Université Paris Cité

École Doctorale 386 — Sciences Mathématiques de Paris Centre
Inria

Towards an efficient and formally-verified convertibility checker

Par Nathanaëlle Courant

Thèse de doctorat d'INFORMATIQUE

Dirigée par Xavier Leroy

Présentée et soutenue publiquement le 19/09/2024

Devant un jury composé de :

Małgorzata Biernacka	Assistant Professor, University of Wroclaw	(rapportrice)
Guillaume Melquiond	Directeur de recherche, Université Paris-Saclay	(rapporteur)
Thibaut Balabonski	Maître de conférences, Université Paris-Saclay	(examinateur)
Sandrine Blazy	Professeure, Université de Rennes	(examinatrice)
Delia Kesner	Professeure, Université Paris Cité	(examinatrice)
Matthieu Sozeau	Chargé de recherche, Inria Rennes	(examinateur)
Xavier Leroy	Professeur, Collège de France	(directeur)

Résumé

Le test de convertibilité, qui vérifie si deux λ-termes sont égaux à β-réduction près, est une partie essentielle de la vérification de types et de preuves dans les assistants de preuve basés sur la théorie des types, comme Coq, Agda et Lean. Naturellement, la correction d'un tel test est nécessaire pour s'assurer que les preuves ainsi vérifiées sont valides ; mais avoir un test efficace est également requis à la fois pour une interaction en temps réel avec l'utilisateur, ainsi que pour la recherche de preuve. Dans cette thèse, nous commençons par proposer une sémantique à grands pas efficace pour l'évaluation forte paresseuse, qui est l'élément critique pour implanter un test de convertibilité de manière classique. Cette sémantique est presque mécaniquement dérivée à partir de la sémantique à petits pas de réduction externe gauche du λ-calcul, en la convertissant en sémantique à grands pas, ajoutant des environnements pour éviter les substitutions, et mémoïsant les évaluations. Des lemmes dits de transfert permettent de partager des calculs qui ne sont pas immédiatement redondants. Nous montrons ensuite comment nous pouvons améliorer cela en considérant le test de convertibilité dans sa globalité, en le voyant comme de la recherche de preuve. Pour cela, nous étudions ce qui peut être considéré comme une preuve de (non-)convertibilité, en se basant sur les travaux existants. Cela donne lieu à un nouvel algorithme parallèle et sans heuristiques, qui ne duplique pas les calculs et vient avec des garanties de complexités dans le pire des cas. Nous avons dérivé une machine virtuelle de cet algorithme, et l'avons rendue plus efficace en suivant Grégoire et Leroy, montrant que notre algorithme est adapté à la compilation des λ-termes en entrée pour plus d'efficacité. Cette sémantique à grands pas et ce test de convertibilité parallèle ont été implémentés en OCaml et validés expérimentalement, et se révèlent significativement plus performants que Coq dans certains cas. Ils ont également été tous les deux formalisés et vérifiés en Coq, augmentant la confiance en nos travaux.

Abstract

The convertibility test, which checks if two λ-terms are equal up to β-reduction, is an essential part of typechecking and proof verification in proofs assistants based on type theory, such as Coq, Agda and Lean. Naturally, the correctness of such a test is necessary to ensure the proofs thus verified are valid; but having an efficient test is also required both for real-time interaction with the user, and for proof search. In this thesis, we start by proposing an efficient, strong call-by-need, big-step semantics for the λ-calculus, which is the critical building block to implement a convertibility test in a classical manner. This semantics is almost mechanically derived from the semantics of small-step leftmost-outermost reduction of the λ-calculus, by converting them to big-step semantics, adding environments to avoid substitution, and memoizing evaluations. So-called transfer lemmas enable the sharing of computations that are not obviously identical. We then show how we can improve upon this by considering the convertibility problem in its entirety, viewing it as proof search. To this end, we study what can be considered a (non-)convertibility proof, drawing upon existing work. This gives rise to a new parallel, heuristic-less algorithm for this test, which does not duplicate computations, and comes with worst-case complexity guarantees. We derived a virtual machine from this algorithm, and made it more efficient by following Grégoire and Leroy, showing that our algorithm is suited to compilation of the input λ-terms for additional efficiency. Both this semantics and this efficient parallel convertibility test have been implemented in OCaml and experimentally validated, significantly outperforming Coq in some cases. They are also both formalised and verified using Coq, building confidence in our work.

