This work gives high-level tools to write and reason about programs, where the theory of traces is studied in depth to identify the applicability of self-adjusting computation and formal reasoning about efficient change propagation.

4.2 *Incrementality*

The approaches presented above probably are the most prominent work around incrementality. The two last contributions, Liu's and Acar's frameworks, perform incrementality in two different ways but keep the spirit on which is based the incrementality-based optimization: the relation between programs and input changes. They also share the field of application of incrementality, both frameworks are directed to improve programs developed under an imperative paradigm. Therefore, program transformations and information flow analysis are the main techniques to improve programs.

We remark that Liu's approach can be classified as a method for *a posteriori* incrementality. In order to obtain an incremental solution, a program has to follow a mechanized process to be transformed into an incremental program. While the approach of Acar is to provide tools to the programmer in order to construct and reason about incremental programs from the beginning of a development.

We define *incrementality* as any computing task addressing program and computation optimization through a reuse of old instances of a program to compute *gradually* new instances.

We keep the following definition of incremental program and change from Liu's work for our purposes:

Definition 4.1 (Incrementality)
Given a program $\mathcal{P}$ and an operation $\oplus$, a program $\mathcal{P}'$ is called an incremental version of $\mathcal{P}$ under $\oplus$ if the latter computes $\mathcal{P}\ (x \oplus \mathsf{d})$ efficiently by making use of the computation of $\mathcal{P}\ (x)$, the intermediate results or auxiliary information of $\mathcal{P}\ (x)$.
The parameter d can be regarded as a change of the input x. Then the input change combines the old input x and a change d to form a new input $x' = x \oplus \mathsf{d}$.

Our proposal is aimed by a detailed approach to describe changes which will lead to a new approach for incrementality analysis. The notion of incrementality depends mainly on the definition of input changes, therefore we claim that the most important factor for incrementality is the change description. Hence, we go on with the state of art of incremental computation by exploring some theories of change.

4.3 Change description

The strong premise used in this part of the work is that programs are sensible to changes, we want to reflect the input changes into the program itself to compute new outputs. Then, an essential characteristic to achieve incrementality is to formalise changes or displacements[2] with an adequate degree of detail in such a way that the input change which will be mapped faithfully into an output change.

We are confident that the description of changes depends on the way the objects are defined and the denotation of differences. We are convinced that a functional paradigm provides a detailed and strong framework and therefore we adopt a paradigm where data-types are the change unit to describe displacements between objects in a collection. These claims are supported by the following techniques for data dissection.

Zipper

The functional pearl of Huet, *The Zipper* [52], is inspired in the functional representation of efficient imperative structures because of their proficient handling of data. The author explains an abstraction for tree representation with efficient edition functions. He proposes a data structure to place under focus a position in a tree in order to highlight the context and lead efficiency by local edition operations and tree navigation in constant time.

A `location` shapes the focus on one of the nodes of the tree, it is made up of a `tree` and a `path` containing the description of a tree traversal[3]:

```
type tree =
    Item of item
  | Section of tree list

type path =
    Top
  | Node of tree list * path * tree list

type location = Loc of tree * path
```

The artificial type `item`, is here a general type for any collection of objects to be manipulated. For instance, the zipper for binary trees is the following:

```
type binary_tree =
    Nil
  | Cons of binary_tree * binary_tree
```

2. From now on, the words increment and change will be used indistinctly together with the word displacement, to refer to what modifies data.
3. We follow the original presentation implemented in OCaml.

```
type binary_path =
    Top
  | Left of binary_path * binary_tree
  | Right of binary_tree * binary_path

type binary_location = Loc of binary_tree * binary_path
```

The depiction of structures into a focus and a context, started by the *zipper*, is then taken as a successful functional representation as we will see in the following works.

Locations in generic programming

The generic approach to functional programming is the abstraction of functions over data-types. Then, a generic, type-indexed or *polytypic* function is a function that can be instantiated on many data types to obtain a class of functions rather than only a definition over a single data-type [55, 50].

Hinze, Jeuring and Löh study this approach [51] and they give an implementation of Huet's zipper in Generic HASKELL. Type-indexed definitions offer to the user a more expressive framework where functions are defined by induction on the structure of types.

As these authors showed, the generic framework allows the description of a *class* of locations: the generic zipper for an arbitrary data-type $Fix\ F$[4], is a pair with an element of the data-type and the context for it. The context is a generic definition of a path in an element, this is done by means of the functor construction for data-types using the types whose *kind*[5] is $\star \to \star$:

$$\begin{array}{rcl} Id & = & \Lambda A.A \\ K\,T & = & \Lambda A.T \\ F1 + F2 & = & \Lambda A.F1\,A + F2\,A \\ F1 \times F2 & = & \Lambda A.F1\,A \times F2\,A \end{array}$$

For example, the data-type definition for lists using the above functor constructions is $List = Fix\ \ (K\,1 + Id \times List)$

The authors also define the tree navigation functions claiming to provide a 'free' movement of the focus through the element. In the following, we exhibit only the definition for locations to show the shape-directed approach to data dissection:

$$\begin{array}{rcl} Loc\langle F :: \star \to \star\rangle & :: & \star \\ Loc\langle F\rangle & = & (Fix\,F, Context\,\langle F\rangle\,(Fix\,F)) \\ & & \\ Context\,\langle F :: \star \to \star\rangle & :: & \star \to \star \\ Context\,\langle F\rangle & = & \Lambda R.Fix\,(\Lambda C.1 + Ctx\,\langle F\rangle\,C\,R) \end{array}$$

4. The formal definition of data-types is given by least-fixed points of functors, following the categorical approach to (co)induction [108].
5. A kind is *the type of a type*.

$$\begin{array}{rcl}
Ctx\langle F :: \star \to \star\rangle & :: & \star \to \star \to \star \\
Ctx\langle Id\rangle & = & \Lambda C\, R.C \\
Ctx\langle K\, 1\rangle & = & \Lambda C\, R.0 \\
Ctx\langle K\, Char\rangle & = & \Lambda C\, R.0 \\
Ctx\langle F1 + F2\rangle & = & \Lambda C\, R.Ctx\langle F1\rangle C\, R + Ctx\langle F2\rangle C\, R \\
Ctx\langle F1 \times F2\rangle & = & \Lambda C\, R.(Ctx\langle F1\rangle C\, R \times F2\, R)\ +\ (F1\, R \times Ctx\langle F2\rangle C\, R)
\end{array}$$

The functional approach to describe a focus and a context is generalized and extensively studied in the next theory. We deviate from the functional programming approach to data dissection to recall some aspects of calculus, which will serve for understand the approaches below.

Differential Calculus

Calculus as a theory of change encourages ideas and appears as a recurrent model in many areas of computer science. The differential calculus studies the rates of change and the main concept of *derivative* governs the theory. A derivative f' of a function f is obtained after a *differentiation* process, and is the measure of how a function changes as its input change, is the quotient of differentials or differences. It is also defined as the best *linear* approximation of a function in a point.

The differential of a function is computed using its derivative: $\mathsf{d}_f(x, \mathsf{d}_x) = f'(x)\,\mathsf{d}_x$ and depends on the input value x and the difference between another value $\mathsf{d}_x = \ominus x x'$, where d_x is considered small.

Containers

McBride proposes a syntactic decomposition of tree-like data-types and elaborates an influential theory for differentiating data structures [72]. This work considers data-types, more precisely equality types in ML, as polynomials. The types are denoted as functors[6] and constructed by a combination of type variables, unit, void, sum and product types, and least fixed points.

The author discovered that giving 'by hand' a definition of a *one-hole context* of a data-type, turns in the same as *deriving* (as in calculus) the polynomial or functor of the concerned data-type. The rules described to obtain a context are very similar to the rules of differentiation of calculus. The reader can notice that these are similarities are also present between this approach and the generic approach presented above.

After computing the context using the rules given by McBride [72] and replicated in Figure 4.1[7], the type of the one-hole context is a functor corresponding to the zipper of the former data-type and named from now on a *container*.

6. Recall that under the categorical approach, data-types are formalised as initial algebras, here the notation for least-fixed points is the μ-abstraction.

7. The notation $T|_{y=S}$ is for substitution in functors.

$$\begin{array}{rcl}
\partial_x x & = & 1 \\
\partial_x y & = & 0 \\
\partial_x 0 & = & 0 \\
\partial_x (S + T) & = & \partial_x S + \partial_x T \\
\partial_x 1 & = & 0 \\
\partial_x (S \times T) & = & \partial_x S \times T + S \times \partial_x T \\
\partial_x (\mu y.F) & = & \mu z.\partial_x F|_{y=\mu y.F} + \partial_y F|_{y=\mu y.F} \times z
\end{array}$$

Figure 4.1 – Derivatives of functors

Intuitively, the container of a data-type is a structure that follows the same shape of a term but has a *hole* in it[8], waiting to be filled. Each data-type and its corresponding derivative are accompanied by a *plug-in* function which computes a new object from a context and a sub-term.

For instance consider the data-type for binary trees whose initial algebra (functor) is $\mathsf{Tree} \overset{\text{def}}{=} \mu X.1 + X \times X$. Then, we obtain a tree decomposition after differentiating with respect to the free variable: $\mathsf{TreeZip} \overset{\text{def}}{=} \mu Z.1 + (\mathsf{Tree} + \mathsf{Tree}) \times Z$. The reader can notice that the container has the shape of the functor for the list data-type $\mu X.1 + A \times X$.

Moreover, we say that the focus over a binary tree is a *structural* focus since the hole in the context takes in a sub-tree. This last remark is revealed by the recursive construction $\mathsf{Tree} \times Z$.

The implementation in HASKELL of the binary trees, its one-hole contexts and the plug-in function is shown below:

```
data Tree = Empty | Node Tree Tree

data TreeCtx = Left Tree | Right Tree

type TreeZip = [TreeCtx]

plugTree :: TreeZip -> Tree  -> Tree
plug [] t = t
plug (Left t1 : z) t = Node (plugTree z t) t1
plug (Right t2 : z) t = Node t2 (plugTree z t)
```

Let us see another example, the zipper for the data-type of lists of type A:

```
data List A = Nil | Cons A (List A)

data ListCtx A = Prefix (List A) | ConsC A (ListCtx A)

type ListZip A = [ListCtx A]
```

8. As the evaluation contexts of the λ-Calculus small-step semantics.

```
plugList :: ListZip A -> List A -> List A
plug [] l = l
plug (h : z) l = h : (plug z l)
```

The derivative is ListZip $\overset{\text{def}}{=} \mu Z.\mathsf{List}A + A \times Z$. This case allows to make what we call an *atomic focus* since we can replace an element in the list due to recursive construction $A \times Z$. It also allows a *structural focus*, which is just matter of list replacement corresponding to the 'left' part of the container.

In this approach to dissect objects, we identify two sorts of focus: the structural and the atomic one, which depend on the unique context obtained after the differentiation process:

- when there are no free variables in the functor the derivation is done with respect to the bound variable under the μ-binder, this gives us a data-type for contexts through a *structural focus*;
- when the derivative is carried out with respect to one of the free variables in the functor, we obtain a data-type for an *atomic* displacement.

As well, this approach seems to be more suited to our requirements for a mechanised and fine grained change description:

- the derivative is syntax-directed and therefore easy to mechanise,
- we can distinguish the kind of focus from the type of the context,
- it is already formalised and largely developed by McBride, Altenkirch and Abbott, together with other collaborators [73, 1, 2].

However,

- this approach is not flexible despite of computing a context for each data-type,
- the unique type for the context should be interpreted in various ways, as to offer different focus (for example the container for the list data-type represents the structural and atomic focus),
- and we want to represent a focus over non standard sub-structures of an object.

To illustrate what we consider as a drawback in this theory, let us analyse the rule to derive a functor of the form $S + T$. The corresponding container describes the change in one of the two sides: $\partial_x S + \partial_x T$. Consider a displacement of an object that 'change of side', that is an object of the form $\mathsf{inl}\, s$ is displaced into an object $\mathsf{inr}\, t$. This means that a change, from S to T must be interpreted by the change $\partial_x T$ and the container does not provides explicitly a construction for this.

The reader can argue that there is no need to add a direct representation for this kind of displacement but we are looking for a fine-grained change description, that is a complete description in the sense that for any two elements x and y of type T, there exists a difference d such that $y = x \oplus \mathsf{d}$.

Derivatives of lambda-terms

The Differential Lambda Calculus, formerly developed by Ehrhard and Régnier[38] and then continued by Vaux [109], is a framework to differentiate λ-abstractions. It is rooted in denotational and linear logic semantics where program interpretation, by partial functions and power series respectively, suggests function differentiation as in calculus.

The notion of approximation of continuous functions, by a sequence of finite functions, leads to consider that this approach could help to develop a theory of changes where a program is considered as a derivable entity.

The following intuition allows us to understand the interpretation of linearity and differentials in the lambda calculus.

Consider a function $f : E \to F$ which is derivable, then its derivative f' is another function from E to the space of linear applications $\mathcal{D}_f : E \to (E \to F)$. If $e \in E$ then, $f'(x) \cdot e$ is the derivative of f at point x in the direction of e. Under the model of Ehrhard and inspired by linear logic [41], the above notion is indeed a linear application of $f'(x)$ to e.

Formally, the differential λ-calculus is a *non-deterministic* calculus for *dynamic* differentiation of functions, that is a calculus extended by means of an operator for λ-abstractions' differentiation together with a linear application of arguments.

The calculus used in this work has an evaluation under the call-by-name strategy. It includes a construction to represent the derivative of a term with respect to its i-th argument, $\mathcal{D}_i t$, and a reduction rule for differentiation which introduces the partial derivative operation ∂ over terms:

$$\mathcal{D}_1\,(\lambda x.\, t) \cdot v = \lambda x.\, \left(\frac{\partial\, t}{\partial x} \cdot v\right)$$

An important feature of this calculus is the non-determinism of reduction, introduced when deriving with respect to a variable that occurs several times in a term or when deriving a term application. The rules are given in Figure 4.2.

$$\begin{aligned}
\frac{\partial\, y}{\partial x} \cdot v &= \begin{cases} v & \text{if } x = y \\ 0 & \text{otherwise} \end{cases} \\
\frac{\partial\, \lambda y.\, s}{\partial x} \cdot v &= \lambda y.\, \left(\frac{\partial\, s}{\partial x} \cdot v\right) \\
\frac{\partial\, r\, s}{\partial x} \cdot v &= \left(\frac{\partial\, r}{\partial x} \cdot v\right) s \;+\; \left(\mathrm{D}\, r \cdot \left(\frac{\partial\, s}{\partial x} \cdot v\right)\right) s \\
\frac{\partial\, \mathrm{D}\, r \cdot s}{\partial x} \cdot v &= \mathrm{D}\left(\frac{\partial\, r}{\partial x} \cdot v\right) \cdot s \;+\; \mathrm{D}\, r \cdot \left(\frac{\partial\, s}{\partial x} \cdot v\right)
\end{aligned}$$

Figure 4.2 – Rules for partial derivatives in the differential lambda calculus

Remarks made by Vaux show the importance of linearity in this calculus. The first notion to recall is that a term is said to be linear if it uses only once an argument in reduction. This is connected with the head linear reduction of the former calculus and with the notion of derivative. The partial derivative operation over terms stands for all the terms obtained by the linear substitution. Its definition follows the rules of function derivation except for function application whose rule is similar to the rule for deriving a function product.

While the differential λ-calculus is a strong theory to study function differentiation, it is desirable to pursue a computational approach. An attempt to do this is the work of Vaux, whose extension of differentials to a non-pure functional calculus is done in the $\lambda\mu$-calculus, a combination of the above theory with the μ-calculus of Parigot [85]. However, this approach remains non-deterministic which takes us to reformulate another modification of the differential lambda calculus to design a practical programming language.

4.4 Towards Incrementality

The two main axes of the bibliography and work reviewed in this chapter are on the one side, program transformations and frameworks for incrementality, and on the other side data dissection and change description. The first developments were practical enhancements of programs and focused on compilation techniques while we choose to analyse the approaches to data description for changes in functional programming.

We want to meet both subjects by means of an input change description for an optimal propagation to realise incrementality in a *deterministic* lambda calculus.

Change description for incrementality in a functional approach

The principal motivation to use incrementality is to take advantage of outputs while keeping the connection between input changes with the output computation. This is well accomplished by the powerful framework of Acar where a tight relation of data and control provides a new model of computation.

While the machinery for change propagation is based on the strong dependencies between data and the program, there is no clear displacement treatment. A new input is never described as the result of applying an input change, but as a low-level approach by means of replacement in memory. The description of change we retrieved from the numerous articles and collaborations in the self-adjusting paradigm is a merely location update which implies a process of re-computation of outputs. The adaptative style of programming is more likely to be applied in imperative programming.

A recent collaboration of Acar with Chen, Dunfeld and Hammer [28] describes a functional application of the self-adjusting paradigm. This work gives some techniques to transform functional programs into a self-adjusting version of it.

Given a program $\mathcal{P}$, the user has to label the type of an input that may change, following the duality of changeable-stable expressions in this paradigm. Then a polymorphic translation generates a family of programs indexed by the changeable arguments in a program. This gives different choices of incremental programs, from the possible combination of changeable inputs and the change propagation in each incremental program.

Despite the functional approach, this application of the self-adjusting paradigm does not describe changes: it just make explicit the ‘changeability’ of an input in order to select the correct (incremental) implementation.

The framework of self-adjusting computation and its implementation are not suitable for reasoning about incrementality, while it is a powerful framework for the user, all the mechanisations and incremental computations are done in background, out of the reach of the user.

The other approaches reviewed in this chapter draw a refinement of ideas that converge into a computational model for incrementality, as the formal approach by the differential λ-calculus. In this calculus, an input is almost treated in the same way as in the self-adjusting approach: since this calculus is an extension of the pure λ-calculus, there is not a notion of state or memory and therefore the input changes are achieved by substitution of term displacements: $x \oplus \mathsf{d}$.

Nevertheless, both include a notion of function optimization where the input displacement is implicitly specified as change by replacement.

The work of Cai, Guiarruso, Rendel and Ostermann [25] proposes a framework for automatic *incrementalization*, ILC (incrementalizing λ-calculi), which supports static differentiation and a change theory to describe term displacements.

The framework is based on a λ-calculus *parametrized* by a user-defined `plugin` containing basic types, primitive operations and a change structure with incremental primitives. The system performs program transformations by means of the operation $\mathcal{Derive}$, which computes the derivatives of programs. The theory of changes is a dependent-type theory where any type has a set of displacements and operations to update a term and to compute a change or displacement.

While our proposal to incrementality was being studied, the ILC framework was elaborated. Both frameworks have been developed independently. We will discuss and compare ILC with our contribution in the last chapter 7. In the following we present and in the next chapter we elaborate our contribution.

Contribution

We want to propose a better approach to program optimization *via* incrementality, according to us economises computations and gives an infrastructure strong and modular enough to reason and to program.

The contribution of this work is to transform programs using derivatives to achieve incrementality in the spirit of the differential λ-calculus. We propose an approach where change description is controlled by the user who should define the granularity of how displacements are defined.

We will develop a theory to describe changes that rests in a deterministic differential lambda calculus where the programmer does not have to change the way he or she reasons when programming, that is in a functional manner.

Dependent type theory as a powerful and expressive language, enables to refine specifications, in particular this could lead to describe changes in great details. It is advantageous to elaborate a theory for changes where an object can be described by another object and a difference. This delivers a more reliable propagation of the change and an efficient computation of outputs.

We propose a *deterministic* differential λ-calculus for a general and functional approach to incrementality where functions are susceptible to propagate changes. The combination of function derivation and a specific granularity of data dissection gives a model of incrementality. We adopt a local representation of change by displacements, the data description by object dissection: an (new) object x' is split into an (old) object x and a difference d: $x \oplus_\tau \mathsf{d} = x' \quad x \ominus_\tau x' = \mathsf{d}$.

The next chapter is devoted to present a calculus for incrementality where the description of changes is not limited to small differences to achieve incremental computations but is headed by data-shape and where a program differentiation is achieved *dynamically*.

Chapter 5

A deterministic differential lambda calculus

The system called λ-diff, to be elaborated in this chapter, assembles a change theory under the name of *displaceable types*, emphasizing the importance of term displacements in order to achieve incrementality, and the function displacement treatment by differentials and partial derivatives.

The differentials and partial derivatives, as center of the λ-diff calculus, permits us to mechanically and dynamically obtain a program transformation to propagate change by means of the so called displacements.

5.1 The $\lambda-$diff calculus

We present a system that allows reasoning about differentials and formal derivatives of functions in the spirit of λ-calculus where functions are first class objects. The result of a program using a given argument can be reused to obtain a new result through the derivative of the program and a *difference* between the old and new arguments. We talk about differences or distances between two terms of same type in the λ-calculus.

The system λ-diff is defined as an extension of the simply typed λ-calculus in Church-style, with pairs and a differential operator D for functions. It is depicted in Figure 5.1.

The terms are built up from fully applied constants and typed variables together with term constructors for pairs, projections, lambda abstractions binding a typed variable to a body-term, applications and a constructor to differentiate functions. The restriction of function differentiation is achieved by the semantics as we will see later.

The second syntactic class describing values includes fully applied constants to values denoted as $\delta_c\,\overline{u}$, pairs of values and closures as values of abstractions to capture the environment.

The third class include basic types, product types and functional types.

The other syntactic classes encompass the partial functions η from variables to closed values called environments and the class of typing contexts Γ, which are collections of all distinct typed variables.

$\boxed{\text{Syntax}}$

$$\begin{array}{rcl} t,\, r,\, s,\, \mathsf{d} & ::= & c\,\bar{t} \mid x^{\tau} \mid \langle s_1,\, s_2\rangle \mid \mathsf{fst}\,t \mid \mathsf{snd}\,t \mid \lambda x^{\sigma}.\,s \mid r\,s \mid \mathrm{D}\,r \\ u,\, v,\, w & ::= & \delta_c\,\overline{u} \mid \langle w_1,\, w_2\rangle \mid (\lambda x^{\sigma}.\,s)\,[\eta] \\ \tau,\, \sigma,\, \rho & ::= & \iota \mid \sigma_1 \times \sigma_2 \mid \sigma \to \tau \end{array} \qquad \begin{array}{rcl} \eta & ::= & \bullet \mid \eta\,;\, x^{\tau} \mapsto v \\ \\ \Gamma & ::= & \varnothing \mid \Gamma,\, x^{\tau} \end{array}$$

$\boxed{\text{Dynamic Semantics}}$

$$\frac{\forall\, t_i \quad \eta \vdash t_i \Downarrow u_i}{\eta \vdash c\,\bar{t} \Downarrow \delta_c\,\overline{u}} \qquad \frac{}{\eta \vdash x \Downarrow \eta\,(x)}$$

$$\frac{\eta \vdash s_1 \Downarrow w_1 \qquad \eta \vdash s_2 \Downarrow w_2}{\eta \vdash \langle s_1,\, s_2\rangle \Downarrow \langle w_1,\, w_2\rangle} \qquad \frac{\eta \vdash t \Downarrow \langle w_1,\, w_2\rangle}{\eta \vdash \mathsf{fst}\,t \Downarrow w_1} \qquad \frac{\eta \vdash t \Downarrow \langle w_1,\, w_2\rangle}{\eta \vdash \mathsf{snd}\,t \Downarrow w_2}$$

$$\frac{}{\eta \vdash \lambda x^{\sigma}.\,s \Downarrow (\lambda x^{\sigma}.\,s)\,[\eta]} \qquad \frac{\eta \vdash r \Downarrow (\lambda x^{\sigma}.\,t)\,[\eta'] \qquad \eta \vdash s \Downarrow w \qquad \eta'\,;\, x^{\sigma} \mapsto w \vdash t \Downarrow v}{\eta \vdash r\,s \Downarrow v}$$

$$\frac{\eta \vdash r \Downarrow (\lambda x^{\sigma}.\,s)\,[\eta']}{\eta \vdash \mathrm{D}\,r \Downarrow \left(\lambda\big\langle x^{\sigma},\, d_x^{\Delta(\sigma)}\big\rangle.\, \dfrac{\partial\, s}{\partial x,\, d_x}\right)[\eta']}$$

$\boxed{\text{Static Semantics}}$

$$\frac{\forall\, t_i \quad \Gamma \vdash t_i : \sigma_i}{\Gamma \vdash c\,\bar{t} : \iota}\ \textsc{Const} \qquad \frac{x^{\tau} \in \Gamma}{\Gamma \vdash x^{\tau} : \tau}\ \textsc{Var}$$

$$\frac{\Gamma \vdash s_1 : \sigma_1 \qquad \Gamma \vdash s_2 : \sigma_2}{\Gamma \vdash \langle s_1,\, s_2\rangle : \sigma_1 \times \sigma_2}\ \textsc{Pair} \qquad \frac{\Gamma \vdash t : \sigma_1 \times \sigma_2}{\Gamma \vdash \mathsf{fst}\,t : \sigma_1}\ \textsc{Fst} \qquad \frac{\Gamma \vdash t : \sigma_1 \times \sigma_2}{\Gamma \vdash \mathsf{snd}\,t : \sigma_2}\ \textsc{Snd}$$

$$\frac{\Gamma,\, x^{\sigma} \vdash s : \tau}{\Gamma \vdash \lambda x^{\sigma}.\,s : \sigma \to \tau}\ \textsc{Abs} \qquad \frac{\Gamma \vdash r : \sigma \to \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash r\,s : \tau}\ \textsc{App}$$

$$\frac{\Gamma \vdash r : \sigma \to \tau}{\Gamma \vdash \mathrm{D}\,r : \sigma \times \Delta(\sigma) \to \Delta(\tau)}\ \textsc{Diff}$$

Figure 5.1 – λ-diff

The syntactic equality between elements is denoted by $=$. We use $\equiv$ for equivalence between terms, a formal definition is given in a later sub-section (5.3.1), meanwhile we refer to it as an equivalence relation where terms r and s are *observationally equivalent*, written $r \equiv s$, if and only if replacing occurrences of term r by term s in a given term t does not affect the observable results of the evaluation of term t. We highlight that constant functions c are partial, injective for equivalent elements and surjective.

The dynamic semantics is defined through a big-step operational semantics with environments, the evaluation process is for well-typed terms which allows us to know the types of the terms. This typing-knowledge is useful because some types will be required while differentiating as we explain later. We write $\eta \vdash t \Downarrow v$ to relate a term and a value under a given environment η and we say that term t has *converged* to value v under η.

This relation is defined inductively over terms: a fully-applied constant c is evaluated to the value of function δ_c, a variable is evaluated to its corresponding value in the environment, a pair of terms reduces to a pair of values, a term projection is evaluated to the corresponding value of a pair of values, a λ-abstraction is evaluated to a function closure and the evaluation of an application rests in the evaluation of the function-body of its left sub-term under an extended environment with the value of the right sub-term evaluation.

Finally, the evaluation of $\mathrm{D}\, r$ introduces the *partial derivative* of the body-term of the function value obtained by the reduction of term r. The corresponding value is a closure which waits for a pair argument-displacement to use it in the partial derivative. The closure is a λ-abstraction which is explicitly shown as an *uncurried* function, abstracting over a pair of variables. The partial derivative function always generates a term in system λ-diff as we will see later.

When evaluating an uncurried function, abstracting over a pair and applied to two arguments, both abstracted variables are introduced in the environment with the corresponding values. A partial evaluation is allowed after *curryfing* the function.

Before introducing the partial derivatives and more importantly the algebra for type displacements, we explain the type assignment of system λ-diff. Typing judgements relate terms to types under typing contexts: $\Gamma \vdash t : \tau$.

A fully-applied constant has basic type where each of its arguments is well-typed, a variable carries its own type then its type assignment is direct just after ensuring that the variable belongs to the typing environment. A pair a has product type constructed with the two types of its sub-terms, a projection has the corresponding type from the product type of the hypothesis and a lambda abstraction has a function-type constructed with the type of the bound variable and the type of the body sub-term. The type of an application results from the types of its sub-terms: if the left sub-term has a function type and the right sub-term agrees with the hypothesis' type of the function then, the type of the application is the conclusion type of the function type.

For term differentiation, we must ensure that the only terms allowed to be differentiated are function terms, this is the assumption in rule DIFF: if the type of the term r is $\sigma \to \tau$, then the type of $\mathrm{D}\, r$ is also a function type from a product type, of type σ together with its displacement type $\Delta(\sigma)$, to the displacement type $\Delta(\tau)$. Function $\Delta(-)$ is given later, in the general framework for displacements (see Definition 5.1).

When assigning type to values, we use the appropriate rules from the ones above using an empty context. The case of closures considers the domain of the abstracted environment as typing context to assign a type to the λ-abstraction[1] :

1. This notion of abstraction of environments, or closures, is due to Milner and Tofte [75].

$$\frac{\Gamma \vdash \lambda x^{\sigma}.\, s : \sigma \to \tau \qquad dom(\Gamma) = dom(\eta)}{\varnothing \vdash (\lambda x^{\sigma}.\, s)\,[\eta] : \sigma \to \tau}\ \textsc{Closure}$$

Partial Derivatives In Figure 5.2, the partial derivative function is defined for any well-typed term in λ-diff, it is a closed function over terms. Remark that the evaluation process is performed on well-typed terms, this allow to remember the type assignation. Also, the evaluation is the only way a partial derivative can be introduced, therefore it makes use of the variable d_x, of type $\Delta(\sigma)$ which is introduced by the differentiation of functions.

$$\begin{aligned}
\frac{\partial\, c\,\bar{t}}{\partial x, d_x} &= \dot{c}\,\bar{t}\left(\frac{\partial\, \bar{t}}{\partial x, d_x}\right)\\
\frac{\partial\, y^{\tau}}{\partial x, d_x} &= \begin{cases} d_x & \text{if } x = y\\ 0_{\sigma} & \text{otherwise}\end{cases}\\
\frac{\partial\, \langle s_1,\, s_2\rangle}{\partial x, d_x} &= \left\langle \frac{\partial\, s_1}{\partial x, d_x}, \frac{\partial\, s_2}{\partial x, d_x}\right\rangle\\
\frac{\partial\, \mathsf{fst}\, t}{\partial x, d_x} &= \mathsf{fst}\left(\frac{\partial\, t}{\partial x, d_x}\right) \qquad\qquad \frac{\partial\, \mathsf{snd}\, t}{\partial x, d_x} = \mathsf{snd}\left(\frac{\partial\, t}{\partial x, d_x}\right)\\
\frac{\partial\, \lambda y^{\sigma}.\, s}{\partial x, d_x} &= \lambda y^{\sigma}.\left(\frac{\partial\, s}{\partial x, d_x}\right)\\
\frac{\partial\, r\, s}{\partial x, d_x} &= (\mathrm{D}\, r)\left\langle s, \frac{\partial\, s}{\partial x, d_x}\right\rangle \odot_{\Delta(\tau)} \left(\frac{\partial\, r}{\partial x, d_x}\left(s \oplus_{\tau'} \frac{\partial\, s}{\partial x, d_x}\right)\right)\\
\frac{\partial\, \mathrm{D}\, r}{\partial x, d_x} &= \mathrm{D}\left(\frac{\partial\, r}{\partial x, d_x}\right)
\end{aligned}$$

Figure 5.2 – Partial derivative of a term with respect to a variable x^{σ}

For any vector of terms, we extend the definition of partial derivatives in a natural way:

$$\frac{\partial\, \bar{t}}{\partial x, d_x} = \frac{\partial\, t_0}{\partial x, d_x} \cdots \frac{\partial\, t_n}{\partial x, d_x}$$

The partial derivative of constants delegates the derivation to the corresponding constant $\dot{c}$ which in its turn makes use of each of the derivatives of the corresponding sub-terms $\bar{t}$. Derivatives of variables depend on whether or not the variable is the one we derive with respect to, in both cases we obtain a displacement. The partial derivation of pairs and projections is recursive in its sub-terms. The partial derivation of functions and function differentials are also recursive, these terms are partially derived by passing through the constructors of abstraction and differentiation.

The case of term application $r\, s$ combines two displacements of the function-term r of type σ: the first one that takes into account the displacement of its argument while the second uses a new argument obtained with the displacement of s of type τ'.

The combination of displacements by functions $\oplus$ and $\odot$ is defined in the following subsection by an algebra which handles the displacements of terms in λ-diff.

Notation We use the letter d in sans-serif font to represent terms which stand for differences whereas the variables representing differences are written $d_x^{\Delta(\tau)}$. The subscript of variables for differences is necessary to remember the displaced term. We omit the type-superscript of these variables when it can be deduced from the context (as in Figure 5.2). And since the use of typed variables could be heavy for the reader, from now on we will omit the type of variables when it can be also deduced from the context.

The product type and their corresponding terms, pairs and projections, are part of the system. In the rest, when developing recursive definitions or proofs over terms, we collapse the cases of projections into only one case for 'fst' since both cases are quite similar.

The explicit use of pairs in some λ-abstractions is to emphasize that a function expects two-arguments. This notation is aimed by the typing rule DIFF in Figure 5.1 and appears in the evaluation of function differentiation. With this notation we avoid the use of term projections when a function argument has a product type and we can access those variables in one step when handling partial derivatives.

5.2 Displaceable types

Terms of type τ can be transformed or displaced, we propose a general framework to displace terms directed by types.

Definition 5.1 (Displaceable types)
A type τ is displaceable *by $(\rho, \oplus_\tau, \ominus_\tau, 0_\tau, \odot_\tau)$ where the displacements have type ρ. There exists an identity displacement 0_τ and the following operators are provided:*

$$\begin{array}{rl} \textit{the displacement operator} & \oplus_\tau : \tau \to \rho \to \tau \\ \textit{the difference operator} & \ominus_\tau : \tau \to \tau \to \rho \\ \textit{the displacement composition} & \odot_\tau : \rho \to \rho \to \rho \end{array}$$

Function $\Delta(-)$ returns the type of displacements of τ, whenever it is displaceable.

The above operators are constants in the system. Instead of following a prefix notation, we use an infix notation for writing any displacement operator $r \oslash s$ where $\oslash \in \{\oplus, \ominus, \odot\}$. Recall that the difference operator $r \ominus s$ is intended to be read as the 'the change from s to r'.

The corresponding value of constant functions remains in infix notation:

$$\frac{\eta \vdash r \Downarrow v_r \qquad \eta \vdash s \Downarrow v_s}{\eta \vdash r \oslash s \Downarrow \delta_{\oslash}\, v_r\, v_s}$$

Figure 5.3 – Dynamic Semantics of displacement operators.

We extend in a natural way the operators applied to vectors of terms of the same length:

$$\overline{r} \oslash \overline{s} = (r_0 \oslash s_0)\ \ldots\ (r_n \oslash s_n) \qquad \delta_{\oslash}\, \overline{u}\, \overline{v} = (\delta_{\oslash}\, u_0\, v_0) \ldots (\delta_{\oslash}\, u_n\, v_n)$$

Displacement of basic types For basic types ι, we suppose that each type is defined together with its displaceable type $\Delta(\iota)$ and all the constants and primitive functions c with their corresponding derivatives $\dot{c}$. Therefore we say that constant values δ_c are given and these include the value functions $\delta_{\dot{c}}$, since they are used in the definition of partial derivatives.

For the product and function types we give their particular definitions:

Definition 5.2 (Product displacement)
Suppose two types, σ_1 and σ_2, are displaceable respectively by $(\rho_1, \oplus_{\sigma_1}, \ominus_{\sigma_1}, 0_{\sigma_1}, \odot_{\sigma_1})$ and by $(\rho_2, \oplus_{\sigma_2}, \ominus_{\sigma_2}, 0_{\sigma_2}, \odot_{\sigma_2})$.
Then the product type $\sigma_1 \times \sigma_2$ is displaceable by $(\rho_1 \times \rho_2, \oplus_{\sigma_1\times\sigma_2}, \ominus_{\sigma_1\times\sigma_2}, 0_{\sigma_1\times\sigma_2}, \odot_{\sigma_1\times\sigma_2})$ where $0_{\sigma_1\times\sigma_2} = \langle 0_{\sigma_1}, 0_{\sigma_2}\rangle$ and constant functions for operators are

$$\begin{array}{rcl}
\delta_{\oplus_{\sigma_1\times\sigma_2}}\ \langle w_1, w_2\rangle\ \langle u_1, u_2\rangle & = & \langle w_1 \oplus_{\sigma_1} u_1, w_2 \oplus_{\sigma_2} u_2\rangle \\
\delta_{\ominus_{\sigma_1\times\sigma_2}}\ \langle w_1, w_2\rangle\ \langle w_3, w_4\rangle & = & \langle w_1 \ominus_{\sigma_1} w_3, w_2 \ominus_{\sigma_2} w_4\rangle \\
\delta_{\odot_{\sigma_1\times\sigma_2}}\ \langle u_1, u_2\rangle\ \langle u_3, u_4\rangle & = & \langle u_1 \odot_{\sigma_1} u_3, u_2 \odot_{\sigma_2} u_4\rangle
\end{array}$$

Definition 5.3 (Function displacement)
Suppose a type τ is displaceable by $(\rho, \oplus_\tau, \ominus_\tau, 0_\tau, \odot_\tau)$. Then the function type $\sigma \to \tau$ is displaceable by $(\sigma \to \rho, \oplus_{\sigma\to\tau}, \ominus_{\sigma\to\tau}, 0_{\sigma\to\tau}, \odot_{\sigma\to\tau})$ where $0_{\sigma\to\tau} = \lambda x^\sigma . 0_\rho$ and constant functions of operators are defined by

$$\begin{array}{rcl}
\delta_{\oplus_{\sigma\to\tau}}\ (\lambda y^\sigma . t_1)\,[\eta_1]\ (\lambda z^\sigma . t_2)\,[\eta_2] & = & (\lambda x^\sigma . t_1 \oplus_\tau t_2)\,[\eta*] \\
\delta_{\ominus_{\sigma\to\tau}}\ (\lambda y^\sigma . t_1)\,[\eta_1]\ (\lambda z^\sigma . t_2)\,[\eta_2] & = & (\lambda x^\sigma . t_1 \ominus_\rho t_2)\,[\eta*] \\
\delta_{\odot_{\sigma\to\tau}}\ (\lambda y^\sigma . t_1)\,[\eta_1]\ (\lambda z^\sigma . t_2)\,[\eta_2] & = & (\lambda x^\sigma . t_1 \odot_\rho t_2)\,[\eta*]
\end{array}$$

where $\eta*$ is the partial function from variables to values obtained from the union of environments η_1 and η_2. There are two renamings inside terms t_1 and t_2 respectively: variables y and z are now variable x. The type of displacements for type $\sigma \to \tau$ is denoted as $\sigma \to \Delta(\tau)$.

Definition 5.4 (Displaceable properties)
Consider a type τ displaceable by $(\rho, \oplus_\tau, \ominus_\tau, 0_\tau, \odot_\tau)$. We assume that the following properties hold for t and s of type τ and displacements d and d':

$$t \ominus_\tau t \equiv 0_\tau \qquad t \oplus_\tau (s \ominus_\tau t) \equiv s \qquad t \oplus_\tau 0_\tau \equiv t$$

$$\mathsf{d} \odot_\tau 0_\tau \equiv 0_\tau \odot_\tau \mathsf{d} \equiv \mathsf{d} \qquad t \oplus_\tau (\mathsf{d} \odot_\tau \mathsf{d}') \equiv (t \oplus_\tau \mathsf{d}) \oplus_\tau \mathsf{d}'$$

5.3 Meta-theory

This subsection presents the meta-theory of λ-diff whose aim is to show the properties of the function differentials and partial derivatives. The notion of contextual equivalence needed to prove the soundness of the derivatives is included at the end of the section.

We start by giving some properties of the system related to the semantics. Partial derivation is a function over terms, we can perform substitutions over them as well as renaming and commutation with displacements.

Lemma 5.1 (Substitution and partial derivatives)
A renaming of the variable of a partial derivative is allowed:

$$\frac{\partial\, t}{\partial x, d_x}\,[x \;:=\; z]\,[d_x \;:=\; d_z] \equiv \frac{\partial\, t\,[x \;:=\; z]}{\partial z, d_z}$$

Consider a term with a partial derivative with respect to a variable x. Applying a substitution $[z \;:=\; s]$ where $z \neq x$ gives:

$$\frac{\partial\, t}{\partial x, d_x}\,[z \;:=\; s] \equiv \frac{\partial\, t\,[z \;:=\; s]}{\partial x, d_x}$$

Proof. Induction over term t using definitions in Figure 5.2.
We analyse the case of renaming a partial derivative of a variable while the rest of the cases follow immediately by induction.

- Case $t = y$ and $y \neq x$.
  On the left side $\frac{\partial\, y}{\partial x, d_x}$ is equal to 0 and therefore the substitutions does not affect the identity displacement. On the right side, the substitution $y\,[x \;:=\; z]$ does not affect variable y. Then, the partial derivation also gives the identity displacement.
- Case $t = x$.
  The first partial derivative is the displacement d_x which after substitution gives d_z. The partial derivative of $x\,[x \;:=\; z]$ is the partial derivative of z whose value is d_z.