Mots-clés

lambda-calcul, vérification de preuve, assistants de preuve, Coq, convertibilité, réduction forte, réduction ouverte, évaluation paresseuse

Keywords

lambda-calculus, proof checking, proof assistants, Coq, convertibility, strong reduction, open reduction, lazy evaluation

Présentation

Introduction

Lorsqu'on s'intéresse à la notion de preuve mathématique, il est important de souligner la nécessité de faire confiance à des propositions de base, les axiomes. Une preuve, pour peu qu'elle soit suffisamment détaillée – quitte à en devenir difficilement compréhensible intuitivement – peut alors être vérifiée pas à pas, de manière algorithmique : c'est là le fondement du concept d'assistant de preuve. Dans cette thèse, nous nous intéressons plus particulièrement à l'un d'entre eux : Coq, né d'un effort de recherche français à la fin des années 1980. Celui-ci a depuis été largement utilisé, que ce soit dans le milieu académique ou industriel, à la fois dans un but de formalisation mathématique ou de correction logicielle. Cependant, lors de l'utilisation un outil de vérification logicielle tel que Coq, il est important de rester vigilant quant à la correction de l'outil lui-même, et de ne pas se fier aveuglément à un outil faillible. Une partie centrale de cet outil consiste en ce que l'on appelle un *test de convertibilité*, notion que nous détaillons plus loin.

Les travaux présentés ici s'inscrivent alors dans une démarche d'optimisation de ce test de convertibilité et de vérification de cette partie de Coq. Nous proposons pour cela divers algorithmes pour la convertibilité, conçus pour être efficaces, et eux-mêmes vérifiés en Coq. En effet, après avoir précisé quelques notions indispensables à la compréhension des résultats obtenus, nous introduirons deux grandes méthodes. La première est une approche incrémentale par rapport à l'état de l'art, dont l'apport consiste majoritairement en la vérification formelle de celle-ci ; la seconde est entièrement nouvelle et est accompagnée, outre sa vérification formelle, de solides garanties d'efficacité.

Commençons par introduire quelques notions de λ-calcul, qui seront cruciales non seulement pour définir le test de convertibilité, mais aussi pour comprendre les contributions apportées dans cette thèse.

Le *λ-calcul* est un modèle de calcul très simple mais extrêmement puissant, capable d'exprimer toute fonction calculable. Il est bâti à partir de seulement trois constructions : les variables x ; les fonctions $\lambda x.t$, qui abstraient une variable x et renvoient un résultat t dépendant de celle-ci ; et l'application $t\ u$ d'une fonction t à un argument u. Ensemble, ces trois constructions donnent naissance à des objets nommés λ-termes, et fournissent une base solide à la théorie des langages de programmation. Le λ-calcul est équipé d'une opération dite de β-réduction, qui permet de transformer une fonction appliquée à un argument $(\lambda x.t)\ u$ en le corps de cette même fonction où le paramètre a été substitué par l'argument $t[x := u]$.

Dans ce contexte, le test de convertibilité consiste à identifier si deux λ-termes sont égaux à β-réduction près. Il s'agit là d'un problème indécidable, mais si l'on impose aux termes en entrée d'être fortement normalisants – c'est-à-dire qu'on ne peut pas écrire de suite infinie de β-réductions à partir de ce terme –, alors le problème devient décidable, par un algorithme qui consiste simplement à réduire les termes jusqu'à obtenir une *forme normale*, qui ne peut plus se réduire ; puis à directement comparer les formes normales pour l'égalité, à renommage des variables près. Cette procédure donne naturellement naissance à l'organisation de ce manuscrit : la première partie est dédiée au calcul efficace d'une forme normale ; tandis que la seconde considère le test de convertibilité dans son ensemble et permet d'obtenir des optimisations significatives, qui ne seraient pas possibles en décomposant celui-ci en normalisation puis comparaison. Dans ces travaux, nous avons par ailleurs considéré un λ-calcul étendu, plus proche de celui sur lequel Coq est fondé et sur lequel il effectue des tests de convertibilité. Nos travaux s'étendraient ainsi naturellement à Coq.

Réduction forte paresseuse

Dans le chapitre 4, nous proposons une sémantique paresseuse pour la réduction forte, c'est-à-dire une sémantique permettant le calcul de formes normales de λ-termes, incluant la réduction sous les abstractions (i.e. à l'intérieur des fonctions), et n'effectuant aucun calcul inutile. Pour cela, on effectue des transformations successives de sémantiques à partir d'une sémantique d'appel par nom, à laquelle on ajoute tout d'abord des environnements pour éviter les opérations de substitution, avant d'effectuer une mémoïsation pour éviter de calculer plusieurs fois des réductions identiques. Dans ce contexte, la mémoïsation est plus complexe qu'habituellement car on peut s'intéresser à deux résultats possibles d'une réduction – un dit profond, correspondant à la forme normale du terme, l'autre dit superficiel, correspondant à sa forme normale de tête faible –, et des lemmes dits de transfert nous permettent de passer d'un résultat à l'autre sans dupliquer les calculs. Nous donnons ensuite une autre présentation équivalente des mêmes règles, qui s'avère plus adaptée à l'implantation, où l'on calcule systématiquement le résultat superficiel de la réduction, avant d'en calculer le résultat profond si c'était celui qui était demandé. Nous étendons ensuite mécaniquement la sémantique obtenue aux extensions du λ-calcul : les constructeurs, le filtrage par motifs et les points fixes.

Dans le chapitre 5, nous détaillons l'implantation en OCaml de cette sémantique. On constate que l'implantation se fait assez naturellement en suivant la sémantique elle-même, et qu'il est possible d'utiliser les constructions de paresse d'OCaml pour simplifier l'implantation. Ceci résulte en un interpréteur concis, écrit en seulement quelques dizaines de lignes de code.

Dans le chapitre 6, nous présentons la preuve en Coq de la correction de cette sémantique. Pour cela, nous avons écrit cette sémantique dans un style dit *pretty-big-step*, qui permet de partager certaines parties des preuves, ainsi que de raisonner sur la non-terminaison. Pour cela, nous implantons successivement toutes les sémantiques présentées dans le chapitre 4, et nous prouvons que chaque sémantique est bien compatible avec la précédente. Nous avons également créé une petite bibliothèque Coq pour permettre de raisonner sur les types inductifs en créant des principes d'induction plus puissants, qui nous ont permis de significativement simplifier les preuves. Ainsi, l'implantation présentée dans ce chapitre représente environ dix mille lignes de preuve Coq.

Dans le chapitre 7, nous présentons une évaluation expérimentale succincte de cette approche, où l'on constate que notre implantation est du même ordre de grandeur d'efficacité que celle actuellement utilisée par Coq. Cela justifie la pertinence de notre approche, puisqu'elle permet d'obtenir une efficacité similaire à Coq tout en étant formellement vérifiée.

Nous passons brièvement sur le chapitre 8, qui détaille la littérature existante ainsi que les extensions possibles des travaux que nous présentons, notamment la possibilité de construire une machine virtuelle à partir de notre sémantique à grands pas, ou également de faire une analyse de la complexité d'une implantation de notre sémantique.

Convertibilité

Dans la partie suivante, nous présentons donc – comme précisé dans l'introduction – une approche nouvelle, intégrant du parallélisme afin d'obtenir un procédé plus efficace pour le test de convertibilité.