The second statement also follows by induction. □

Lemma 5.2 (Partial derivatives equivalences)
The α-conversion of terms with partial derivatives holds:

$$\lambda\langle x^{\sigma},\, d_x^{\Delta(\sigma)}\rangle.\, \frac{\partial\, t}{\partial x, d_x} \equiv \lambda\langle z^{\sigma},\, d_z^{\Delta(\sigma)}\rangle.\, \left(\frac{\partial\, t}{\partial x, d_x}\,[x \;:=\; z]\,[d_x \;:=\; d_z]\right)$$

Moreover, the commutation between displacements holds:

$$\frac{\partial\, t_1}{\partial x, d_x} \oplus_{\Delta(\tau)} \frac{\partial\, t_2}{\partial x, d_x} \equiv \frac{\partial\, t_1 \oplus_{\tau} t_2}{\partial x, d_x}$$

where τ is the type of t_1 and $\Delta(\tau)$ is the type of t_2.

Proof. Induction over term t and term t_1 using Lemma 5.1 and definitions in Figure 5.2. □

Definition 5.5 (Consistent environment)
An environment η is Γ-consistent or consistent with respect to context Γ, written $\eta : \Gamma$, if both have the same domain and for each $\eta\ (x^\tau) = v$ we can derive $\varnothing \vdash v : \tau$ when $x^\tau \in \Gamma$.

Lemma 5.3 (Weakening of typing contexts)
If $\Gamma \vdash t : \tau$ and $x^\sigma \notin dom(\Gamma)$ then $\Gamma, x^\sigma \vdash t : \tau$.

Proof. We proceed by induction over typing derivations. □

Lemma 5.4 (Weakening of environments)
Consider an environment η which is Γ-consistent and a closed value w of type σ.
If $\eta \vdash t \Downarrow v$ and $x \notin dom(\eta)$ then $\eta\,;\, x^\sigma \mapsto w \vdash t \Downarrow v$.

Proof. Induction over the evaluation derivations. □

Lemma 5.5 (Inversion of typing in λ-diff)

- *If $\Gamma \vdash c\,\bar{t} : \tau$ then $\tau = \iota$ for a basic type and there exist σ_i such that $\Gamma \vdash t_i : \sigma_i$ for each sub-term.*
- *If $\Gamma \vdash x^\tau : \tau$ then $x^\tau \in \Gamma$.*
- *If $\Gamma \vdash \langle s_1, s_2 \rangle : \tau$ then there exist σ_1 and σ_2 such that $\tau = \sigma_1 \times \sigma_2$, $\Gamma \vdash r : \sigma_1$ and $\Gamma \vdash s : \sigma_2$.*
- *If $\Gamma \vdash \mathsf{fst}\, t : \tau$ then there exist σ_1 and σ_2 such that $\tau = \sigma_1$ and $\Gamma \vdash t : \sigma_1 \times \sigma_2$.*
- *If $\Gamma \vdash \mathsf{snd}\, t : \tau$ then there exist σ_1 and σ_2 such that $\tau = \sigma_2$ and $\Gamma \vdash t : \sigma_1 \times \sigma_2$.*
- *If $\Gamma \vdash \lambda x^\sigma .\, s : \tau$ then there exists τ' such that $\tau = \sigma \to \tau'$ and $\Gamma, x^\sigma \vdash s : \tau'$.*
- *If $\Gamma \vdash r\, s : \tau$ then there exists σ such that $\Gamma \vdash r : \sigma \to \tau$ and $\Gamma \vdash s : \sigma$.*
- *If $\Gamma \vdash \mathrm{D}\, r : \tau$ then there exists σ such that $\tau = \sigma \times \Delta(\sigma) \to \Delta(\tau)$ and $r : \sigma \to \tau$.*
- *If $\varnothing \vdash (\lambda x^\sigma .\, s)\, [\eta'] : \tau$ then there exist τ' and η' such that $\tau = \sigma \to \tau'$ and $\Gamma', x^\sigma \vdash s : \tau'$.*

Proof. Induction over the typing rules in Figure 5.1. We review the last two cases.

- Case $\Gamma \vdash \mathrm{D}\, r : \tau$.
  The last rule used to assign type τ to term $\mathrm{D}\, r$ is necessarily rule DIFF. Then, type τ has the form $\sigma \times \Delta(\sigma) \to \Delta(\tau)$ and the sub-term r is a function.
- Case $\varnothing \vdash (\lambda x^\sigma .\, s)\, [\eta'] : \tau$.
  The unique rule to assign a type to a closure is rule CLOSURE described in page 80. Therefore, taking the domain of the environment η we can obtain the typing context Γ. □

Lemma 5.6 (Uniqueness of types)
Each term of λ-diff has a unique type obtained by just one derivation using the rules given in Figure 5.1.

Proof. This proof is syntax-directed by the typing rules to derive a unique type for a given term and typing context. □

Up to now, we give some standard properties for the system λ-diff and we want to ensure that the partial derivative operation is compatible with the system, in particular with the type assignation. The following lemma shows that the partial derivatives are well typed.

Lemma 5.7 (Typing the Partial Derivative)
Consider a typing context Γ including variable x^σ and a term t whose type τ is displaceable by $(\rho,\, \oplus_\tau,\, \ominus_\tau,\, 0_\tau,\, \odot_\tau)$.
If $\Gamma',\, x^\sigma \vdash t : \tau$ then its derivative $\dfrac{\partial\, t}{\partial x, d_x}$ has type $\Delta(\tau) = \rho$ under the context $\Gamma',\, x^\sigma,\, d_x^{\Delta(\sigma)}$[2].

Proof. We proceed by induction on t of displaceable type τ. We will show that each term in Figure 5.2 has type $\Delta(\tau)$ under the typing context $\Gamma',\, x^\sigma,\, d_x^{\Delta(\sigma)}$.

- Case $t = c\,\bar{t}$
  The proof that term $\dot{c}\,\bar{t}\left(\dfrac{\partial\,\bar{t}}{\partial x, d_x}\right)$ has type $\Delta(\iota)$, since the definition of the derivative $\dot{c}$ must be $\Delta(\iota)$ as remarked in paragraph 5.2.
- Case $t = y$
  - If $y = x$ then $\sigma = \tau$ and therefore the proof is direct, since term $d_x^{\Delta(\sigma)}$ has type $\Delta(\sigma)$.
  - If $y \neq x$ then the derivative is the identity displacement 0_τ which has the desired type $\Delta(\tau)$.

For the inductive cases, consider the following induction hypotheses for terms r and s.

I.H.r
If $\Gamma \vdash r : \tau' \to \tau$ and displaceable by $(\rho',\, \oplus_{\tau'},\, \ominus_{\tau'},\, 0_{\tau'},\, \odot_{\tau'})$, then $\dfrac{\partial\, r}{\partial x, d_x}$ has type $\rho' = \tau' \to \Delta(\tau)$ **(1)** in context $\Gamma',\, x^\sigma,\, d_x^{\Delta(\sigma)}$.

I.H.s
If $\Gamma \vdash s : \tau'$ with displaceable type $(\rho',\, \oplus_{\tau'},\, \ominus_{\tau'},\, 0_{\tau'},\, \odot_{\tau'})$, then $\dfrac{\partial\, s}{\partial x, d_x}$ has type $\Delta(\tau') = \rho'$ **(2)** in context $\Gamma',\, x^\sigma,\, d_x^{\Delta(\sigma)}$.

- Case $t = \langle s_1,\, s_2\rangle$
  We want to proof that $\dfrac{\partial\,\langle s_1,\, s_2\rangle}{\partial x, d_x}$ **(3)** has type $\Delta(\sigma_1 \times \sigma_2)$.
  The typing hypothesis supposes $\Gamma,\, x^\rho \vdash \langle s_1,\, s_2\rangle : \sigma_1 \times \sigma_2$. By inversion of this judgement, we know that $\Gamma,\, x^\rho \vdash s_1 : \sigma_1$ and $\Gamma,\, x^\rho \vdash s_2 : \sigma_2$. Instantiate the induction hypothesis **I.H.s** for terms s_1 and s_2: $\dfrac{\partial\, s_1}{\partial x, d_x}$ has type $\Delta(\sigma_1)$ **(4)** and $\dfrac{\partial\, s_2}{\partial x, d_x}$ has type $\Delta(\sigma_2)$ **(5)**.
  By definition of partial derivatives in Figure 5.2, the derivative (3) is the pair constructed by $\dfrac{\partial\, s_1}{\partial x, d_x}$ and $\dfrac{\partial\, s_2}{\partial x, d_x}$. Thus, we can construct this pair using the hypotheses (4) and (5) which has type $\Delta(\sigma_1) \times \Delta(\sigma_2)$ and from Definition 5.2 is equal to $\Delta(\sigma_1 \times \sigma_2)$.

2. Variable x^σ is not forced to be a free variable of term t. The variables x^σ and $d_x^{\Delta(\sigma)}$ appear in the last or rightmost position in the typing context to emphasise that they are fresh variables. Recall the definition of rule DIFF, here the pair of variables appears *curried* in Γ.

- Case $t = \mathsf{fst}\, s$
  By inversion of the type assignation $\Gamma \vdash \mathsf{fst}\, s : \tau$, we know that $\Gamma \vdash s : \sigma_1 \times \sigma_2$ with $\tau = \sigma_1$. We want to proof that $\dfrac{\partial\ (\mathsf{fst}\, s)}{\partial x, d_x}$ has type $\Delta(\sigma_1)$ **(6)**.

  Instantiate the induction hypothesis **[I.H.s]** with $\tau' = \sigma_1 \times \sigma_2$, where $\dfrac{\partial\, s}{\partial x, d_x}$**(7)** has type $\Delta(\sigma_1) \times \Delta(\sigma_2)$. Then, the first projection of (7) is the first projection of term s which has type (6).

- Case $t = \lambda y^{\tau'}.\, s$
  We want to show that if term t has type $\tau' \to \tau$ then the abstraction $\lambda y^{\tau'}.\left(\dfrac{\partial\, s}{\partial x, d_x}\right)$ **(8)** has type $\Delta(\tau' \to \tau)$ which by Definition 5.3 is equal to $\tau' \to \Delta(\tau)$ **(9)**.
  Take the induction hypothesis **I.H.s** with $\Gamma' = \Gamma,\, y^{\sigma'},\, x^{\sigma},\, d_x^{\Delta(\sigma)}$ **(10)** and $\rho' = \Delta(\sigma)$. Construct a lambda abstraction of $y^{\sigma'}$ over term (2) from a permutation of context (10) where variable y appears in the last position, and using rule ABS to obtain term (8) of type (9).

- Case $t = r\, s$
  Consider σ as the type of $r\, s$, and by inversion of typing we know that it exists τ' such that $r : \tau' \to \tau$ and $s : \tau'$. The proof of this case is to show that the following term has type $\Delta(\sigma)$

  $$(\mathrm{D}\, r)\left\langle s,\, \frac{\partial\, s}{\partial x, d_x}\right\rangle \odot_{\Delta(\sigma)} \left(\frac{\partial\, r}{\partial x, d_x}\left(s \oplus_{\sigma'} \frac{\partial\, s}{\partial x, d_x}\right)\right)\ \textbf{(11)}$$

  Instantiate the induction hypotheses **I.H.r** and **I.H.s**, and consider the typing context $\Gamma,\, x^{\sigma},\, d_x^{\Delta(\sigma)}$. We proceed by dissecting (11) in the following sub-proofs:

  1. Show that $(\mathrm{D}\, r)\left\langle s,\, \dfrac{\partial\, s}{\partial x, d_x}\right\rangle$ has type $\Delta(\tau')$.
     Since term r has function type we can apply the typing rule DIFF of Figure 5.1 to ensure that $\mathrm{D}\, r$ has type $\tau' \times \Delta(\tau') \to \Delta(\tau)$.
     From hypothesis **I.H.s** we can derive that $\left(\dfrac{\partial\, s}{\partial x, d_x}\right) : \Delta(\tau')$ **(12)**.
     Therefore, the application of term $\mathrm{D}\, r$ to the pair constructed with term s and its derivative (12) has type $\Delta(\tau)$.
  2. Show that $\left(\dfrac{\partial\, r}{\partial x, d_x}\right)$ has type $\tau' \to \Delta(\tau)$.
     This holds by induction hypothesis **I.H.r**.
  3. Show that $\left(s \oplus_{\tau'} \dfrac{\partial\, s}{\partial x, d_x}\right)$ has type τ'.
     The displacement operator $\oplus$ given in Definition 5.1 returns a term of type τ' which is the displacement of term s by (12) of type $\Delta(\tau')$.

  Finally we use the corresponding displacement functions of the displaceable type $\Delta(\tau)$, to combine the above terms and to obtain a term of type $\Delta(\tau)$.

- Case $t = \mathrm{D}\, r$
  This case requires to prove that whenever $r : \tau' \to \tau$ then $\mathrm{D}\left(\dfrac{\partial\, r}{\partial x, d_x}\right)$ has type $\Delta(\tau \times \Delta(\tau) \to \Delta(\tau'))$.
  Take the induction hypothesis **I.H.r**. We can apply the typing rule DIFF from Figure 5.1 in order to construct a differential of type $\tau \times \Delta(\tau) \to \Delta(\Delta(\tau'))$ corresponding to term $\dfrac{\partial\, r}{\partial x, d_x}$ of type $\tau' \to \Delta(\tau)$. This is achieved by Definition 5.3. □

Lemma 5.8 (Deterministic evaluation)
Consider an environment η and a term t such that $\mathsf{FV}(t) \subseteq dom(\eta)$. *If $\eta \vdash t \Downarrow v$ and $\eta \vdash t \Downarrow v'$ then $v = v'$.*

Proof. Consider derivation $\eta \vdash t \Downarrow v$ **(1)**. Suppose that there exists another derivation $\eta \vdash t \Downarrow v'$ **(2)**, then the we want to show that $v = v'$.

The evaluation rules in Figure 5.1 are unique for each term, therefore there is always a unique choice of the last rule in the derivation (2) which is the same in (1). □

The evaluation in system λ-diff ensures the preservation of types.

Lemma 5.9 (Type preservation)
Consider a well typed term $\Gamma \vdash t : \tau$ and an environment η which is Γ-consistent. If there exists a value v such that $\eta \vdash t \Downarrow v$, then $\varnothing \vdash v : \tau$.

Proof. Suppose $\Gamma \vdash t : \tau$. The proof is done by induction on the last rule used in the derivation of $\eta \vdash t \Downarrow v$ **(1)**.
For each case consider an environment $\eta : \Gamma$ following Definition 5.5, where we assume any set of values u_i for η. The induction hypotheses ensures the property for sub-derivations in (1).

- Case $\eta \vdash c\,\bar{t} \Downarrow \delta_c\, \overline{w}$
  A constant is reduced to a value $\delta_c\, \overline{w}$. Then we want to proof that the above value has type ι.
  The function value δ_c is given and maps values to basic values, then the type of this function verifies that the type of its codomain is ι.
- Case $\eta \vdash x^\tau \Downarrow \eta\ (x^\tau)$
  Following Definition 5.5, each value in environment η has the same type as the variable it is linked to. Therefore, $\eta\ (x) = u_i$ has type τ.
- Case $\eta \vdash \langle s_1,\, s_2\rangle \Downarrow \langle w_1,\, w_2\rangle$
  In order to proof that $\langle w_1,\, w_2\rangle$ has type $\sigma_1 \times \sigma_2$ we consider the induction hypotheses for each s_i with $i \in \{1, 2\}$: $\eta \vdash s_i \Downarrow w_i$ **(2)** with $\Gamma \vdash s_i : \sigma_i$ and $\varnothing \vdash w_i : \sigma_i$. Using rule PAIR with the hypotheses (2), we can assign the product type to the pair of values.
- Case $\eta \vdash \mathsf{fst}\, t \Downarrow w_1$
  The induction hypothesis states that if $\Gamma \vdash t : \sigma_1 \times \sigma_2$ and $\eta \vdash t \Downarrow v$ then $\varnothing \vdash v : \sigma_1 \times \sigma_2$ **(3)** where $v = \langle w_1,\, w_2\rangle$.
  Therefore, we can apply the fst projection to value v and conclude that $\mathsf{fst}\, v$ has type σ_1 by using the typing rule FST.

- Case $\eta \vdash \lambda x^\sigma . s \Downarrow (\lambda x^\sigma . s)\,[\eta]$
  Assigning a type to a closure is assigning the type to the lambda abstraction under a typing context which is the domain of the environment η. This is exactly the induction hypothesis.
- Case $\eta \vdash r\,s \Downarrow v$
  This case has three induction hypotheses:
  - If $\eta \vdash r \Downarrow w_1$ and $\Gamma \vdash r : \sigma \to \tau$ where $w_1 = (\lambda x^\sigma . t)\,[\eta']$ **(4)**, then $\varnothing \vdash w_1 : \sigma \to \tau$.
  - If $\eta \vdash s \Downarrow w_2$ and $\Gamma \vdash s : \sigma$ then $\varnothing \vdash w_2 : \sigma$.
  - If $\eta' ; x^\sigma \mapsto w_2 \vdash t \Downarrow v$ where t is the body-term obtained in (4), and its corresponding type judgement is $\Gamma', x^\sigma \vdash t : \tau$ where Γ' is the domain of η' as defined in rule CLOSURE on page 5.1. Then $\varnothing \vdash v : \tau$ **(5)**.

  The proof is to show that value v from $\eta \vdash r\,s \Downarrow v$ has type τ under the empty typing context. But this is the third induction hypothesis (5).
- Case $\eta \vdash \mathrm{D}\,r \Downarrow w$
  We want to proof that $\Gamma \vdash w : \sigma \times \Delta(\sigma) \to \Delta(\tau)$ where w is the closure $\left(\lambda \left\langle x^\sigma, d_x^{\Delta(\sigma)}\right\rangle . \dfrac{\partial\, t}{\partial x, d_x}\right)[\eta']$ **(6)**. The typing hypothesis is $\Gamma \vdash \mathrm{D}\,r : \sigma \times \Delta(\sigma) \to \Delta(\tau)$.

  The induction hypothesis about $\eta \vdash r \Downarrow (\lambda x^\sigma . t)\,[\eta']$, ensures that the closure $(\lambda x^\sigma . t)\,[\eta']$ **(7)** has function type $\sigma \to \tau$ **(8)** under an empty context.
  The inversion Lemma 5.5 in (8) gives a type assignation for the body-term t using context Γ' such that $dom(\Gamma') = dom(\eta')$ and hence $\Gamma', x^\sigma \vdash t : \tau$ **(9)**.
  The partial derivative of t has type $\Delta(\tau)$ as consequence of Lemma 5.7 applied to (9) and therefore $\Gamma', x^\sigma, d_x^{\Delta(\sigma)} \vdash \dfrac{\partial\, t}{\partial x, d_x} : \Delta(\tau)$ **(10)**.
  After abstracting twice in (10) and *uncurrying* the variables x and d_x, we have the following: $\Gamma' \vdash \lambda\left\langle x^\sigma, d_x^{\Delta(\sigma)}\right\rangle . \dfrac{\partial\, t}{\partial x, d_x} : \sigma \times \Delta(\sigma) \to \Delta(\tau)$, which finally can be used to assign a type to the closure (6) keeping the same environment η'. $\square$

5.3.1 Equivalence

In earliest definitions we used an equivalence between terms without a formal definition, now we will state a definition for it which allows us to support the definitions and to prove theorems in this chapter.

The equivalence of terms or programs is based on the idea of getting equivalent outputs when the equivalent terms are used interchangeably inside term-contexts. These contexts follow the same structure of terms but they have a missing sub-term, they have a *hole*. The name of *contextual equivalence* to refer term equivalence, is originated by term-contexts [94]. But this notion is *a.k.a observational equivalence*, that is when filling a term-context with any of the pretended equivalent terms, gives the same observation.

Since we use a big-step semantics, we use an observational approach to term equivalence via *extensionality*. Let us take an example to show the desired behaviour.

Example (Equivalence). *Consider the following evaluations* $\eta \vdash (\lambda y^{\tau}.\, x)\ [x := \lambda z^{\sigma}.\, z] \Downarrow v$ *and* $\eta\,;\, x^{\rho} \mapsto (\lambda z^{\sigma}.\, z)\,[\eta] \vdash \lambda y^{\tau}.\, x \Downarrow v'$*. Ideally, we may think of* $v = v'$ *but this is not the case, since* $v = (\lambda y.\, \lambda z.\, z)\,[\eta]$ *and* $v' = (\lambda y.\, x)\,[\eta\eta\,;\, x \mapsto (\lambda z.\, z)\,[\eta]]$*, are different syntactic values. Nonetheless, these terms together with their evaluation contexts give the impression of returning the same answer or the same observable results when applied to the same values.*

Therefore, instead of using syntactic equality we can relate both terms by an observational equivalence. The equivalence of terms is achieved by a ground definition for equivalent values which is mutually defined with equivalent environments since they are needed for the equivalence between function closures in the same spirit of the terms in the example. The reader may notice the *one-way* relation for equivalence in both definitions below, Definition 5.7 is not symmetric.