Nous effectuons tout d'abord le constat suivant : lorsque, au cours d'un test de convertibilité, nous obtenons deux variables libres différentes en tête comme dans $x\ t_1 \overset{?}{\equiv} y\ t_2$, les deux termes ne peuvent pas être convertibles, et nous pouvons conclure immédiatement sans avoir besoin de réduire en forme normale les arguments t_1 et t_2 passés à ces variables. Par ailleurs, si nous avons un test de la forme $f\ t_1 \overset{?}{\equiv} f\ t_2$, et que nous sommes capables de prouver que t_1 et t_2 sont convertibles, nous pouvons également conclure immédiatement que $f\ t_1$ et $f\ t_2$ le sont, sans avoir besoin de déplier le corps de f pour calculer ces valeurs. Cependant, un problème dans ces situations est qu'il y a un choix à faire, et qu'il est difficile voire impossible d'anticiper quelle option permettra de conclure le plus rapidement. Par exemple, dans le cas du test de convertibilité $x\ t_1\ t_2 \overset{?}{\equiv} x\ t_3\ t_4$, l'algorithme utilisé dans Coq effectue la comparaison de droite à gauche ; et dans le cas de $f\ t_1 \overset{?}{\equiv} f\ t_2$, il tente toujours de prouver la convertibilité de t_1 et de t_2, même s'il aurait été plus efficace de déplier f. Notre stratégie consiste alors à explorer les deux pistes simultanément plutôt qu'essayer de proposer une heuristique qui serait nécessairement incomplète.

Pour cela, dans le chapitre 9, on commence par présenter une machine abstraite pour la réduction en forme normale de tête faible, proche de la machine utilisée par Coq, et qui peut être redémarrée en préservant les calculs déjà effectués pour la suite du test de convertibilité en cours. Nous la modifions ensuite pour obtenir une machine capable de réduire plusieurs termes en parallèle, ce qui nous permet d'implanter l'optimisation mentionnée plus haut. Cette machine est ainsi constituée d'un certain nombre de *threads* ou fils d'exécution qui effectuent des réductions, certains pouvant attendre d'obtenir le résultat d'autres threads avant de continuer leurs calculs. Le test de convertibilité lui-même requiert l'ajout d'un nouveau type de thread, dit de convertibilité, qui va attendre d'avoir à sa disposition le résultat de deux threads de réduction pour tester la convertibilité des valeurs associées. Certains threads de convertibilité correspondent également au *et* ou au *ou* logique de deux autres threads de convertibilité, et sont créés lorsqu'on doit faire un choix entre deux pistes à explorer. Ces pistes s'exécutent alors en parallèle, et l'on dispose d'une sémantique de court-circuit permettant de mettre fin précocement au calcul si on peut déterminer son résultat à partir d'un seul de ces threads. Cela nécessite de pouvoir interrompre certains threads, tout en leur permettant de s'exécuter à nouveau dans le futur s'ils redeviennent nécessaires, ce que nous implantons en maintenant une liste de dépendances entre threads dans la machine. La machine ainsi obtenue constitue une contribution majeure de cette thèse.

Dans le chapitre 10, nous transformons la machine abstraite obtenue en une machine virtuelle, afin de permettre la compilation des termes initiaux pour une performance supérieure. Nous nous inspirons ensuite de la littérature existante, et plus précisément des travaux de Grégoire et Leroy, pour améliorer encore cette machine virtuelle.

Ensuite, dans le chapitre 11, nous détaillons comment nous avons construit une preuve Coq de la correction de cet algorithme. Dans la preuve Coq, nous avons formalisé la machine de réduction et la stratégie de convertibilité, mais pas la gestion des threads actifs, qui est laissée non-déterministe pour obtenir une sur-approximation de la sémantique effective. La preuve elle-même est seulement modérément complexe – environ dix mille lignes de preuve Coq – mais l'enjeu majeur, qui a demandé des efforts significatifs, a consisté en la détermination des invariants précis nécessaires à la preuve de correction. Au total, ces invariants représentent plus d'une centaine de lignes en Coq, dont soixante constituant la définition de trois prédicats mutuellement inductifs. Ceux-ci étant particulièrement complexes à manier en Coq, l'itération nécessaire pour affiner les invariants n'a été possible que grâce à l'écriture d'une bibliothèque permettant de générer les principes d'induction associés automatiquement, que Coq n'était pas capable d'obtenir.