Definition 5.6 (Environment and value equivalence)
Given a typing context Γ*, two* Γ*-consistent environments* η_1 *and* η_2 *are equivalent if and only if their domains are the same and for each variable* x*,* $\eta_1\ (x) \equiv_v \eta_2\ (x)$ *holds.*
The relation $\equiv_v$ *stands for the relation between values of the same type:*

1. adequacy *for fully applied constants:*
   values obtained from the same constant functions are compared syntactically

$$\delta_c\ \overline{w} \equiv_v \delta_c\ \overline{u} \ \text{ if and only if } \ \delta_c\ \overline{w} = \delta_c\ \overline{u}$$

2. *function closures are compared* extensionally*:*

$$(\lambda x^{\sigma}.\, t)\ [\eta] \equiv_v (\lambda y^{\sigma}.\, t')\ [\eta'] \text{ if and only if for each } \varnothing \vdash s : \sigma,$$

$$\text{if } \ \eta \vdash (\lambda x^{\sigma}.\, t)\ s \Downarrow v \ \text{ then } \eta' \vdash (\lambda y^{\tau}.\, t')\ s \Downarrow v' \text{ and } \ v \equiv_v v'$$

3. *pairs of values are compared* point-wise*:*

$$\langle w_1,\, w_2 \rangle \equiv_v \langle u_1,\, u_2 \rangle \ \text{ if and only if } \ w_1 \equiv_v u_1 \text{ and } \ w_2 \equiv_v u_2$$

Definition 5.7 (Contextual equivalence)
Given a context Γ*, we say that two terms* r *and* s *are related or equivalent,* $t \equiv s$*, when they have the same type under* Γ *and for a given consistent environment* η*, if there exists a value* u *for* $\eta \vdash t \Downarrow u$ *then there exists* v *such that* $\eta \vdash s \Downarrow v$ *and* $u \equiv_v v$*.*

5.4 Soundness

The goal of this section is to demonstrate that the behaviour of computations in presence of differentials and derivatives, that is an evaluation of a term *incrementally*, gives the same result as if it is evaluated in the 'traditional' way.

The notion of result equality is treated by the observational equivalence defined in 5.3.1. As well, the proofs in this section are strongly based on the properties stated about our change theory defined in subsections 5.2 and 5.3, for instance the equivalences between term displacements (see the properties in definition 5.1). Then differential approach to incremental computation is sound and allows to reason under displacements.

A refinement of the evaluation of displacements of constants has to be settle down for later arguments.

Definition 5.8 (Displacements of constants)
Consider a basic type ι displaceable by $(\rho,\, \oplus_\iota,\, \ominus_\iota,\, \mathsf{0}_\iota,\, \odot_\iota)$ and a fully-applied constant c such that $\Gamma \vdash c\,\overline{t} : \iota$.
Given a Γ-consistent environment η, if the evaluation of the constant converges $\eta \vdash c\,\overline{t} \Downarrow \delta_c\,\overline{w}$, then the value is unique.
Moreover, suppose that there is a value displacement say $\delta_{\dot{c}}\,\overline{w}\,\overline{v}$, then the constant values commute under the displacements:

$$\delta_{\oplus_\iota}\;(\delta_c\,\overline{w})\;(\delta_{\dot{c}}\;\overline{w}\,\overline{v}) = \delta_c\;(\delta_{\oplus_{\overline{\sigma}}}\,\overline{w}\,\overline{v})$$

where σ_i is the corresponding type of the i-th argument of c.

The proof of the principal Theorem 5.14 shows the equivalence between a displaced term by its partial derivative and the same term under the substitution of the new term $(x \oplus_\rho d_x)$. This theorem makes use of each of the lemmas in this section, which state equivalence properties of displacements $\oplus$.

First, Lemma 5.10, states the equivalence between a displacement of a given function and its application to another term. Lemma 5.11, is about the distribution of the differential $\mathrm{D}\,t$ over a term and its displacement. This lemma leads to achieve the demonstration of the immediate Lemma 5.12. This lemma shows that, the evaluation of a term displaced by the change in variable x and the evaluation of same term under an environment where the variable is updated by a displaced value, behaves the same.

Lemma 5.10 (Function displacement I)
If $\Gamma \vdash t : \sigma \to \tau$ and $\Gamma \vdash s : \sigma$ for any terms t and s and given a displacement of the former then the following property holds:

$$(t \oplus_{\sigma\to\tau} \mathsf{d})\; s \equiv t\, s \oplus_\sigma \mathsf{d}\, s$$

Proof. We want to prove that both sides of the equivalence converge respectively to a value and those values are equivalent $v \equiv v'$.
Consider an environment η which is Γ-compatible.
Suppose that the evaluation of the left side $\eta \vdash (t \oplus_{\sigma\to\tau} \mathsf{d})\; s \Downarrow v$ **(1)**, converges with the following sub-derivations:

- $\eta\;\vdash\; t \oplus_{\sigma\to\tau} \mathsf{d} \;\Downarrow\; (\lambda x^\sigma.\, t_1' \oplus_\tau t_2')\,[\eta*]$ **(2)** which follows from rule in Figure 5.3 instantiated for $\oplus_{\sigma\to\tau}$ and from Definition 5.3 where $\eta \;\vdash\; t \;\Downarrow\; (\lambda y^\sigma.\, t_1)\,[\eta_1]$ **(3)** and $\eta \vdash \mathsf{d} \Downarrow (\lambda z^\sigma.\, t_2)\,[\eta_2]$ **(4)**. The environment $\eta*$ is the union of environments η_1 and η_2, sub-terms t_1' and t_2' in (2) have respectively the renaming of y and z by a fresh variable x;

- $\eta \vdash s \Downarrow w$ **(5)** for some value w
- and $\eta * \,;\, x^{\sigma} \mapsto w \vdash t_1' \oplus_{\tau} t_2' \Downarrow v$ which requests two sub-derivations, one for each argument for the displacement operator: $\eta * \,;\, x^{\sigma} \mapsto w \vdash t_1' \Downarrow v_1$ **(6)** and $\eta * \,;\, x^{\sigma} \mapsto w \vdash t_2' \Downarrow v_2$ **(7)**, to give the left-value v in (1) is $\delta_{\oplus_\tau} v_1 \, v_2$.

On the right side, the evaluation $\eta \vdash t\, s \oplus_{\tau} \mathsf{d}\, s \Downarrow v'$ converges when the following sub-derivations converge: $\eta \vdash t\, s \Downarrow v_3$ **(8)** and $\eta \vdash \mathsf{d}\, s \Downarrow v_4$ **(9)**.
Since evaluation is deterministic, we use derivations above to obtain $v' = \delta_{\oplus_\tau} v_3\, v_4$:

- derivations (3) and (5) to obtain (8): $\eta_1 \,;\, y \mapsto w \vdash t_1 \Downarrow v_3$ **(10)**
- derivations (4) and (5) to obtain (9): $\eta_2 \,;\, z \mapsto w \vdash t_2 \Downarrow v_4$ **(11)**

Now, by renaming the variable y by x in derivation (10), following Definition 5.6, we obtain $\eta_1 \,;\, y \mapsto w \vdash t_1' \Downarrow v_3$. And by weakening the environment to add elements in environment η_2, using Lemma 5.4, we obtain the evaluation (6) and therefore we can conclude that $v_3 = v_1$ since evaluation is deterministic.
Both actions are possible since variable x does not appear in (10), and the weakening can be carried out since the domain of η_2 contains the free variables of $\lambda z^{\sigma}.\, t_2$ and not the free variables of t_1.
The same reasoning can be applied to convert (11) into (7) which implies that $v_4 = v_2$. And therefore conclude that both sides converge to the same value. □

Lemma 5.11 (Function differential distribution over displacements)
Consider a term of function type, $\Gamma \vdash t : \sigma \to \tau$ then the following holds:

$$\mathrm{D}\, t \oplus \mathrm{D} \left(\frac{\partial\, t}{\partial x, d_x} \right) \equiv \mathrm{D} \left(t \oplus \frac{\partial\, t}{\partial x, d_x} \right)$$

Proof. Let the evaluation of term t converge, $\eta \vdash t \Downarrow (\lambda y^{\sigma}.\, s)\,[\eta']$ **(1)**. Then, we want to prove that $v = v'$ from $\eta \vdash \mathrm{D}\, t \oplus \mathrm{D} \left(\frac{\partial\, t}{\partial x, d_x} \right) \Downarrow v$ **(2)** and $\eta \vdash \mathrm{D} \left(t \oplus \frac{\partial\, t}{\partial x, d_x} \right) \Downarrow v'$ **(3)**.
From (2), the value v is a closure following Definition 5.3 for function displacement:

$$\left(\lambda \left\langle z_1^{\sigma},\, d_{z_1}^{\Delta(\sigma)} \right\rangle .\, s' \oplus_{\Delta(\tau)} r' \right) [\eta *]$$

where $s' = \left(\frac{\partial\, s}{\partial y, d_y} \right) [y := z_1]\,[d_y := d_{z_1}]$ and $r' = \left(\frac{\partial\, r}{\partial y_1, d_{y_1}} \right) [y_1 := z_1]\,[d_{y_1} := d_{z_1}]$ which are respectively derived from:

- $\eta \vdash \mathrm{D}\, t \Downarrow \left(\lambda \left\langle y^{\sigma},\, d_y^{\Delta(\sigma)} \right\rangle .\, \frac{\partial\, s}{\partial y, d_y} \right) [\eta_1]$
- $\eta \vdash \mathrm{D} \left(\frac{\partial\, t}{\partial x, d_x} \right) \Downarrow \left(\lambda \left\langle y_1^{\sigma},\, d_{y_1}^{\Delta(\tau)} \right\rangle .\, \frac{\partial\, r}{\partial y_1, d_{y_1}} \right) [\eta_2]$ with $\eta \vdash \frac{\partial\, t}{\partial x, d_x} \Downarrow (\lambda y_1^{\sigma}.\, r)\,[\eta_2]$ **(4)**

On the other hand, the evaluation to get v' (3) uses the evaluation of the function displacement: $\eta \vdash t \oplus \frac{\partial\, t}{\partial x, d_x} \Downarrow w$ where $w = (\lambda z_2^{\sigma}.\, s\,[y := z_2] \oplus_{\tau} r\,[y_1 := z_2])\,[\eta *]$ using previous evaluations (1) and (4) again on Definition 5.3 for function displacement.

Then, by the evaluation rule for differentials we obtain the value

$$v' = \left(\lambda \left\langle z_2^\sigma,\, d_{z_2}^{\Delta(\sigma)} \right\rangle . \frac{\partial\, s\,[y \ := \ z_2] \oplus_\tau r\,[y_1 \ := \ z_2]}{\partial z_2, d_{z_2}} \right) [\eta *]$$

The definition of observational equivalence for function closures takes any term of the corresponding argument-type to be applied in order to compare the results of functions. We proceed to apply a value $\langle u,\, \mathsf{d} \rangle$ to closures v and v' and test the equivalence of their results.

Given that the all these terms are values, we analyse the following evaluations:

$$\eta * \,;\, z_1 \mapsto u \,;\, d_{z_1} \mapsto \mathsf{d} \vdash \left(\frac{\partial\, s\,[y \ := \ z_1]\,[d_y \ := \ d_{z_1}]}{\partial z_1, d_{z_1}} \right) \oplus_{\Delta(\tau)} \left(\frac{\partial\, r\,[y_1 \ := \ z_1]\,[d_{y_1} \ := \ d_{z_1}]}{\partial z_1, d_{z_1}} \right) \Downarrow w$$

$$\eta * \,;\, z_2 \mapsto u \,;\, d_{z_2} \mapsto \mathsf{d} \vdash \frac{\partial\, s\,[y \ := \ z_2] \oplus_\tau r\,[y_1 \ := \ z_2]}{\partial z_2, d_{z_2}} \Downarrow w'$$

Both judgements are α-equivalent and following Lemma 5.2 the above displacements using $\oplus_{\Delta(\tau)}$ and $\oplus_\tau$, are equivalent. Therefore, by determinism and since both environments are equivalent, the values w and w' are equivalent. □

Lemma 5.12 (Displacements in Environments) *Consider a well-typed term t under context Γ', x^ρ and a displacement d corresponding to the free variable x. If η' is Γ'-compatible, then the values of the following evaluations are equivalent:*

$$\eta' \,;\, x \mapsto v \,;\, d_x \mapsto \mathsf{d} \vdash t \oplus_\tau \frac{\partial\, t}{\partial x, d_x} \Downarrow w \qquad\qquad \eta' \,;\, x \mapsto \left(\delta_{\oplus_\rho}\, v\, \mathsf{d}\right) \vdash t \Downarrow w'$$

Proof. Induction over t. Let η an extended environment: $\eta = \eta' \,;\, x \mapsto v \,;\, d_x \mapsto \mathsf{d}$.

- Case $t = y$

  Suppose $\eta \vdash y \oplus_\tau \dfrac{\partial\, y}{\partial x, d_x} \Downarrow w$ **(1)** and $\eta' \,;\, x \mapsto \left(\delta_{\oplus_\rho}\, v\, \mathsf{d}\right) \vdash y \Downarrow w'$ **(2)** then we want to proof that $w \equiv w'$. This case gives two sub-cases:

  1. From hypothesis (1) where $y = x$ we know that $\eta \vdash x \oplus_\rho d_x \Downarrow w$ where value w is the displacement of $\eta\ (x)$ by $\eta\ (d_x)$.

     On the other side, evaluation (2) returns the value of variable x which is the same term above.

  2. Now take the hypothesis (1) where $y \neq x$. Then the identity displacement and the evaluation is $\eta \vdash y \oplus_\tau 0_\tau \Downarrow w$. Which following Definition 5.4, is simplified into term $y \oplus_\tau 0_\tau \equiv y$ and therefore $w = \eta'\ (y)$.

     On the other side, the evaluation gives the value of variable y in environment $\eta' \,;\, x \mapsto \left(\delta_{\oplus_\rho}\, v\, \mathsf{d}\right)$ which is the same value as before.

  Therefore $w = w'$.

- Case $t = c\,\bar{t}$

  This case remains abstract.

I.H.r

Consider $\Gamma \vdash r : \sigma' \to \sigma$, and η consistent with $\Gamma, d_x^{\Delta(\rho)}$. If $\eta \vdash r \oplus_{\sigma' \to \sigma} \dfrac{\partial\, r}{\partial x, d_x} \Downarrow u$ and $\eta' \,;\, x \mapsto \big(\delta_{\oplus_\rho}\, v\, \mathsf{d}\big) \vdash r \Downarrow u'$ then $u \equiv u'$.

I.H.s

Consider $\Gamma \vdash s : \sigma$, then for η compatible with $\Gamma, d_x^{\Delta(\rho)}$. If $\eta \vdash s \oplus_\sigma \dfrac{\partial\, s}{\partial x, d_x} \Downarrow u$ and $\eta' \,;\, x \mapsto \big(\delta_{\oplus_\rho}\, v\, \mathsf{d}\big) \vdash s \Downarrow w'$ then $u \equiv u'$.

- Case $t = \langle s_1,\, s_2 \rangle$
Consider the induction hypotheses for sub-terms s_i as instances of **I.H.s** above. Suppose that $\eta \vdash \langle s_1,\, s_2 \rangle \oplus_{\sigma_1 \times \sigma_2} \dfrac{\partial\, \langle s_1,\, s_2 \rangle}{\partial x, d_x} \Downarrow w$ **(3)** and $\eta' \,;\, x \mapsto \big(\delta_{\oplus_\rho}\, v\, \mathsf{d}\big) \vdash \langle s_1,\, s_2 \rangle \Downarrow w'$ **(4)**, then the proof is to show that $w \equiv w'$.
Following the definition of partial derivatives and the definition of pair displacements, the term in (3) converges into $\langle v_1 \oplus_{\sigma_1} u_1,\, v_2 \oplus_{\sigma_2} u_2 \rangle$ where each sub-term s_i converges to v_i from the induction hypothesis and the derivative $\dfrac{\partial\, s_i}{\partial x, d_x}$ converges to u_i.
The pair in (4) converges to value $\langle w_1',\, w_2' \rangle$ where w_i' is the evaluation of the terms s_i under environment $\eta' \,;\, x \mapsto \big(\delta_{\oplus_\rho}\, v\, \mathsf{d}\big)$. We can conclude by induction hypothesis that the value pairs are equivalent since the comparison is made point wise.

- Case $t = \mathsf{fst}\, s$
Take an instance of the induction hypothesis **I.H.s** as follows:
if $\eta \vdash s \oplus_{\sigma_1 \times \sigma_2} \dfrac{\partial\, s}{\partial x, d_x} \Downarrow \langle w_1,\, w_2 \rangle$ and $\eta' \,;\, x \mapsto \big(\delta_{\oplus_\rho}\, v\, \mathsf{d}\big) \vdash s \Downarrow \langle w_1',\, w_2' \rangle$ then $w_i \equiv w_i'$ for each $i \in \{1, 2\}$.
Using the term projections, fst and snd we can conclude that $w_i \equiv w_i'$.

- Case $t = \lambda y^{\sigma'} .\, s$
Consider the induction hypothesis **I.H.s**, to proof that:
if $\eta \vdash \lambda y^{\sigma'} .\, s \oplus_{\sigma' \to \sigma} \lambda y^{\sigma'} .\, \dfrac{\partial\, s}{\partial x, d_x} \Downarrow w$ **(5)** and $\eta' \,;\, x \mapsto \big(\delta_{\oplus_\rho}\, v\, \mathsf{d}\big) \vdash \lambda y^{\sigma'} .\, s \Downarrow w'$ **(6)** then $w \equiv w'$.
Following Definition 5.3, the value w in (5) is equal to $\left(\lambda y^{\sigma'} .\, s \oplus_{\sigma'} \dfrac{\partial\, s}{\partial x, d_x}\right) [\eta]$ **(7)**.
We use Definition 5.6: take any term r of type σ' to be applied to both (7) and (6). Suppose also that this term converges to value u.
Then, by induction hypothesis we can conclude that the evaluation of the body-term in (7) under environment $\eta \,;\, y \mapsto u$ and term s under environment $\eta' \,;\, x \mapsto \big(\delta_{\oplus_\rho}\, v\, \mathsf{d}\big) \,;\, y \mapsto u$ are equivalent.

- Case $t = r\, s$
Consider the induction hypotheses for terms r and s with environment $\eta = \eta' \,;\, x \mapsto v \,;\, d_x \mapsto \mathsf{d}$.
The left evaluation is $\eta \vdash r\, s \oplus_\tau \dfrac{\partial\, r\, s}{\partial x, d_x} \Downarrow w$ **(8)**, which after applying the definition

of partial derivatives gives:
$\eta \vdash r\, s \oplus_\tau \left((\mathrm{D}\, r) \left\langle s, \dfrac{\partial\, s}{\partial x, d_x} \right\rangle \odot_{\Delta(\tau)} \dfrac{\partial\, r}{\partial x, d_x} \left(s \oplus_\sigma \dfrac{\partial\, s}{\partial x, d_x} \right) \right) \Downarrow w.$
And after using the equivalence for displacement composition in Definition 5.4:
$\eta \vdash \left(r\, s \oplus_\tau (\mathrm{D}\, r) \left\langle s, \dfrac{\partial\, s}{\partial x, d_x} \right\rangle \right) \oplus_\tau \left(\dfrac{\partial\, r}{\partial x, d_x} \left(s \oplus_\sigma \dfrac{\partial\, s}{\partial x, d_x} \right) \right) \Downarrow w$ **(9)**.

Recall that $\eta \vdash s \Downarrow v_s$ and $\eta \vdash r \Downarrow (\lambda z^\sigma . t')\,[\eta'']$ where η'' is an extension of η.
Then, the above evaluation (9) has the following sub-evaluations:

(a) the term $r\, s$ converges to value u which is obtained from the evaluation of t' under $\eta''\, ;\, x \mapsto v\, ;\, d_x \mapsto \mathsf{d}\, ;\, z \mapsto v_s$;

(b) $\mathrm{D}\, r$ converges into function $\left(\lambda \left\langle z^\sigma, d_z^{\Delta(\sigma)} \right\rangle . \dfrac{\partial\, t'}{\partial z, d_z} \right) [\eta'']$;

(c) the pair $\left\langle s, \dfrac{\partial\, s}{\partial x, d_x} \right\rangle$ converges to value $\langle v_s, \mathsf{d}_s \rangle$ under η ;

(d) the partial derivative $\dfrac{\partial\, r}{\partial x, d_x}$ reduces to $\left(\lambda z^\sigma . \dfrac{\partial\, t'}{\partial x, d_x} \right) [\eta'']$ and

(e) $s \oplus_\sigma \dfrac{\partial\, s}{\partial x, d_x}$ reduces to $\delta_{\oplus_\sigma}\, v_s\, \mathsf{d}_s$.