Dans le chapitre 12, nous présentons une borne de complexité de notre algorithme, par rapport à la longueur d'une plus courte preuve raisonnable de convertibilité existante. Pour cela, nous devons d'abord définir le concept de *longueur d'une preuve raisonnable*, ce que nous détaillons à travers le concept de structure de réduction. Nous étendons ensuite ce concept en celui de structure de réduction effective, qui correspond à notre machine, et nous permet d'obtenir une borne de complexité exponentielle en la longueur de la preuve la plus courte. Cette borne est significativement meilleure que ce dont Coq est capable, puisque celui-ci a un temps de calcul qui ne peut être borné par aucune fonction calculable.

Le chapitre 13 consiste en une évaluation expérimentale détaillée de cette deuxième méthode, où nous considérons un certain nombre de tests différents. Nous analysons la durée du test de convertibilité lorsqu'effectué par Coq ou par notre algorithme, implémenté en OCaml, ce qui nous permet de constater notamment que notre algorithme est exponentiellement plus rapide que Coq dans un certain nombre de cas. Nous présentons également un cas pathologique où notre algorithme est tout de même exponentiellement plus lent que Coq.

Le chapitre 14 est quant à lui consacré à la littérature liée à nos travaux, ainsi qu'aux travaux futurs pouvant être envisagés. Dans les pistes de travaux futurs particulièrement notables, on peut mentionner un ordonnanceur plus sophistiqué pour l'exécution qui nous permettrait d'améliorer la borne de complexité afin qu'elle soit linéaire en le nombre d'étapes de réduction, ou encore une capacité à produire et rejouer des traces de preuves permettant une vérification substantiellement plus rapide que le test initial.

En conclusion, cette thèse présente des travaux améliorant significativement l'état de l'art concernant les tests de convertibilité efficaces et vérifiés. Deux approches indépendantes y sont détaillées, accompagnées de leurs implantations et d'une discussion incluant une évaluation expérimentale.

À Orphée,
pour ce que je ne saurais exprimer en mots :
merci.

À Ariane,
pour m'avoir rappelé que tout pouvait être source d'émerveillement.

Acknowledgements

Writing this manuscript has been one of the most difficult, yet one of the most rewarding work of my life. As I struggle to bring it the finishing touches, I cannot help but look back and thank those without whom this text would not exist.

First, many thanks to Xavier, who was my adviser and trusted me with such an open subject, allowing me to focus on the research I found the most interesting. Your encouragements led me to outdo myself, and I do not believe this work would be as interesting if you never pushed me to innovate.

I am grateful to Małgorzata Biernacka and Guillaume Melquiond, who accepted to report on this manuscript, and whose comments have been valuable in improving it. I am also grateful to all the members of the jury for coming to the defence.

Thank you to everyone in the Cambium team, who made it such a nice place to work: François, Didier, Luc, Jean-Marie, Damien, Florian, Gabriel, Jacques, as well as Gabriel and Jacques-Henri who were often there; special thanks to all the other Ph.D. students, whose heroic work and determination helped me finish my own Ph.D.: Basile, Alexandre, Armaël, Paulo, Glen, Frédéric, Clément.

I would also like to thank Jean-Christophe, for letting me teach the practical classes of his compilation course I had so thoroughly enjoyed several years earlier. You fueled my passion for compilation, and it was thanks to you that I was able to work with Xavier.

I collaborated on many occasions during this Ph.D. with Gabriel Scherer, who has a knack for turning ideas into papers. Your enthusiasm made me write and share things I did that would have remained in obscurity without it; you have my gratitude for that.

Xavier Rival was my tutor when I was at ENS; and our discussions were infrequent but extremely valuable. Thank you for them, as they helped me decide what I wanted to do.

When my Ph.D. grant expired after three years, I started working at OCamlPro. They let me finish my manuscript, on my work time, at the pace I wanted. If not for them, this manuscript would have remained half-finished, in limbo.

Among the people at OCamlPro, Muriel was always there to support me, encourage me and push me to work on my manuscript; Guillaume, Pierre, Vincent and now Basile of the flambda team were key to such a pleasant work environment, without which I would never have found the strength to work on this manuscript. I am grateful to all of you.