The value w in (9) is $\delta_{\oplus_\tau}\, w_1\, w_2$ and the respective values w_i are obtained from the displacement and the differential described as follows:
the displacement obtained from term in (b) is the evaluation of $\dfrac{\partial\, t'}{\partial z, d_z}$ under environment $\eta''\, ;\, z \mapsto v_s\, ;\, d_z \mapsto \mathsf{d}_s$, and the differential is obtained from term in (d) by the evaluation of $\dfrac{\partial\, t'}{\partial x, d_x}$ under environment $\eta''\, ;\, z \mapsto \delta_{\oplus_\sigma}\, v_s\, \mathsf{d}_s$.

The proof is to show that value w is equivalent to the value obtained in evaluation $\eta'\, ;\, x \mapsto \left(\delta_{\oplus_\rho}\, v\, \mathsf{d}\right) \vdash r\, s \Downarrow w'$, which is indeed the evaluation of term t' under context $\eta''\, ;\, x \mapsto \left(\delta_{\oplus_\rho}\, v\, \mathsf{d}\right)\, ;\, z \mapsto v_s$.
We can conclude that this holds by induction hypothesis over term t': the evaluation of $\left(t' \oplus_\tau \dfrac{\partial\, t'}{\partial z, d_z} \right) \oplus_\tau \dfrac{\partial\, t'}{\partial x, d_x}$ under environment $\eta''\, ;\, x \mapsto v\, ;\, d_x \mapsto \mathsf{d}\, ;\, z \mapsto v_s\, ;\, d_z \mapsto \mathsf{d}_s$.

- Case $t = \mathrm{D}\, r$
The induction hypothesis is for term r of function type $\sigma \to \tau$ where $\eta \vdash r \Downarrow (\lambda z^\sigma . t')\,[\eta']$.
The proof is to show that $w \equiv w'$ from evaluations: $\eta \vdash \mathrm{D}\, r \oplus_{\tau'} \dfrac{\partial\, \mathrm{D}\, r}{\partial x, d_x} \Downarrow w$ **(10)** where $\tau' = \sigma \times \Delta(\sigma) \to \Delta(\tau)$ and $\eta'\, ;\, x \mapsto \left(\delta_{\oplus_\rho}\, v\, \mathsf{d}\right) \vdash (\mathrm{D}\, r) \Downarrow w'$ **(11)**.
From evaluation (10) and following definition in Figure 5.2 the term to be evaluated is $\mathrm{D}\, r \oplus_{\tau'} \mathrm{D}\, \dfrac{\partial\, r}{\partial x, d_x}$ which by Lemma 5.11 is equivalent to term $\mathrm{D} \left(r \oplus_{\tau'} \dfrac{\partial\, r}{\partial x, d_x} \right)$.
Then, we can apply the induction hypothesis to conclude. □

Lemma 5.13 (Function displacement II)
Consider a program r applied to an argument $\Gamma \vdash r\, s : \tau$ and a displacement of argument s say d, then the following holds:

$$r\, s \oplus_\tau (\mathrm{D}\, r)\ \langle s,\, \mathsf{d}\rangle \equiv r\ (s \oplus_\sigma \mathsf{d})$$

Proof. Suppose that $\Gamma \vdash r\, s : \tau$ and let $\eta : \Gamma$ be a consistent environment. Then we proceed by evaluating both terms, if $\eta \vdash r\, s \oplus_\tau (\mathrm{D}\, r)\ \langle s,\, \mathsf{d}\rangle \Downarrow v$ **(1)** and $\eta \vdash r\ (s \oplus_\sigma \mathsf{d}) \Downarrow v'$ **(2)** then $v \equiv v'$ must hold.

The evaluation of (1) is achieved by the displacement evaluation rule in Figure 5.3 for $\oplus$, where its sub-terms converge to values as follows:

- $r\, s$ converges to v_1 where
  $\eta \vdash r \Downarrow (\lambda z^\sigma.\, t)\, [\eta']$ **(3)**, $\eta \vdash s \Downarrow u$ **(4)** and $\eta'\, ;\, z \mapsto u \vdash t \Downarrow v_1$ **(5)**.
- $(\mathrm{D}\, r)\ \langle s,\, \mathsf{d}\rangle$ converges to v_2 through the following:
  $\eta \vdash \mathrm{D}\, r \Downarrow \left(\lambda \left\langle z^\sigma,\, d_z^{\Delta(\sigma)}\right\rangle.\, \dfrac{\partial\, t}{\partial z, d_z}\right) [\eta']$ is the evaluation of the differential using the rule defined in Figure 5.1 with (3) as hypothesis; the evaluation of the pair $\langle s,\, \mathsf{d}\rangle$ converges into $\langle u,\, \mathsf{d}\rangle$ using (4) and given that d is also a value, and finally the value v_2 is obtained by the evaluation $\eta'\, ;\, z \mapsto u\, ;\, d_z \mapsto \mathsf{d} \vdash \dfrac{\partial\, t}{\partial z, d_z} \Downarrow v_2$.

The evaluation of the second term (2), has the following evaluations as hypotheses:

- the evaluation of term r, which is the same evaluation as before (3);
- the evaluation of the displacement of argument s, $\eta \vdash s \oplus_\sigma \mathsf{d} \Downarrow v_3$ where $v_3 = \delta_{\oplus_\sigma}\, u\, \mathsf{d}$
- and $\eta'\, ;\, z \mapsto v_3 \vdash t \Downarrow v'$.

Then, we have for first term (1), the value $v = \delta_{\oplus_\tau}\, v_1 v_2$. For the second (2), the value $\eta'\, ;\, z \mapsto v_3 \vdash t \Downarrow v'$. The final step of the proof is to demonstrate that values v and v' are equivalent. But this is ensured by Lemma 5.12. □

Theorem 5.14 (Soundness of partial derivatives)
Consider a typing context $\Gamma = \Gamma',\, d_x^{\Delta(\rho)}$ where Γ' is the context in $\Gamma' \vdash t : \tau$. Then the following holds:

$$t \oplus_\tau \frac{\partial\, t}{\partial x, d_x} \equiv t\, [x \ := \ x \oplus_\rho d_x]$$

Proof. Induction on derivation $\Gamma' \vdash t : \tau$. Let η a Γ-consistent environment, we show that if $\eta \vdash t \oplus_\tau \dfrac{\partial\, t}{\partial x, d_x} \Downarrow v$ and $\eta \vdash t\, [x \ := \ x \oplus_\rho d_x] \Downarrow v'$ then $v \equiv v'$.

- Case $t = y$
  Suppose $\eta \vdash y \oplus_\tau \dfrac{\partial\, y}{\partial x, d_x} \Downarrow v$ **(1)** and $\eta \vdash y\, [x \ := \ x \oplus_\rho d_x] \Downarrow v'$ **(2)** then we want to prove that $v \equiv v'$. This case has two sub-cases:

1. Instantiate the hypothesis (1) where $y = x$, that is $\eta \vdash x \oplus_\rho d_x \Downarrow v$.
   Then, value v is the outcome of function $\delta_{\oplus_\rho}$ applied to values obtained from: $\eta \vdash x \Downarrow \eta\ (x)$ and $\eta \vdash d_x \Downarrow \eta\ (d_x)$.
   On the other side, evaluation (2) gives the same displacement operation after applying the substitution, since executions are deterministic (see Lemma 5.8) we have $v = v'$.
2. The hypothesis (1), where $y \neq x$, takes the identity displacement and the evaluation becomes $\eta \vdash y \oplus_\tau 0_\tau \Downarrow v$. By Definition 5.4, we can simplify the term $y \oplus_\tau 0_\tau \equiv y$ and obtain $v = \eta\ (y)$.
   On the other side, the substitution does not affect y leading to the evaluation of variable y, $\eta \vdash y \Downarrow \eta\ (y)$ and $v' = \eta\ (y)$.

Therefore $v \equiv v'$ since in both sub-cases we obtained $v = v'$.

- Case $t = c\,\overline{t}$
  Suppose $\eta \vdash c\,\overline{t} \oplus_\iota \dfrac{\partial\, c\,\overline{t}}{\partial x, d_x} \Downarrow v$ **(3)** and $\eta \vdash (c\,\overline{t})\ [x\ :=\ x \oplus_\rho d_x] \Downarrow v'$ **(4)** to prove that $v \equiv v'$.
  The induction hypothesis for this case holds for each sub-term t_i: if $\Gamma \vdash t_i : \sigma_i$, then for η compatible with Γ, if $\eta \vdash t_i \oplus_{\sigma_i} \dfrac{\partial\, t_i}{\partial x, d_x} \Downarrow u_i$ **(5)** where $\eta \vdash t_i \Downarrow v_i$ and $\eta \vdash \dfrac{\partial\, t_i}{\partial x, d_x} \Downarrow w_i$, and $\eta \vdash t_i\,[x\ :=\ x \oplus_\rho d_x] \Downarrow u_i'$ **(6)** then $u_i \equiv u_i'$.
  Hypothesis (3) requires the evaluations of $c\,\overline{t}$ and the partial derivative $\dfrac{\partial\, c\,\overline{t}}{\partial x, d_x}$. They reduce respectively to $\delta_c\,\overline{u}$ **(7)** and $\delta_{\dot{c}}\ \overline{u}\,\overline{w}$ **(8)** following Definitions in 5.1 and 5.2.
  In evaluation (4), the substitution takes place inside the sub-terms. Then, the evaluation has the sub-derivations of the induction hypothesis (6).
  Finally, the proof is to show the equivalence between values $\delta_c\,\overline{u} \oplus_\iota \delta_{\dot{c}}\ \overline{u}\,\overline{w}$ and $\delta_c\,\overline{u'}$. For this we appeal to Definition 5.8.

I.H.r

Consider $\Gamma \vdash r : \sigma' \to \sigma$, and η consistent with Γ, $d_x^{\Delta(\rho)}$. If $\eta \vdash r \oplus_{\sigma' \to \sigma} \dfrac{\partial\, r}{\partial x, d_x} \Downarrow w$ and $\eta \vdash r\,[x\ :=\ x \oplus_\rho d_x] \Downarrow w'$ then $w \equiv w'$.

I.H.s

Consider $\Gamma \vdash s : \sigma$, then for η compatible with Γ, $d_x^{\Delta(\rho)}$. If $\eta \vdash s \oplus_\sigma \dfrac{\partial\, s}{\partial x, d_x} \Downarrow w$ and $\eta \vdash s\,[x\ :=\ x \oplus_\rho d_x] \Downarrow w'$ then $w \equiv w'$.

- Case $t = \langle s_1,\ s_2 \rangle$
  The induction hypotheses for sub-terms s_i are two instances of **I.H.s** above. Suppose that $\eta \vdash \langle s_1,\ s_2 \rangle \oplus_{\sigma_1 \times \sigma_2} \dfrac{\partial\, \langle s_1,\ s_2 \rangle}{\partial x, d_x} \Downarrow v$ **(9)** and $\eta \vdash \langle s_1,\ s_2 \rangle\ [x\ :=\ x \oplus_\rho d_x] \Downarrow v'$ **(10)**, then the proof is to show that $v \equiv v'$.
  From hypothesis (9) and following the definition of partial derivative for pairs in Figure 5.2 and Definition 5.2, the value v is the pair $\langle v_1 \oplus_{\sigma_1} u_1,\ v_2 \oplus_{\sigma_2} u_2 \rangle$ where s_i

converges to v_i from the induction hypothesis and the derivative $\dfrac{\partial\, s_i}{\partial x,\, d_x}$ converges to u_i.

The second hypothesis (10) converges to the pair $\langle w_1',\, w_2'\rangle$ where w_i' is the evaluation of $s_i\,[x\ :=\ x \oplus_\rho d_x]$ from induction hypotheses.

The values $v_i {\oplus_{\sigma_i}} u_i$ and w_i' are equivalent, also by induction hypothesis, and therefore the pairs are equivalent since the comparison is made point wise.

- Case $t = \mathsf{fst}\, s$
  Consider the induction hypothesis **I.H.s** where $\sigma = \sigma_1 \times \sigma_2$ that is:
  if $\eta \vdash s \oplus_{\sigma_1\times\sigma_2} \dfrac{\partial\, s}{\partial x,\, d_x} \Downarrow \langle w_1,\, w_2\rangle$ and $\eta \vdash s\,[x\ :=\ x \oplus_\rho d_x] \Downarrow \langle w_1',\, w_2'\rangle$ then $w_i \equiv w_i'$ for each $i \in \{1, 2\}$.

  The above evaluations can be used as hypotheses in the rules for evaluate a term projection, fst and snd and therefore $w_i \equiv w_i'$.

- Case $t = \lambda y^{\sigma'}.\, s$
  Consider again the induction hypothesis **I.H.s**, we want to prove that given $\eta \vdash \lambda y^{\sigma'}.\, s \oplus_{\sigma'\to\sigma} \lambda y^{\sigma'}.\, \dfrac{\partial\, s}{\partial x,\, d_x} \Downarrow v$ **(11)** and $\eta \vdash \left(\lambda y^{\sigma'}.\, s\right)\,[x\ :=\ x \oplus_\rho d_x] \Downarrow v'$ **(12)** then $v \equiv v'$.

  Following Definition 5.3, the value v in (11) is equal to $\left(\lambda y^{\sigma'}.\, s \oplus_{\sigma'} \dfrac{\partial\, s}{\partial x,\, d_x}\right)[\eta]$ **(13)**, and in evaluation (12), the value v' is equal to $\left(\lambda y^{\sigma'}.\, (s\,[x\ :=\ x \oplus_\rho d_x])\right)[\eta]$ **(14)**.

  In order to prove that $v \equiv v'$, let us use Definition 5.6: take any term r of type σ' to be applied to both (13) and (14). Suppose also that this term converges to value u.

  This changes the proof, instead of proving that $v \equiv v'$ from (11) and (12), we will proof that $w \equiv w'$ which are the values obtained from evaluating the applications (13) and (14) to r, respectively.

  From the induction hypothesis, we know that the body-terms of the above closures behave equivalently and therefore the applications will result in equivalent values, $w \equiv w'$, since the evaluation environments are the same, η extended with $y \mapsto u$.

- Case $t = r\, s$
  Consider the induction hypotheses for terms r and s.

  The left evaluation is: $\eta \vdash r\, s \oplus_\tau \dfrac{\partial\, r\, s}{\partial x,\, d_x} \Downarrow v$ **(15)** where $v = \delta_{\oplus_\tau}\, w_1\, w_2$ and the respective values w_i are obtained from the evaluations $r\, s$ and $\dfrac{\partial\, r\, s}{\partial x,\, d_x}$.

  On the other hand : $\eta \vdash (r\, s)\,[x\ :=\ x \oplus_\rho d_x] \Downarrow v'$ relies on derivations $\eta \vdash r\,[x\ :=\ x \oplus_\rho d_x] \Downarrow v_1'$ and $\eta \vdash s\,[x\ :=\ x \oplus_\rho d_x] \Downarrow v_2'$ following the distribution of the substitution operation. By induction hypotheses, these terms are equivalent to those in derivations: $\eta \vdash r \oplus_{\sigma\to\tau} \dfrac{\partial\, r}{\partial x,\, d_x} \Downarrow v_1$ **(16)** and $\eta \vdash s \oplus_\sigma \dfrac{\partial\, s}{\partial x,\, d_x} \Downarrow v_2$ **(17)**.

  From now we will reason through equivalences to proof that the terms $(r\ s){\oplus_\tau}\dfrac{\partial\, r\, s}{\partial x,\, d_x}$ and $(r\ s)\ [x\ :=\ x \oplus_\rho d_x]$ are equivalent.

Consider the terms in (16) and (17), following Lemma 5.10 we can distribute the application of them into: $\left(r\left(s \oplus_\sigma \dfrac{\partial\, s}{\partial x, d_x}\right)\right) \oplus_\tau \left(\dfrac{\partial\, r}{\partial x, d_x}\left(s \oplus_\sigma \dfrac{\partial\, s}{\partial x, d_x}\right)\right)$.
Then, by Lemma 5.13 we can use the equivalence on the left side to obtain:

$$\left(r\, s \oplus_\tau (\mathrm{D}\, r)\left\langle s, \frac{\partial\, s}{\partial x, d_x}\right\rangle\right) \oplus_\tau \left(\frac{\partial\, r}{\partial x, d_x}\left(s \oplus_\sigma \frac{\partial\, s}{\partial x, d_x}\right)\right)$$

Finally, through definition 5.4 we can obtain a term that composes the displacements:

$$r\, s \oplus_\tau \left((\mathrm{D}\, r)\left\langle s, \frac{\partial\, s}{\partial x, d_x}\right\rangle \odot_{\Delta(\tau)} \frac{\partial\, r}{\partial x, d_x}\left(s \oplus_\sigma \frac{\partial\, s}{\partial x, d_x}\right)\right)$$

This term gives exactly the term (15) after applying the corresponding definition in Figure 5.2.

- Case $t = \mathrm{D}\, r$
  Consider the induction hypothesis for term r of type $\sigma \to \tau$ where $\eta \vdash r \Downarrow (\lambda z^\sigma.\, t')\,[\eta']$.
  Then we want to show that $v \equiv v'$ from evaluations: $\eta \vdash \mathrm{D}\, r \oplus_{\tau'} \dfrac{\partial\, \mathrm{D}\, r}{\partial x, d_x} \Downarrow v$ **(18)** where $\tau' = \sigma \times \Delta(\sigma) \to \Delta(\tau)$ and $\eta \vdash (\mathrm{D}\, r)\ [x := x \oplus_\rho d_x] \Downarrow v'$ **(19)**.
  On hypothesis (19) we can internalise the substitution and apply the induction hypothesis: $\eta \vdash \mathrm{D}\left(r \oplus \dfrac{\partial\, r}{\partial x, d_x}\right) \Downarrow v'$ **(20)**
  Then, in order to proof that (18) and (20) converge to equivalent values we use Lemma 5.11. □

Chapter Conclusions

We finish this chapter by recalling the incremental approach by differentials: the system λ-diff permits the incremental evaluation of a program, the user only has to define the types and its displacements. However, this pure calculus is not expressive enough to develop useful programs. In the next chapter, we discuss two extensions to provide a more practical system in which one can study and reason about incremental computation.

Chapter 6

Recursion and Data-Types in the deterministic differential lambda calculus

The system λ-diff includes the basic constructions to compute functions and its derivatives. Now, we give some extensions to the system where the most representative are the expressions for fixed-points and data-types all along with their derivatives to analyse the incrementality over them.

6.1 Differentiation of multiple-argument functions

A generalisation of the system is letting the user to choose the variable with respect to a function of multiple arguments will be derived.

Definition 6.1 (Differentiation with respect to the i-th argument)
Let the term t be evaluated into an abstraction of n-arguments, that is $(\lambda x_0^{\sigma_0} \ldots x_n^{\sigma_n}.s)\,[\eta']$. Then, the differentiation of t with respect to argument i is obtained while evaluating the term $\mathrm{D}_i\, t$ and gives the following closure:

$$\left(\lambda x_0^{\sigma_0} \ldots \left\langle x_i^{\sigma_i},\, d_{x_i}^{\Delta(\sigma_i)} \right\rangle \ldots\, x_n^{\sigma_n}.\frac{\partial\, s}{\partial x_i, d_{x_i}}\right)[\eta']$$

The dynamic semantics respects the position in which the displacement d_{x_i} is introduced, that is by adding a pair of the variable x_i and the displacement to the abstraction of the partial derivative, the abstractions that are before or after variable x_i remain unchanged.

All definitions and properties in the previous chapter are likely to be preserved under the exchange of symbol D by the symbol D_i to point out the position of the concerned variable.

6.2 Fixed-Points in λ-diff

Recursive functions are a powerful construction in programming languages, in the following we add to the differential λ-calculus a construction allowing the definition of general fixed-points. This is achieved by the term $\mathsf{fun} f.\lambda x^{\rho}.\, s$.

The above term is a function in which the variable f may or may not appear free in term s. Therefore, recursive functions and named lambda-abstractions are represented by the function expression fun. This expression could be a fixed point or just a function as before: if f appears free in s it means that the expression is a fixed point, while if it does not appear it means that it is a standard λ-abstraction and therefore the function-term is a definition under name f.

The definitions for λ-abstractions given in the system λ-diff, in Figures 5.1 and 5.2, are supplemented by the definitions in Figure 6.1.

Syntax

$$\begin{array}{rcl} t,\, r,\, s & ::= & \cdots \mid \mathsf{fun} f.\lambda x^{\sigma}.\, s \\ v,\, w & ::= & \cdots \mid (\mathsf{fun} f.\lambda x^{\sigma}.\, s)\, [\eta] \end{array}$$

Dynamic Semantics

$$\frac{}{\eta \vdash \mathsf{fun} f.\lambda x^{\sigma}.\, s \Downarrow (\mathsf{fun} f.\lambda x^{\sigma}.\, s)\, [\eta]}$$

$$\frac{\eta \vdash r \Downarrow (\mathsf{fun} f.\lambda x^{\sigma}.\, t)\, [\eta'] \qquad \eta \vdash s \Downarrow w \qquad \eta' \,;\, f \mapsto (\mathsf{fun} f.\lambda x^{\sigma}.\, t)\, [\eta'] \,;\, x \mapsto w \vdash t \Downarrow v}{\eta \vdash r\, s \Downarrow v}$$

$$\frac{\eta \vdash r \Downarrow (\mathsf{fun} f.\lambda x^{\sigma}.\, s)\, [\eta']}{\eta \vdash \mathrm{D}\, r \Downarrow \left(\mathsf{fun} f.\lambda \left\langle x^{\sigma},\, d_x^{\Delta(\sigma)} \right\rangle.\, \dfrac{\partial\, s}{\partial x, d_x}\right) [\eta']}$$

Static Semantics

$$\frac{\Gamma,\, f^{\sigma \to \tau},\, x^{\sigma} \vdash t : \tau}{\Gamma \vdash \mathsf{fun} f.\lambda x^{\sigma}.\, t : \sigma \to \tau}\ \textsc{Fun}$$

Partial derivative

$$\frac{\partial\, \mathsf{fun} f.\lambda y^{\sigma}.\, t}{\partial x, d_x} \;=\; \mathsf{fun} f.\lambda y^{\sigma}.\, \frac{\partial\, t}{\partial x, d_x}$$

Figure 6.1 – λ-diff: Functions.

The new construction, for named and recursive functions, uses the same notion as before for the semantics as λ-abstractions. The evaluation of a function leads to a closure, the term application has a second rule where the first term has to be a named function and then the evaluation of the body-term of the function is performed using an extended environment binding the name f with the function and the variable x with the value of the second term.

The differentiation has also a second rule when the term to be differentiated is a named function. This rule accomplishes the differentiation in the same way as for λ-abstractions: the differential of a function will wait for the pair argument–displacement. When calculating the partial derivative, the action to be performed is also the same as before, the derivative operation passes through the function-name and the argument abstraction.

Meta-theory for functions

The following lemmas related to typing properties of functions are proven immediately from rules in Figure 6.1.

Lemma 6.1 (Inversion)
If $\Gamma \vdash \mathsf{fun} f.\lambda x^{\sigma}.t : \tau$ *then there exists* σ' *such that* $\tau = \sigma \to \sigma'$ *and* $\Gamma,\, f^{\sigma\to\sigma'},\, x^{\sigma} \vdash t : \sigma'$.

Lemma 6.2 (Typing the Partial Derivative)
Consider a typing context Γ *and a term* t *whose type* τ *is displaceable by* $(\rho, \oplus_\tau, \ominus_\tau, 0_\tau, \odot_\tau)$. *If* $\Gamma,\, y^{\sigma'} \vdash \mathsf{fun} f.\lambda x^{\sigma}.t : \sigma \to \tau$ *then its derivative has type* $\sigma \to \Delta(\tau)$ *under the context* $\Gamma,\, y^{\sigma'},\, d_y^{\Delta(\sigma')}$.

Lemma 6.3 (Type Preservation)
Consider a well typed function $\Gamma \vdash \mathsf{fun} f.\lambda x^{\sigma}.t : \sigma \to \tau$ *and an environment* η *which is* Γ*-consistent. If there exists a value* v *such that* $\eta \vdash \mathsf{fun} f.\lambda x^{\sigma}.t \Downarrow v$, *then* $\varnothing \vdash v : \sigma \to \tau$.

This extension has more interesting applications when used over data-types like natural numbers, lists or binary trees. We propose another extension of λ-diff with algebraic data-types.

6.3 Structural displacements over algebraic data-types

In this section we define data-types in the differential lambda calculus using the following notation:

Definition 6.2 (Algebraic data-type)
Let $\mathfrak{K}$ *be an enumerable set of data-constructor identifiers, where* K *ranges over it. An algebraic data-type* $\mathbb{I}$ *is defined by an equation of the form:*

$$\mathbb{I} \overset{def}{=} \sum_{i\in\mathcal{I}} \mathsf{K}_i : \sigma_i \qquad \text{where } \sigma_i = \sigma_{i,0} \to \cdots \to \sigma_{i,n_i}$$

$\mathcal{I}$ is a finite set of indices for the number of constructors and K_i's with $i \in \mathcal{I}$ are pairwise distinct with zero or more arguments [1].

This way of describing data-types resembles the functional definitions like `data` in HASKELL or the `Inductive` type in COQ. For instance, the unit type whose unique constructor is $\star$, is defined as $\mathbb{I}_1 \overset{\text{def}}{=} \star$. We write $\tau + \sigma$ as a short cut for the *sum*-data-type $\mathbb{I}_{\tau+\sigma}$, whose constructors are L (left) and R (right), it is defined by:

$$\mathbb{I}_{\tau+\sigma} \overset{\text{def}}{=} \mathsf{L} : \tau \; + \; \mathsf{R} : \sigma$$

Other examples are the data-types for booleans and natural numbers:

$$\mathsf{bool} \overset{\text{def}}{=} \mathsf{true} \; + \; \mathsf{false} \qquad\qquad \mathsf{nat} \overset{\text{def}}{=} \mathsf{zero} \; + \; \mathsf{suc} \; : \mathsf{nat}$$

The significant additions to system λ-diff are depicted in Figure 6.2.

The abstract base types ι will now be instantiated by the types $\mathbb{I}$. Some constants c become the constructors K_i and the general elimination form of any data-type is the case operator which is labelled with the corresponding inductive type. Recall that data-types used here define inductive objects *constructed* using the data identifiers K_i's and *destructed* by an instance of the elimination term case.

A constructor K of an inductive type $\mathbb{I}$ has as many arguments as defined in the corresponding type definition. We call a *branch* the function associated (by $\Rightarrow$) to a constructor identifier in the elimination expression $\mathsf{case}_{\mathbb{I}}$, it also has as many arguments as the constructor has. The $\mathsf{case}_{\mathbb{I}}$ elimination contain the corresponding branch for each constructor of the type $\mathbb{I}$. A branch or a sequence of branches are defined by the syntactic category b where the *head*-branch is the left-most branch in a sequence.

The evaluation of a case-term considers the value of the term to be destructed. Then, the evaluation continues by comparing the value against the sequence $\{b_0 \mid \cdots \mid b_m\}$, the comparison is denoted by $\rightsquigarrow$.

When comparing a term (a full applied constructor) with a branch, the semantic rules check whether or not the constructor identifier is the same as the constructor of the *head*-branch in the sequence. If they are the same, the evaluation proceed under the extended environment using the values of the value-constructor and taking the body-term of the associated function. If the identifiers are different, then the comparison continues with the rest of the sequence. We ensure that this exhaustive process succeeds since the static semantics takes the elimination expression containing all cases for the corresponding constructors of the inductive type $\mathbb{I}$. Therefore at some point both constructors, the one of the value and the one in head-position, will coincide making progress in the evaluation.

The definition of partial derivatives is extended with the cases for constructors and elimination forms. The definitions given in Figure 6.2 are dedicated to support our particular choice to define data-type displacements in the section that follows (see Definition 6.3).

1. Note that the equation could be recursive, *i.e.*, $\sigma_{i,k}$ may be some $\mathbb{I}$. If K_i has zero arguments, then it does not have an associated type σ_i.

Syntax

$$\begin{array}{rcl}
t,\, r,\, s & ::= & x^{\tau} \mid \mathsf{K}_i \mid \mathsf{case}_{\mathbb{I}}\, t \text{ of } \{b\} \mid \langle s_1,\, s_2\rangle \mid \mathsf{fst}\, t \mid \mathsf{snd}\, t \\
 & & \mid \lambda x^{\sigma}.\, s \mid \mathsf{fun} f.\lambda x^{\sigma}.\, s \mid r\, s \mid \mathrm{D}\, t \\
b & ::= & \mathsf{K}_i \Rightarrow \lambda y_0^{\sigma_{i,0}}, \ldots, y_{n_i}^{\sigma_{i,n_i}}.\, r \mid \;\; \mathsf{K}_i \Rightarrow \lambda y_0^{\sigma_{i,0}}, \ldots, y_{n_i}^{\sigma_{i,n_i}}.\, r | b \\
v,\, w & ::= & \mathsf{K}_i\, \overline{w} \mid (\lambda x^{\sigma}.\, s)\, [\eta] \mid (\mathsf{fun} f.\lambda x^{\sigma}.\, t)\, [\eta] \\
\tau,\, \sigma,\, \rho & ::= & \mathbb{I} \mid \sigma_1 \times \sigma_2 \mid \sigma \to \tau
\end{array}$$

Dynamic Semantics

$$\frac{\forall\, s_{i,j}, \qquad \eta \vdash s_{i,j} \Downarrow w_j}{\eta \vdash \mathsf{K}_i\, \overline{s} \Downarrow \mathsf{K}_i\, \overline{w}} \qquad \frac{\eta \vdash t \Downarrow w \qquad\qquad \eta \vdash \{b_0|\, \cdots\, |b_m\} \rightsquigarrow w \Downarrow v}{\eta \vdash \mathsf{case}_{\mathbb{I}}\, t \text{ of } \{b_0|\, \cdots\, |b_m\} \Downarrow v}$$

$$\frac{\eta; y_0^{\sigma_{i,0}} \mapsto w_0; \ldots; y_{n_i}^{\sigma_{i,n_i}} \mapsto w_{n_i} \vdash r_i \Downarrow v}{\eta \vdash \{\mathsf{K}_i \Rightarrow \lambda y_0^{\sigma_{i,0}}, \ldots, y_{n_i}^{\sigma_{i,n_i}}.\, r_i | b\} \rightsquigarrow \mathsf{K}_i\, \overline{w} \Downarrow v}$$

$$\frac{\eta \vdash \{b\} \rightsquigarrow \mathsf{K}_\ell\, \overline{w} \Downarrow v}{\eta \vdash \{\mathsf{K}_i \Rightarrow \lambda y_0^{\sigma_{i,0}}, \ldots, y_{n_i}^{\sigma_{i,n_i}}.\, r | b\} \rightsquigarrow \mathsf{K}_\ell\, \overline{w} \Downarrow v \quad \textit{where } \; \mathsf{K}_i \neq \mathsf{K}_\ell}$$

Static Semantics

$$\frac{\forall\, i \in \mathcal{I},\; j \in \{0, n_i\}, \quad \Gamma \vdash \mathsf{K} : \sigma_{i,0} \to \cdots \to \sigma_{i,n_i} \qquad \forall\, s_{i,j},\; \Gamma \vdash s_{i,j} : \sigma_{i,j}}{\Gamma \vdash \mathsf{K}\, \overline{s} : \mathbb{I}}\ \textsc{Constr}$$

$$\frac{\Gamma \vdash \mathsf{K}_i : \sigma_{i,0} \to \cdots \to \sigma_{i,n_i} \qquad \Gamma \vdash \lambda y_0^{\sigma_{i,0}}, \ldots, y_{n_i}^{\sigma_{i,n_i}}.\, r_i : \tau}{\Gamma \vdash \mathsf{K}_i \Rightarrow \lambda y_0^{\sigma_{i,0}}, \ldots, y_{n_i}^{\sigma_{i,n_i}}.\, r_i : \overline{\sigma_i} \to \tau}\ \textsc{Branch}$$

$$\frac{\Gamma \vdash t : \mathbb{I} \qquad \forall\, b_i,\;\; \Gamma \vdash b_i : \overline{\sigma_i} \to \tau}{\Gamma \vdash \mathsf{case}_{\mathbb{I}}\, t \text{ of } \{b_0|\, \cdots\, |b_m\} : \tau}\ \textsc{Case}$$

Figure 6.2 – λ-diff: Algebraic types

When deriving an inductive constructor, an auxiliary function $\mathsf{cong}_{\mathsf{K}}$ computes the displacement of the element by internalising the change of x on each sub-expression s_i. The function cong is a congruence operation whose aim is to generate a displacement over the same constructor:

$$\mathsf{K}_i\, \overline{s_i} \oplus_{\mathbb{I}} \mathsf{cong}_{\mathsf{K}_i}\, \left(\overline{\mathsf{d}}\right) = \mathsf{K}_i\, \overline{(s_i \oplus_{\sigma} \mathsf{d})}$$

The partial derivative of the elimination expression $\mathsf{case}_{\mathbb{I}}$ makes use of a given derivative $\mathsf{dcase}_{\mathbb{I}}$. This auxiliary function is expected to be given, in the sense that the user provides the derivatives of the primitive operations, included the elimination forms. The arguments of this function are described in the next definition of the displacement.

$$\begin{aligned}
\frac{\partial\, \mathsf{K}_i\, \overline{s_i}}{\partial x, d_x} &= \mathsf{cong}_{\mathsf{K}_i} \left(\frac{\partial\, \overline{s_i}}{\partial x, d_x} \right) \\
\frac{\partial\, \mathsf{case}_{\mathbb{I}}\, t \text{ of } \{b_0 \mid \cdots \mid b_m\}}{\partial x, d_x} &= \mathsf{dcase}_{\mathbb{I}}\, t \left(\frac{\partial\, t}{\partial x, d_x} \right) \left(\lambda \overline{y}^{\overline{\sigma_0}} .\, r_0\, \varphi \right) \left(\lambda \overline{y}^{\overline{\sigma_0}} .\, \lambda \overline{d_y}^{\overline{\Delta(\sigma_0)}} .\, \frac{\partial\, r_0}{\partial x, d_x} \right) \\
&\qquad \cdots \\
&\qquad \left(\lambda \overline{y}^{\overline{\sigma_m}} .\, r_m\, \varphi \right) \left(\lambda \overline{y}^{\overline{\sigma_m}} .\, \lambda \overline{d_y}^{\overline{\Delta(\sigma_m)}} .\, \frac{\partial\, r_m}{\partial x, d_x} \right)
\end{aligned}$$

$$\begin{aligned}
&\textit{where } \ i \in \{0, m\} \\
b_i &= \mathsf{K}_i \Rightarrow \lambda y_0^{\sigma_{i,0}}, \ldots, y_{n_i}^{\sigma_{i,n_i}} .\, r_i \qquad \textit{and} \qquad \varphi = x \oplus_\sigma d_x \\
&= \mathsf{K}_i \Rightarrow \lambda \overline{y}^{\overline{\sigma_i}} .\, r_i
\end{aligned}$$

Figure 6.3 – Partial derivatives for algebraic data-types

Data-type displacement

We proceed to characterise a proposal of displacement for data-types where the change propagation permeates in every element of a data-type. This definition is syntax-directed following the constructors of the algebraic data-type.

Definition 6.3 (Algebraic data-types displacement)
Assume an algebraic data-type $\mathbb{I}$ with constructors K_i, $i \in \mathcal{I}$. If each type σ_i of constructor K_i is displaceable by $(\rho_i, \oplus_{\sigma_i}, \ominus_{\sigma_i}, \mathsf{0}_{\sigma_i}, \odot_{\sigma_i})$, then $\mathbb{I}$ is displaceable by $(\rho, \oplus_{\mathbb{I}}, \ominus_{\mathbb{I}}, \mathsf{0}_{\mathbb{I}}, \odot_{\mathbb{I}})$ such that

$$\rho \overset{def}{=} \mathsf{Z}_{\mathbb{I}} \ + \sum_{i,\ell \in \mathcal{I}} \mathsf{K}_{i-\ell} : \sigma_\ell \ + \sum_{i \in \mathcal{I}} \mathsf{K}_i : \Delta(\sigma_i)$$

where

$$\begin{cases} \mathsf{Z} & \textit{is a constructor for the zero-displacement} \\ \mathsf{K}_{i-\ell} & \textit{is a constructor for displacement from } \mathsf{K}_i \text{ to } \mathsf{K}_\ell \\ \mathsf{K}_i & \textit{is a constructor for the inner displacement in } \mathsf{K}_i \end{cases}$$

Additionally, the displacement for a data-type must be accompanied of the corresponding primitive definition $\mathsf{cong}_{\mathsf{K}_i}$ for each constructor K_i and the elimination expression $\mathsf{dcase}_{\mathbb{I}}$.

Of course, the number of constructors of $\Delta(\mathbb{I})$ depends on the combinations of the constructors of $\mathbb{I}$ and could be large enough to be handled easily. The reader can argue that some of the displacement constructors are redundant and a shorter definition could be given instead. For sure, there are more smarter and specialized displacement definitions for particular cases of algebraic data-types, here we continue to elaborate this one which we say, it can be obtained mechanically from a given inductive type $\mathbb{I}$.

Meta-theory for data-type displacements

Lemma 6.4 (Inversion)

- *If $\Gamma \vdash \mathsf{K}_i\, \overline{s} : \tau$ then $\tau = \mathbb{I}$ for some inductive type and there exist $\sigma_{i,j}$ such that $\Gamma \vdash s_{i,j} : \sigma_{i,j}$ for each sub-term.*

- *If* $\Gamma \vdash \mathsf{case}_{\mathbb{I}}\, t \;\mathsf{of}\; \{b_0 \mid \cdots \mid b_m\} : \tau$ *then there exist* $\mathbb{I}$ *and* σ_i *such that* $\Gamma \vdash t : \mathbb{I}$ *and* $\Gamma \vdash b_i : \overline{\sigma_i} \to \tau$ *for all* b_i.

Proof. Immediate from typing rules in Figure 6.2. □

Lemma 6.5 (Canonical forms)
If $\varnothing \vdash v : \mathbb{I}$ *then* $v = \mathsf{K}_i\, \overline{w}$ *for some* K_i *of the inductive type* $\mathbb{I}$.

Proof. By analysis of the values in this extension, the type assignation can only be used to give type to an inductive constructor as shown in the previous lemma. □

Lemma 6.6 (Typing the Partial Derivative)
Consider a typing context Γ *and a term* t *whose type* τ *is displaceable by* $(\rho, \oplus_\tau, \ominus_\tau, 0_\tau, \odot_\tau)$. *If* $\Gamma, x^\sigma \vdash t : \tau$ *then its derivative* $\dfrac{\partial\, t}{\partial x, d_x}$ *has type* $\Delta(\tau)$ *under the context* $\Gamma, x^\sigma, d_x^{\Delta(\sigma)}$.

Proof. We proceed to demonstrate the inductive cases for any constructor and any elimination form.

- Case $t = \mathsf{K}_i\, \overline{s}$
  The hypotheses of induction are valid for the sub-terms s_i, that is:
  whenever $\Gamma', x^\rho \vdash s_i : \sigma_i$ then the derivative $\dfrac{\partial\, s_i}{\partial x, d_x}$ has type $\Delta(\sigma_i)$.
  Then, to prove that $\mathsf{cong}_{\mathsf{K}}\left(\dfrac{\partial\, \overline{s}}{\partial x, d_x}\right)$ has type $\Delta(\mathbb{I})$ we appeal to the definition of function $\mathsf{cong}_{\mathsf{K}}$ which internalise the displacement.
- Case $t = \mathsf{case}_{\mathbb{I}}\, t \;\mathsf{of}\; \{b_0 \mid \ldots \mid b_m\}$
  The proof is to show that $\dfrac{\partial\, \mathsf{case}_{\mathbb{I}}\, t \;\mathsf{of}\; \{b_0 \mid \cdots \mid b_m\}}{\partial x, d_x}$ has type $\Delta(\tau)$.
  Following the definition of the partial derivative this points to the auxiliary function $\mathsf{dcase}_{\mathbb{I}}\, t$ defined in the tuple for $\mathbb{I}$ displaceable as in Definition 6.3. □

Lemma 6.7 (Type Preservation)
Consider a well typed term $\Gamma \vdash t : \tau$ *and an environment* η *which is* Γ*-consistent. If there exists a value* v *such that* $\eta \vdash t \Downarrow v$, *then* $\varnothing \vdash v : \tau$.

Proof. We analyse the cases for the new terms, a constructor and the case elimination. The induction hypotheses are instances of terms s_i and t' in each case.

- Case $t = \mathsf{K}_i\, \overline{s}$
  By induction hypotheses, each sub-term s_i evaluates into w_i and therefore the type of $\mathsf{K}_i\, \overline{w}$ has inductive type $\mathbb{I}$.
- Case $t = \mathsf{case}_{\mathbb{I}}\, t' \;\mathsf{of}\; \{b_0 \mid \cdots \mid b_m\}$
  For this case, the induction hypotheses are the typing assignations for term t and the branches.
  Then, term t converges into value w and the hypothesis of the evaluation ensures the correct branch to perform the evaluation which also converges. □

Theorem 6.8 (Correctness of partial derivatives of data-types)
Consider a typing context $\Gamma = \Gamma', d_x^{\Delta(\rho)}$ *where* Γ' *is the context in* $\Gamma' \vdash t : \tau$. *Then for* η *a* Γ*-consistent environment the following holds:*

$$t \oplus_\tau \frac{\partial\, t}{\partial x, d_x} \equiv t\,[x \ := \ x \oplus_\rho d_x]$$

Proof. We have the cases for the new terms, any constructor and the case elimination. The induction hypotheses are instances of terms s_i and t' respectively.