My parents lit my passion for computer science from an early age, and encouraged me to study and do a Ph.D. Thank you for what you did for me.

To all the health and mental health professionals who helped me during the years of my Ph.D. and enabled my transition, I am deeply grateful. You will most likely never read this, but this work would not exist if not for you. You helped me turn into the blooming person I am today, instead of letting me wither.

I would not be there either if not for my friends. Many thanks to the group from the BOcal: Corbeau, DrNo, elarnon, Lwenn, Milton, Minithorynque, Tito, tobast; I was always glad to see you, even if we did not see each other on that many occasions.

Victoire, you have always been a trusted friend. You were also the first of my friends from my class at ENS to obtain your Ph.D., showing me it was possible to actually complete it; and you were supportive at all times and in all circumstances. Thank you for helping me not to give up.

Alexandre, Milla, Jeanne, since I met you, you were always there when I needed it. We spent many moments together, and they still bring warmth to my heart when I think about them. Thank you so much, and I wish you all the best, both for your own Ph.D. and in life.

To Ariane, who was born at the end of the first year of my thesis, you brought me many sleepless nights, but also the most joy in the world. You were like a sun, bringing light even when I was in the darkest places.

Finally, Orphée, I owe you everything. You were my beacon in the storm, you always supported me, from when I made my most difficult decisions to when I stared blankly at the page in front of me, unable to write a single word. You helped me understand myself more than anyone ever did, and you taught me so much. Marrying you and the birth of Ariane were the two most beautiful days of my life; but I wanted to thank you for all the ordinary days we spent together: they make life beautiful.

Contents

Part I.

Introduction

1. Introduction

How can we ever be sure of something? What does it mean to prove an assertion? More succinctly and more deeply at the same time, what is a proof?

If we look at the most likely meaning of the word in daily life,[1] *to prove something* means to convince someone, be it one's interlocutor or a court of law, that what one is saying it true. In scientific fields such as physics or biology, proving something will mean being able to convince the majority of scientists working in this field of the validity of one's claim. Even in mathematics or computer science, where the rules for what is considered a proof are formal, such kinds of proofs are only rarely written, and proving a theorem means convincing the rest of the community that there exists such a formal proof – not writing one explicitly.

What this means is that there is a deep connection between the concept of proof and the need to communicate. A proof, then, will rely on a set of rules for this communication: both axioms, which are assumptions that both parties will agree to – for instance the facts in a court, that two sets are equal if they have the same elements for a set theorist –, and rules to reason about those, the most well-known being the *modus ponens*: if A is true and A implies B, then B is true as well. With those rules, we can share our reasoning with others, and others can verify that our proof does respect those rules. Those rules will of course depend on the context as well: if an experiment gives the same result each time it is performed, this will be a proof that it will always give the same result for a physicist, while a mathematician will not call it a proof.

The axioms and rules used by mathematicians since the foundational crisis of mathematics in the 19th century are both purely abstract, in the sense that no access to our world is needed to find or check them like a biologist or physicist would need, and well-defined, meaning that one can precisely write them. These two facts together make it possible to write a computer program, often called a theorem checker or proof assistant, that will check if a given proof is correct by those rules.

In order to manipulate such proofs to check they are correct, and more generally to reason about then, we need to see them as mathematical objects – and this is exactly what logicians do, both facts mentioned above ensuring that we can indeed define what they are precisely. But then, when we think again about the modus ponens rule for these objects, we have a proof of A and a proof of A implies B, and we need to produce a proof of B: a natural way to obtain this was if the proof of A implies B was a function that

[1] Although it is probably not the most likely meaning for the readers of this manuscript!

took proofs of A and produced proofs of B from them! In this setting, a proof becomes the same thing as a program, and this correspondence has been further developed and enriched, becoming known as the *Curry-Howard isomorphism*[How80].

This equivalence is the foundation of using *type theory* as one of the many possible axiomatisations of mathematics. Type theory is based on the idea of giving types to objects: it is not possible to add a number and a function, for instance. Using the Curry-Howard isomorphism, we can then use those same types as statements of theorems, and the functions and other objects we can build as proofs for such theorems, thereby giving a unified framework for both definitions (and programs), and proofs.