- Case $t = \mathsf{K}_i\,\overline{s}$
  This case is proved by using equivalences, the left part begins as a displacement: $\mathsf{K}_i\,\overline{s} \oplus_{\mathbb{I}} \dfrac{\partial\, \mathsf{K}_i\,\overline{s}}{\partial x, d_x}$.
  By applying definition for partial derivatives, the term is: $\mathsf{K}_i\,\overline{s} \oplus_{\mathbb{I}} \mathsf{cong}_{\mathsf{K}_i}\left(\dfrac{\partial\,\overline{s}}{\partial x, d_x}\right)$.
  Then, by definition $\mathsf{K}_i\,\overline{s} \oplus_{\mathbb{I}} \mathsf{cong}_{\mathsf{K}_i}\left(\overline{\mathsf{d}}\right) = \mathsf{K}\,\overline{(s \oplus_\sigma \mathsf{d})}$ and induction hypotheses for terms s_i we obtain the right part: $\mathsf{K}_i\ (\overline{s}\,[x \ := \ x \oplus_\rho d_x])$.
- Case $t = \mathsf{case}_{\mathbb{I}}\, t'$ of $\{b_0 \mid \cdots \mid b_m\}$
  This case remain abstract since the definition of $\mathsf{dcase}_{\mathbb{I}}$ is not provided. □

Our approach to displacements over data-types is aimed to be mechanised while it helps to propagate the change *smoothly* over terms in the system. The choice of inductive definitions for data-type representation arises naturally in the functional and certified approach to programming as we exposed in Chapter 4 in the various approaches to change description.

However a possible mechanisation is not entirely independent, the user is required to give the primitive definitions for the derivative of the case elimination and the definition of the cong function. We mentioned this in Definition 6.3, where a displaceable algebraic data-type expects those definitions. This could be alleviated since the type $\Delta(\mathbb{I})$ is also algebraic and therefore the definition of dcase is the case elimination of $\Delta(\mathbb{I})$.

In the following, we elaborate two examples to exhibit the use of the above extensions. Since our goal is to define total functions, we suppose a constant fail to denote an *undefined* value.

6.4 Examples

Example (Booleans). *In Figure 6.4 appears the extension of* λ*-diff for the boolean data-type:* `bool = true | false`. *It includes the corresponding type for boolean displacements* $\Delta(\mathsf{bool}) = \mathsf{dbool}$*, which is obtained 'mechanically' by Definition 6.3.*

The syntactic category for constructors in λ*-diff is extended with the two constructors for booleans and the boolean displacements which describe a detailed way to change: the* $\mathsf{Z}_{\mathsf{bool}}$ *constructor state the empty-change, the* TT *and* FF *displacements characterise a change*

$$\begin{array}{lcl} \mathbb{I} & ::= & \cdots \mid \mathsf{bool} \mid \mathsf{dbool} \\ \mathsf{K} & ::= & \cdots \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{Z_{bool}} \mid TT \mid FF \mid TF \mid FT \end{array}$$

$$\mathsf{case_{bool}}\, t \text{ of true} \Rightarrow s_1 | \text{false} \Rightarrow s_2$$

$$\mathsf{dcase_{bool}}\, t\, \mathsf{d}\ (s_1\,[x \ := \ x \oplus_\sigma d_x])\left(\frac{\partial\, s_1}{\partial x, d_x}\right)\ (s_2\,[x \ := \ x \oplus_\sigma d_x])\left(\frac{\partial\, s_2}{\partial x, d_x}\right)$$

Figure 6.4 – Displaceable Booleans

that does not modifies the shape of the original object, and finally the last two constructors TF and FT show the transformation of a boolean construction.

Therefore, the boolean type is displaceable by means of: $(\mathsf{dbool}, \oplus_{\mathsf{bool}}, \ominus_{\mathsf{bool}}, 0_{\mathsf{bool}}, \odot_{\mathsf{bool}})$ *together with the definition of* $\mathsf{dcase_{bool}}$. *The corresponding constants δ are defined for a full-value application*[2]*:*

$$\delta_{\oplus_{\mathsf{bool}}}\, \mathsf{b\, d} \quad = \quad \mathsf{case_{dbool}}\, \mathsf{d} \text{ of } \{\, TF \Rightarrow \mathsf{false} | FT \Rightarrow \mathsf{true} \mid \_ \Rightarrow \mathsf{b}\, \}$$

$$\delta_{\ominus_{\mathsf{bool}}}\, \mathsf{b_1\, b_2} \quad = \quad \begin{array}{l} \mathsf{case_{bool}}\ \mathsf{b_2} \text{ of } \{\ \ \mathsf{false} \Rightarrow \mathsf{case}\, \mathsf{b_1} \text{ of } \{\mathsf{false} \Rightarrow FF | \mathsf{true} \Rightarrow FT\} \\ \qquad\qquad\quad \mid\ \mathsf{true} \Rightarrow \mathsf{case}\, \mathsf{b_1} \text{ of } \{\mathsf{true} \Rightarrow TT | \mathsf{false} \Rightarrow TF\}\ \ \} \end{array}$$

$$\delta_{\odot_{\mathsf{bool}}}\, \mathsf{d_1\, d_2} \quad = \quad \begin{array}{l} \mathsf{case_{dbool}}\ \mathsf{d_1} \text{ of } \{\ \ TT \Rightarrow \mathsf{case}\, \mathsf{d_2} \text{ of } \{TT \Rightarrow TT | TF \Rightarrow TF \mid \_ \Rightarrow \mathsf{fail}\} \\ \qquad\qquad\qquad \mid FF \Rightarrow \mathsf{case}\, \mathsf{d_2} \text{ of } \{FF \Rightarrow FF | FT \Rightarrow FT \mid \_ \Rightarrow \mathsf{fail}\} \\ \qquad\qquad\qquad \mid TF \Rightarrow \mathsf{case}\, \mathsf{d_2} \text{ of } \{FT \Rightarrow TT | FF \Rightarrow TF \mid \_ \Rightarrow \mathsf{fail}\} \\ \qquad\qquad\qquad \mid FT \Rightarrow \mathsf{case}\, \mathsf{d_2} \text{ of } \{TT \Rightarrow FT | TF \Rightarrow FF \mid \_ \Rightarrow \mathsf{fail}\} \\ \qquad\qquad\qquad \mid \mathsf{Z_{bool}} \Rightarrow \mathsf{d_2}\ \ \} \end{array}$$

An implementation of the case elimination of displacements shows the detailed processing of change:

$$\begin{array}{ll} \mathsf{dcase_{bool}} & (\mathsf{b} : \mathsf{bool})\ \ (\mathsf{d} : \mathsf{dbool}) \\ & (\mathsf{btrue} : \tau)\ \ (\mathsf{dbtrue} : \Delta(\tau)) \\ & (\mathsf{bfalse} : \tau)\ \ (\mathsf{dbfalse} : \Delta(\tau)) := \\ & \qquad\qquad \mathsf{case_{dbool}}\, \mathsf{d} \ \text{ of } \{\ TT \Rightarrow \mathsf{case}\, \mathsf{b} \text{ of } \{\mathsf{true} \Rightarrow \mathsf{dbtrue}\ \ |\_ \Rightarrow \mathsf{fail}\} \\ & \qquad\qquad\qquad\qquad\quad \mid FF \Rightarrow \mathsf{case}\, \mathsf{b} \text{ of } \{\mathsf{false} \Rightarrow \mathsf{dbfalse} |\_ \Rightarrow \mathsf{fail}\} \\ & \qquad\qquad\qquad\qquad\quad \mid TF \Rightarrow \mathsf{case}\, \mathsf{b} \text{ of } \{\mathsf{true} \Rightarrow !\,(\mathsf{bfalse})\ \ |\_ \Rightarrow \mathsf{fail}\} \\ & \qquad\qquad\qquad\qquad\quad \mid FT \Rightarrow \mathsf{case}\, \mathsf{b} \text{ of } \{\mathsf{false} \Rightarrow !\,(\mathsf{btrue})\ \ |\_ \Rightarrow \mathsf{fail}\} \\ & \qquad\qquad\qquad\qquad\quad \mid \mathsf{Z_{bool}} \Rightarrow 0_{\Delta(\tau)}\ \ \} \end{array}$$

Recall in the above term, that the sub-terms of the case*-term have type τ and therefore the displacements have type $\Delta(\tau)$. The bang-operator (!), performs an* absolute *displacement by*

2. We use the term b to denote any value of type bool.

replacing the value with a recomputation. It produces a displacement and its type takes the elements to reconstruct a term.

We claim that this description, while could seem exaggerated to describe booleans transformations, is an optimal representation for change, and promotes an efficient change propagation in functions that depend on boolean inputs.

Let us see two boolean functions, the not *and* xor *operations.*

$$\begin{aligned}\mathsf{not} &= \lambda x.\,\mathsf{case}\,x\ \mathsf{of}\ \{\mathsf{true} \Rightarrow \mathsf{false} | \mathsf{false} \Rightarrow \mathsf{true}\}\\ \mathsf{xor} &= \lambda x\,y.\,\mathsf{case}\,x\ \mathsf{of}\ \{\mathsf{true} \Rightarrow \mathsf{not}\,y | \mathsf{false} \Rightarrow y\}\end{aligned}$$

Their respective partial derivatives are the functions whose body-terms are the partial derivatives of the boolean values and the not *operation. The corresponding derivatives are the following:*

$$\begin{aligned}\mathrm{D}\,\mathsf{not} &= \lambda\langle x,\,d_x\rangle\,.\,\mathsf{dcase}_{\mathsf{bool}}\,x\,d_x\,\mathsf{true}\ \ TT\ \ \mathsf{false}\ \ FF\\ \mathrm{D}\,\mathsf{xor} &= \lambda\langle x,\,d_x\rangle\,y.\,\mathsf{dcase}_{\mathsf{bool}}\,x\,d_x\,(\mathsf{not}\,y)\ \left(\frac{\partial\,\mathsf{not}\,y}{\partial x, d_x}\right)\ (y)\ \ 0_{\mathsf{dbool}}\end{aligned}$$

Example (Natural numbers). *The data-type for natural numbers with its displeceable type, the definition of term* $\mathsf{dcase}_{\mathsf{nat}}$ *has the displacements for replacement, those with a explicit substitution of the new value of x, and the displacements where a change propagation will be done.*

$$\begin{aligned}\mathbb{I} &::= \cdots \mid \mathsf{nat} \mid \mathsf{dnat}\\ \mathsf{K} &::= \cdots \mid \mathsf{zero} \mid \mathsf{suc} \mid \mathsf{Z}_{\mathsf{nat}} \mid ZZ \mid SS \mid ZS \mid SZ\end{aligned}$$

$$\mathsf{case}_{\mathsf{nat}}\,t\ \mathsf{of}\ \{\mathsf{zero} \Rightarrow s_1 | \mathsf{suc}\,n \Rightarrow s_2\,n\}$$

$$\mathsf{dcase}_{\mathsf{nat}}\,t\,\mathsf{d}\ (s_1\,[x\ :=\ x \oplus_\sigma d_x])\ \left(\frac{\partial\,s_1}{\partial x, d_x}\right)\ (((\lambda y^{\mathsf{nat}}.\,s_2)\ n)\ [x\ :=\ x \oplus_\sigma d_x])\ \left(\lambda y^{\mathsf{nat}}.\,\frac{\partial\,s_2}{\partial x, d_x}\right)$$

Figure 6.5 – Displeceable Natural numbers

$$\delta_{\oplus_{\mathsf{nat}}}\,\widehat{m}\,\mathsf{d} \;=\; \begin{array}{l}\mathsf{case}_{\mathsf{nat}}\ \widehat{m}\ \mathsf{of}\ \{\ \mathsf{zero} \Rightarrow \mathsf{case\,d\,of}\\ \qquad\qquad \{0_{\mathsf{nat}} \Rightarrow \mathsf{zero} | ZZ \Rightarrow \mathsf{zero} \mid ZS\,\widehat{n} \Rightarrow \widehat{n} \mid \_ \Rightarrow \mathsf{fail}\ \}\\ \qquad \mid \mathsf{suc}\ \widehat{m'} \Rightarrow \mathsf{case\,d\,of}\ \{0_{\mathsf{nat}} \Rightarrow \mathsf{suc}\ \widehat{m'} | SS\,\mathsf{d}' \Rightarrow \mathsf{suc}\ (\widehat{n} \oplus \mathsf{d}')\\ \qquad\qquad\qquad \mid SZ \Rightarrow \mathsf{zero} \mid \_ \Rightarrow \mathsf{fail}\ \}\ \ \}\end{array}$$

$$\delta_{\ominus_{\mathsf{nat}}}\,\widehat{m}\,\widehat{n} \;=\; \begin{array}{l}\mathsf{case}_{\mathsf{nat}}\ \widehat{n}\ \mathsf{of}\ \{\ \mathsf{zero} \Rightarrow \mathsf{case}\,\widehat{m}\,\mathsf{of}\ \{\ \mathsf{zero} \Rightarrow ZZ | \mathsf{suc}\,\widehat{n} \Rightarrow SZ\ \}\\ \qquad \mid \mathsf{suc}\ \widehat{n'} \Rightarrow \mathsf{case}\,\widehat{m}\,\mathsf{of}\ \{\mathsf{zero} \Rightarrow ZS\,\widehat{m'} | \mathsf{suc}\,\widehat{n} \Rightarrow SS\left(\widehat{n} \ominus \widehat{m'}\right)\}\}\end{array}$$

$$\delta_{\odot_{\mathsf{nat}}}\,\mathsf{d}_1\,\mathsf{d}_2 \;=\; \begin{array}{l}\mathsf{case}_{\Delta(\mathsf{nat})}\ \mathsf{d}_1\ \mathsf{of}\ \{\ ZZ \Rightarrow \mathsf{case}\,\mathsf{d}_2\,\mathsf{of}\\ \qquad\qquad \{\ 0_{\mathsf{nat}} \Rightarrow \mathsf{d}_1 | ZZ \Rightarrow ZZ \mid ZS\,\widehat{n} \Rightarrow ZS\,\widehat{n} \mid \_ \Rightarrow \mathsf{fail}\ \}\\ \qquad \mid ZS\,\widehat{m'} \Rightarrow \mathsf{case}\,\mathsf{d}_2\,\mathsf{of}\\ \qquad\qquad \{\ 0_{\mathsf{nat}} \Rightarrow \mathsf{d}_1 | ZZ \Rightarrow ZZ \mid ZS\,\widehat{n} \Rightarrow ZS\,\widehat{n} \mid \_ \Rightarrow \mathsf{fail}\ \}\\ \qquad \mid SS\,\mathsf{d}' \Rightarrow \mathsf{case}\,\mathsf{d}_2\,\mathsf{of}\\ \qquad\qquad \{\ 0_{\mathsf{nat}} \Rightarrow \mathsf{d}_1 | SZ \Rightarrow SZ \mid SS\,\mathsf{d}'' \Rightarrow SS\ (\mathsf{d}' \odot \mathsf{d}'')\\ \qquad\qquad\qquad \mid \_ \Rightarrow \mathsf{fail}\ \}\\ \qquad \mid SZ \Rightarrow \mathsf{case}\,\mathsf{d}_2\,\mathsf{of}\\ \qquad\qquad \{\ 0_{\mathsf{nat}} \Rightarrow \mathsf{d}_1 | ZZ \Rightarrow SZ \mid ZS\,\widehat{m'} \Rightarrow SS \mid \_ \Rightarrow \mathsf{fail}\ \}\\ \qquad \mid 0_{\mathsf{nat}} \Rightarrow \mathsf{d}_2\ \ \}\end{array}$$

As the reader can see, this choice of data-type displacement scales quickly because of the number of constructors. We end with an implementation of $\mathsf{dcase}_{\mathsf{nat}}$ *and the differentiation of the sum operation with respect to its first argument:*

$$\begin{array}{ll}\mathsf{dcase}_{\mathsf{nat}} & (\mathsf{n} : \mathsf{nat})\ \ (\mathsf{d} : \mathsf{dnat})\\ & (\mathsf{bzero} : \tau)\ \ (\mathsf{dbzero} : \Delta(\tau))\\ & (\mathsf{bsuc} : \tau)\ \ (\mathsf{dbsuc} : \Delta(\tau)) :=\\ & \qquad \mathsf{case}_{\mathsf{dbool}}\,\mathsf{d}\ \ \mathsf{of}\ \{\ ZZ \Rightarrow \mathsf{case\,n\,of}\ \{\mathsf{zero} \Rightarrow \mathsf{dbzero}\ \ |\_ \Rightarrow \mathsf{fail}\}\\ & \qquad\qquad \mid SS\,\mathsf{d}' \Rightarrow \mathsf{case\,n\,of}\ \{\mathsf{suc\,n'} \Rightarrow \mathsf{dbsuc\,d'} |\_ \Rightarrow \mathsf{fail}\}\\ & \qquad\qquad \mid ZS\,\mathsf{m} \Rightarrow \mathsf{case\,n\,of}\ \{\mathsf{zero} \Rightarrow !\,(\mathsf{bsuc\,m})\ \ |\_ \Rightarrow \mathsf{fail}\}\\ & \qquad\qquad \mid SZ \Rightarrow \mathsf{case\,n\,of}\ \{\mathsf{suc\,n'} \Rightarrow !\,(\mathsf{bzero})\ \ |\_ \Rightarrow \mathsf{fail}\}\\ & \qquad\qquad \mid \mathsf{Z}_{\mathsf{nat}} \Rightarrow 0_{\Delta(\tau)}\ \ \}\end{array}$$

$$\begin{array}{rcl}\mathsf{sum} & = & \lambda x, y.\,\mathsf{case}\,x\ \mathsf{of}\ \{\mathsf{zero} \Rightarrow y | \mathsf{suc}\,z \Rightarrow \mathsf{suc}\ (\mathsf{sum}\,z\,y)\}\\ \mathrm{D}_0\,\mathsf{sum} & = & \lambda\,\langle x,\,d_x\rangle\,.\,\lambda y.\,\mathsf{dcase}_{\mathsf{nat}}\,x\,d_x\ \ y\ \ 0_{\mathsf{nat}}\ \ (\mathsf{sum}\,x\,y)\ \ \left(\dfrac{\partial\,\mathsf{sum}\,x\,y}{\partial x, d_x}\right)\end{array}$$

Chapter 7

Closing remarks

7.1 $\lambda-$diff Related Work

The second part of this work, presented and formalised in the last two chapters, contributes with a dependent[1] approach to changes and a differential lambda calculus.

In this chapter we discuss a related work which is very close to our incremental proposal elaborated so far. We finish with some closing remarks.

7.1.1 A theory of changes for Higher-Order Languages

As mentioned in Chapter 4, a very similar framework to incrementality was elaborated, Cai, Guiarruso, Rendel and Ostermann[2] . In the following we describe and compare this framework, focusing on differentiation by dependent change structures and the $\mathcal{Derive}$ operator over terms, the reader may see the details in the article where the system named ILC for incrementalizing λ-calculi, is elaborated [25].

ILC: Incrementalizing λ-calculi

The first component to achieve incrementality, is to define a dedicated theory for change description. This is managed by the theory of changes in ILC as a dependent theory[3] where any type has a set of displacements and operations to update a term and to compute a change or displacement. Given type ρ, it introduces a structure $\widehat{\rho}$ formed by a dependent set of changes (every change depends on an element s of ρ) and the incremental primitives or change operations for data-update $\oplus$ and change calculation $\ominus$. A change structure $\widehat{\rho}$ is a mathematical instance of an abelian group.

1. The dependent approach is not related to a dependent type system *per se* but to the fact that any displacement 'depends' on the old value.
2. Visit the site of the Incremental λ-Calculus project: http://www.informatik.uni-marburg.de/~pgiarrusso/ILC/
3. As we said, the dependent theory is for change description.

Definition 7.1 (Change structure)
A change structure for a given type ρ is a tuple $\widehat{\rho} = (\rho, \Delta, \oplus, \ominus)$ if:

1. *ρ is a set, called the base set.*
2. *Given $v \in \rho$ then Δv is a set, the change set.*
3. *Given $v \in \rho$ and $\mathsf{d} \in \Delta v$ then $v \oplus \mathsf{d} \in \rho$.*
4. *Given $v,\ v' \in \rho$ then $v \ominus u \in \Delta v$.*
5. *Given $v,\ u \in \rho$ then $v \oplus (u \ominus v)$ equals u.*

The ILC framework is based on a lambda calculus *parametrized* by a `plugin` containing basic types, its primitive operations and a change structure with incremental primitives.

$$\text{basic type change structure:}\quad \widehat{\iota} = (\iota, \Delta_\iota, \oplus_\iota, \ominus_\iota)$$
$$\text{where } \oplus_\iota = \texttt{plugin-defined}$$
$$\text{and } \ominus_\iota = \texttt{plugin-defined}$$

$$\text{function type change structure:}\quad \widehat{\tau \to \sigma} = (\tau \to \sigma, \Delta_{\tau\to\sigma}, \oplus_{\tau\to\sigma}, \ominus_{\tau\to\sigma})$$
$$\text{where } \Delta_{\tau\to\sigma} = \tau \to \Delta_\tau \to \Delta_\sigma$$

The $\mathcal{Derive}$ operator allows not only differentiation of functions but full differentiation of terms with respect to all free variables, it is defined recursively:

$$\begin{aligned}
\mathcal{Derive}(c) &= \texttt{plugin-defined} \\
\mathcal{Derive}(\lambda x.\, t) &= \lambda x\ dx.\, \mathcal{Derive}(t) \\
\mathcal{Derive}(s\, t) &= \mathcal{Derive}(s) t\ \mathcal{Derive}(t) \\
\mathcal{Derive}(x) &= dx
\end{aligned}$$

The above definition promotes a term derivation with respect to all its free variables. This is attested by the corresponding typing rule for $\mathcal{Derive}$ which needs a context for changes in order to assign a type for the derivative of a term:

$$\frac{\Gamma \vdash t : \tau}{\Gamma,\ \Delta\Gamma \vdash \mathcal{Derive} t : \Delta\tau}\ \textsc{Derive}$$

The change-context $\Delta\Gamma$ has $dx : \Delta\tau$ for each variable $x : \tau$ in Γ. Therefore, if $\Gamma \vdash t : \tau$ holds, then the derivative is a change.

The authors show the correctness of differentiation using a correspondence between the differential framework ILC and the non-standard denotational semantics of the lambda calculus. The main theorem of correctness resumes in the following equivalence:

$$f\ (s \oplus \mathsf{d}_s) = (f\ s) \oplus (\mathcal{Derive}(f)\ s\, \mathsf{d}_s)$$

The article includes a formalization of the system and the meta-theory in Agda[4] and an implementation in Scala[5].

4. http://wiki.portal.chalmers.se/agda/pmwiki.php
5. http://www.scala-lang.org/

7.1.2 Discussion

Although some similarities between the system just described and our approach (the dependent change description, an operator to compute derivatives and function differentiation) we can distinguish some differences.

Differentiation We claim that the ILC system allows a *static* differentiation of any λ-term, not only functions, while the system λ-diff includes a formal constructor for function differentiation and an extra term-operation for partial derivatives of terms. This means that the latter is a framework to *dynamically* compute derivatives, without user intervention for computing any program transformation. This also ensures a change propagation within the evaluation of functions, that is done automatically.

For instance, the evaluation of a function differential in λ-diff computes the partial derivative of the body sub-term of the function while propagates the displacement of the argument. In ILC the derivative of the function must be computed before evaluation which is also an automatic transformation.

Change management In both systems, λ-diff and ILC, the efficiency lies on the change description. Any incremental function, at the end, depends on the most basic functions and values that is, the basic types and the way they express and manage changes within the primitive operations. Therefore, the foundations of incrementality are the change descriptions, the plugins in ILC and the displaceable types in λ-diff.

The use of plugins, which are stated by the user, makes impossible to deduce mechanically the definitions of base types derivatives. The user is 'responsible' for the plugins, while in our approach the user is free to give an inductive definition for data-types which can be syntactically derived a type for data-displacements.

With respect to the efficiency achieved by means of change management, the ILC system has a notion of *self-maintainability* for those functions which can obtain an output without using the inputs but just their changes. Hence, efficient incremental computations arise naturally with self-maintainable derivatives. This statement is follows from the remark characterising the optimizations under an incremental framework: the gradual computation through changes ensure an output whose run-time does not depend on the input size.

7.2 Conclusions

The long and wide road to achieve program optimizations has many different ways to be walked. We chose the incrementality path aiming to understand the first steps of those who already walked through it and to seek a new and improved proposal. While doing this, we found another group of colleagues trying to pursue the same goal and fortunately.

We want to emphasize our position to bring incrementality in a functional paradigm:

- A faithful change description:
  given a term, restrict the ways it can change and keep track of how an object can

change or transform into another, that is a formal way to link the changes inside the structures, a kind of factorising the behaviour of changes in elements.

- The change reflection and propagation:
  make consistent and smooth maps between input changes and outputs, that is accurate input changes enable accurate outputs.
- A program transformation by means of differentiation for programs to strengthen the reuse of values.

7.3 Future work

We hopefully will continue to extend the road of incrementality *via* the following subjects.

The λ-diff calculus

We consider continuing the development of the deterministic λ-diff-calculus. The idea is to extend the system until reaching the formalisation of the Calculus of Inductive Constructions to provide a framework to ease the reasoning of incremental programs inside the Coq proof assistant.

Improve memoization

One of the most used techniques to program optimization is memoization, as we stated and reported by the self-adjusting paradigm.

Since our approach keep track of the dependencies of the inputs to reflect the changes in the outputs we suggest to improve memoization in the following way: when searching a key in the table which does not exist, we can reuse the value of the *closer* entry in the table.

The notion of data dissection and displacements could help to decide which is the closer element to the current input, in order to reuse its value. We can generalise this by choosing a subset of reusable outputs. Then, a notion of *neighbourhood* is necessary in order to decide and maintain the reusable values.

Incremental type-checking

Our motivation to develop an incremental approach for the lambda calculus is to improve the type theory on which a proof assistant is based.

Since the proof construction is done inside a system which will finally validate the proof by a type checking, we want to take advantage of the interactive nature of the system to apply an incremental approach.

The idea of checking the proofs incrementally has been already suggested before [98]. Recently, the work of Puech in his PhD dissertation [96], offers a practical tool to perform incremental type-checking addressed by means of a certifying approach to programming. The formal program verification using the certifying approach uses the basic concept of *programming with certificates*. A program has a witness of the correctness of the computations performed, the certificate could be verified without the program.

Puech takes the logical framework LF of Pfenning [88] as a language where the proofs can be represented and manipulated. He offers an OCAML library (Gasp) for programming with certificates. An incremental type checker is included in this library which can reuse sub-derivations of typing derivations.

The document model The document model in the Paral-ITP project expects a typed repository to perform formal analysis of proof versions. The efforts achieved to this purpose agree with a framework for incrementality. The interactive nature of the proof assistants allows the incorporation of this kind of optimizations.

Remember that the process of proof-construction is conducted (locally) by the user in a proof management system. The development of a theory evolves dynamically, that is, it is large and non-linear, due to the inclusion of new definitions and all kind of *changes* in statements, definitions and proofs. Moreover, the user does not start and finish a theory or a development with just a single phase of 'coding' nor even with only one file or without editions that may last for months. Additionally, the collaborative work suggests shared developments where more than one user may take advantage of verified parts of the theory to continue working without loosing time while re-type-checking the theorems.

We want to take advantage of incremental computing to offer a better performance when a user is developing a theory in the COQ proof management system. A proposal is to consider the development of a theory as a process suitable to apply incrementality.

For instance, whenever a proof is done and afterwards a change in a definition or the inclusion of a new theorem simplifies a finished proof by making small changes. Then we want to earn the time spent when checking the original proof and reuse some pieces of the proof that did not change and are already checked.

The idea of considering a proof as an object constructed incrementally suggests a proof representation by differences leading to keep and reuse those parts which are already proven or even type-checked in order to alleviate the proof checking.

The approach to incrementality proposed so far and a possible combination with a powerful language extension as the Cybele plugin offers, could lead to manage proof scripts in a formal repository to study the changes and to follow an incremental verification of them. Then, the need of *locally* correct proof steps, in order to have a safe change description to be propagated while computing, leads to work around the (ambitious) goal of make an incremental type-checker and the development of a typed tactic language for COQ.

However this is not a simple task, some obstacles have to be defeated:

1. A proof not always succeeds type-checking when typing QED at the end of a demonstration. This leads to the question of how to benefit of those proofs.
2. A theorem could have different proofs. This opens the question about the change representation of proofs.

The remarks above are related directly to the tactics used in the proofs, hence a language with types for tactics. The more a tactic is reliable, the more the proofs will be verified.

Our proposal focuses on the process of creation and development of proofs by means of proof scripts and the generation of proof-terms. A change in the proof script triggers a change in the rest of the proof and in the proof-check. A 're-check' is just a matter of verifying the parts (in)directly affected by the change.

A formal repository can be maintained with the checked and therefore *safe* proof-terms. This proposal is not aimed on deriving explicitly incremental algorithms but to apply methods discussed in this work to obtain results efficiently.

Bibliography

[1] Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In *Typed Lambda Calculi and Applications, 6th International Conference*, TLCA '03, pages 16–30, 2003.

[2] Michael Abbott, Thorsten Altenkirch, Conor McBride, and Neil Ghani. ∂ for data: Differentiating data structures. *Fundam. Inform.*, 65(1-2):1–28, 2005.

[3] Andreas Abel. Weak normalization for the simply-typed lambda-calculus in twelf (extended abstract). In *In Logical Frameworks and Metalanguages (LFM 04), IJCAR*, 2004.

[4] Umut Acar. *Self-Adjusting Computation.* PhD thesis, Carnegie Mellon University, May 2005.

[5] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. *SIGPLAN Not.*, 38(1):14–25, January 2003.

[6] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Shan Leung Maverick Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 531–540, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[7] 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*, pages 1–38, 2008.

[8] Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006*, pages 69–83, March 2006.

[9] A. Asperti, W. Ricciotti, C.S. Coen, and E. Tassi. A new type for tactics. Technical report, Department of Computer Science, University of Bologna, 2009.

[10] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 3–15. ACM, 2008.

[11] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge Univ. Press, Cambridge, 1998.

[12] Hendrik Pieter Barendregt. *The lambda calculus : its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland, Amsterdam, New-York, Oxford, 1981.

[13] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.

[14] Henk Barendregt and Herman Geuvers. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter Proof-assistants Using Dependent Type Systems, pages 1149–1238. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[15] Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In Alan Robinson and Andrei Voronkov, editors, *Handbook of automated reasoning*, pages 1149–1238. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 2001.

[16] Bruno Barras. Proof assistants. `http://www.lix.polytechnique.fr/~barras/mpri/notes/index-2-7-2.html`, 2010. Master Parisien de Recherche en Informatique, notes of the course.

[17] Bruno Barras, Carst Tankink, and Enrico Tassi. Asynchronous processing of coq documents: From the kernel up to the user interface. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, pages 51–66, 2015.

[18] Gilles Barthe, John Hatcliff, and Peter Thiemann. Monadic type systems: Pure type systems for impure settings. In *Proceedings of the Second HOOTS Workshop*. Elsevier, 1998.

[19] P. N. Benton, Gavin M. Bierman, and Valeria de Paiva. Computational types from a logical perspective. *J. Funct. Program.*, 8(2):177–193, 1998.

[20] Yves Bertot and P. Casteran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, Berlin, Germany, 2004.

[21] Małgorzata Biernacka and Dariusz Biernacki. A context-based approach to proving termination of evaluation. *Electronic Notes in Theoretical Computer Science*, 249(0):169 – 192, 2009. Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009).

[22] Jan Olaf Blech and Benjamin Grégoire. Certifying compilers using higher-order theorem provers as certificate checkers. *Formal Methods in System Design*, 38(1):33–61, 2011.

[23] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *TACS'97. Springer-Verlag LNCS 1281*, pages 515–529. Springer-Verlag, 1997.

[24] Ana Bove and Venanzio Capretta. Computation by prophecy. In *Typed Lambda Calculi and Applications. 8th International Conference, TLCA 2007*, pages 70–83. Springer, 2007.

[25] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. *SIGPLAN Not.*, 49(6):145–155, June 2014.

[26] Pierre Castéran and Matthieu Sozeau. A Gentle Introduction to Type Classes and Rewriting in Coq, May 2012.

[27] Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'12, pages 413–424, New York, USA, 2012. ACM.

[28] Yan Chen, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit self-adjusting computation for purely functional programs. *J. Funct. Program.*, 24(1):56–112, 2014.

[29] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. Last draft version, April 8 2015, `http://adam.chlipala.net/cpdt/cpdt.pdf`.

[30] Guillaume Claret, Lourdes Del Carmen González Huesca, Yann Régis-Gianas, and Beta Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *Interactive Theorem Proving*, Rennes, France, Jul 2013.

[31] Sylvain Conchon and Jean-Christophe Filliâtre. A Persistent Union-Find Data Structure. In *ACM SIGPLAN Workshop on ML*, Freiburg, Germany, October 2007.

[32] Evelyne Contejean and Pierre Corbineau. Reflecting proofs in first-order logic with equality. In *CADE*, pages 7–22, 2005.

[33] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.4*, 2012. August 12, 2012.

[34] Pierre Corbineau. Autour de la clôture de congruence avec coq. Master's thesis, ENS, 2001. In French.

[35] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[36] David Delahaye. A tactic language for the system coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000.

[37] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, October 1980.

[38] Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theor. Comput. Sci.*, 309(1-3):1–41, 2003.

[39] John Field and Tim Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 307–322, New York, NY, USA, 1990. ACM.

[40] John Henry Field. *Incremental Reduction in the Lambda Calculus and Related Reduction Systems*. PhD thesis, Cornell University, 1991.

[41] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987.

[42] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A small scale reflection extension for the coq system. Research Report RR-6455, Inria Saclay Ile de France, 2015.

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

[44] Benjamin Grégoire, Loïc Pottier, and Laurent Théry. Proof certificates for algebra and their application to automatic geometry theorem proving. In Thomas Sturm and Christoph Zengler, editors, *Automated Deduction in Geometry*, volume 6301 of *Lecture Notes in Computer Science*, pages 42–59. Springer Berlin Heidelberg, 2011.

[45] Robert Harper. *Practical Foundations for Programming Languages*. Electronic textbook, 2015. Revision 2.03 `http://www.cs.cmu.edu/~rwh/plbook/2nded.pdf`.

[46] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[47] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.

[48] Dimitri Hendriks. Proof reflection in coq. *Journal of Automated Reasoning*, 29(3-4):277–307, 2002.

[49] Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. Logical relations and parametricity – a reynolds programme for category theory and programming languages. *Electronic Notes in Theoretical Computer Science*, 303:149 – 180, 2014. Proceedings of the Workshop on Algebra, Coalgebra and Topology (WACT 2013).

[50] Ralf Hinze. A new approach to generic functional programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 119–132, New York, NY, USA, 2000. ACM.

[51] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51(1–2):117 – 151, 2004. Mathematics of Program Construction (MPC 2002).

[52] Gérard Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, September 1997.

[53] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, april 1989.

[54] Daniel W. H. James and Ralf Hinze. A Reflection-based Proof Tactic for Lattices in Coq. In *Trends in Functional Programming*, pages 97–112, 2009.

[55] Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming: An introduction. In *3rd International Summer School on Advanced Functional Programming*, pages 28–115. Springer-Verlag, 1999.

[56] Felix Joachimski and Ralph Matthes. Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. *Arch. Math. Log.*, 42(1):59–87, 2003.

[57] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, March 1966.

[58] Slawomir Lasota, David Nowak, and Yu Zhang. On completeness of logical relations for monadic types. *CoRR*, abs/cs/0612106, 2006.

[59] Xavier Leroy. Functional programing and type systems. http://gallium.inria.fr/~xleroy/mpri/2-4/, 2012. Master Parisien de Recherche en Informatique 2.4, notes of the course.

[60] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system Documentation and user's manual release 4.02*. September 24, 2014. http://caml.inria.fr/pub/docs/manual-ocaml/index.html.

[61] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004. In French.

[62] Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[63] Pierre Letouzey. Coq Extraction, an Overview. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, editors, *Logic and Theory of Algorithms, 2008*, volume 5028 of *LNCS*. Springer-Verlag, 2008.

[64] Ruy Ley-Wild. *Programmable Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, October 2010.

[65] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.

[66] Yanhong A. Liu. CACHET: an interactive, incremental-attribution-based program transformation system for deriving incremental programs. In *KBSE*, pages 19–26, 1995.

[67] Yanhong A. Liu. Dependence analysis for recursive data. In *ICCL*, pages 206–215, 1998.

[68] Yanhong A. Liu. Efficiency by incrementalization: An introduction. *Higher-Order and Symbolic Computation*, 13(4):289–313, 2000.

[69] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. Strengthening invariants for efficient computation. *Sci. Comput. Program.*, 41(2):139–172, 2001.

[70] Yanhong A. Liu and Tim Teitelbaum. Caching intermediate results for program improvement. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, USA, June 21-23, 1995*, pages 190–201, 1995.

[71] Pattie Maes. Concepts and experiments in computational reflection. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 147–155, New York, NY, USA, 1987. ACM.

[72] Conor McBride. The derivative of a regular type is its type of one-hole contexts (extended abstract), 2001.

[73] Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 287–295, 2008.

[74] D. Michie. “memo” functions and machine learning. *Nature*, 218:19–22, april 1968.

[75] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209 – 220, 1991.

[76] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[77] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3):23:1–23:49, 2008.

[78] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980.

[79] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998.

[80] Martijn Oostdijk and Herman Geuvers. Proof by computation in the coq system. In *Theoretical Computer Science*, pages 293–314. Elsevier, 2000.

[81] Bryan O’Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O’Reilly Media, Inc., 1st edition, 2008.

[82] Özgür Sümer. *Adaptive Inference for Graphical Models*. PhD thesis, University of Chicago, March 2012.

[83] R. Paige. *Formal Differentiation: A Program Synthesis Technique*, volume 7. UMI Research Press, Ann Arbor, Michigan, 1981. Revision of Ph.D. Thesis (1979).

[84] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3), july 1982.

[85] Michel Parigot. $\Lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer Berlin Heidelberg, 1992.

[86] C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d’Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996. In French.

[87] Frank Pfenning. Computation and deduction, 1997. Draft (April 2, 1997).

[88] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1063–1147. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 2001.

[89] David Pichardie and Vlad Rusu. Defining and reasoning about general recursive functions in type theory: a practical method. Research Report PI 1766, LANDE - INRIA - IRISA, VERTECS - INRIA, 2005.

[90] Brigitte Pientka and Ryan Kavanagh. *A beginners guide to programming in Beluga*. June 20, 2012. http://www.cs.mcgill.ca/~complogic/beluga/tutorial.pdf.

[91] Benjamin C. Pierce. Types and programming languages: The next generation, 2003. Invited tutorial at *Logic in Computer Science (LICS)*.

[92] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2015. Version 3.2 http://www.cis.upenn.edu/~bcpierce/sf.

[93] Laurence Pierre and Thomas Kropf, editors. *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*. Springer, 1999.

[94] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997.

[95] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1), January 2003.

[96] Matthias Puech. *Certificates for Incremental Type Checking*. PhD thesis, Università di Bologna and Université Paris Diderot, April 2013.

[97] Florian Rabe and Kristina Sojakova. Logical relations for a logical framework. *ACM Trans. Comput. Logic*, 14(4):32:1–32:34, November 2013.

[98] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 502–510, New York, NY, USA, 1993. ACM.

[99] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.*, 5(3):449–477, july 1983.

[100] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1982.

[101] Matthieu Sozeau. Program-ing finger trees in coq. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 13–24, New York, NY, USA, 2007. ACM.

[102] Antonis Stampoulis and Zhong Shao. Veriml: typed computation of logical terms inside a language with effects. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP'10, pages 333–344, New York, NY, USA, 2010. ACM. Extended version: http://flint.cs.yale.edu/flint/publications/verimltr.pdf.

[103] Antonis Stampoulis and Zhong Shao. Static and user-extensible proof checking. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 273–284, New York, NY, USA, 2012. ACM. Extended version: http://flint.cs.yale.edu/flint/publications/sv11exprf.pdf.

[104] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, March 1984.

[105] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.

[106] Simon Thompson. *Type theory and functional programming*. International computer science series. Addison-Wesley, 1991.

[107] Duru Türkoğlu. *Stable Algorithms and Kinetic Mesh Refinement*. PhD thesis, University of Chicago, March 2012.

[108] Vene Varmo. *Categorical programming with inductive and coinductive types*. PhD thesis, University of Tartu, Estonia, august 2000.

[109] Lionel Vaux. The differential $\lambda\mu$-calculus. *Theor. Comput. Sci*, pages 166–209, 2007.

[110] Lionel Vaux. *λ-calcul différentiel et logique classique: interactions calculatoires*. PhD thesis, Université Aix-Marseille 2, 2007. In French.

[111] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[112] Philip Wadler. Propositions as types, November 2014.

[113] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Log.*, 4(1):1–32, 2003.

[114] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.

[115] P.M. Whitman. *Free Lattices*. Harvard University, 1941.

[116] Freek Wiedijk. Comparing mathematical provers. In *Mathematical Knowledge Management, Second International Conference*, MKM 2003, pages 188–202, 2003.

[117] D. Yellin and R. Strom. Inc: A language for incremental computations. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 115–124. ACM, 1988.

[118] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in coq. *SIGPLAN Not.*, 48(9), September 2013.

[119] Justin Zobel. *Writing for Computer Science*. Springer-Verlag, 2004.