An extension of type theory, called the calculus of constructions[CH88], is the theory on which the Coq proof assistant[The24] is based. It is a software, first released in 1989, that allows the user to write and check proofs written in the calculus of constructions. Since then, it has been used for numerous applications, the most famous of them being the verification of the four colour theorem[Gon08], the Feit-Thompson theorem concerning the order of finite simple groups[Gon+13], and in another domain, the CompCert C compiler, the first verified optimising compiler[Ler09].

At the heart of Coq, and in general, type theory, is the need of computation of programs; more specifically λ-terms. The λ-calculus is a way to write such programs and functions, together with rules for computation on them: a function that simply returns its argument, then applied to x, should morally speaking be considered the same as x. This is a core rule of type theory: if two objects are the same up to some computation, then they should be considered the same. Testing whether this is true is called the *convertibility test*, and will be the focus of this thesis.

There has of course been considerable prior work concerning the convertibility test of Coq, both to make an efficient version of it, illustrated by the `vm_compute`[GL02] and `native_compute` tactics[BDG11]; and to make versions that are formally verified in Coq, as in the MetaCoq project[Soz+20]. However, our work is, to our knowledge, the first effort on doing both at the same time.

Even if efficiency is almost always a desirable property, let us explain why it matters here. Coq proof are written by a human, interactively. It is a folklore in interface design that response times are primordial: a response time less than 0.1s feels instantaneous to the user, a response time more than 1s is enough to break the flow of thought of the user, while a response time above 10s will even lose the user's attention.

Verification is a key concern as well. Indeed, a proof assistant is not magically protected from bugs in its implementation, and such bugs can make the proof assistant be able to prove false theorems. The code that is necessary to trust to ensure we can trust the proofs verified by the proof assistant is called the *trusted code base*; we want this code to be as simple as possible, to minimise the risk of bugs there. However, this directly conflicts with our efficiency goal: more efficient programs tend to be more complex, and therefore, more likely to be subject to bugs. Thus, verification gives us the best of both worlds: efficient code, but which has been proved correct, and therefore not part of the trusted code base.[2]

Thus, our objective is to obtain an efficient and formally verified convertibility checker for Coq. Before we dive into the details of our work, let us stop and consider what we mean and what we are looking for by that. First, let us ponder over the meanings of the word *efficient*.

One possible meaning of *efficient* would be to consider pure speed or memory use. If that was the choice we were interested in, we would have compiled the λ-terms to low-level C code, used CompCert to compile said code, and run the resulting program at full speed. Although of definite practical interest, and although there may be a few points of theoretical interest, this would probably not have brought new insights: an unverified convertibility test for Coq terms that compiles the term to machine code (using the OCaml compiler) already exists.

Another possible meaning would try to minimise the number of simulated β-reduction steps. However enticing, this approach suffers from a serious drawback. Indeed, optimal strategies are known for β-reduction, but they may perform a non-elementary[3] number of administrative reductions[GS17]. Thus, the cost of β-reductions is completely dominated by administrative steps, and the number of β-reductions is no longer a good proxy for the time complexity of the resulting program. Since no better strategy is known despite optimal reduction being studied for almost half a century, it seemed unlikely for us to both find and formally prove a better strategy for optimal reduction.

Instead, we chose to try to optimise for worst-case complexity. By this, we mean that if there is a way to prove or disprove convertibility in a certain amount of time, for a measure that is to be defined, we want the time taken not to be too large compared to this minimum amount of time.

The other objective of this work is for our convertibility checker to be *formally verified*. However, formal verification always has its limits, and these need to be defined to know what is really verified and what is trusted.

A first question is to know what kind of verification effort we are doing. One possibility is *a posteriori validation*: we have an unverified program for checking convertibility which outputs a proof (in a format that is either generic for proofs, or specific for this problem), and then a verified checker reads that proof and verifies it is correct. With this method, some amount of proof work has to be done for every input, either by a generic proof system (such as Coq) if the proof is in a generic format, or by a specific checker that is formally verified. In both cases, we have a serious drawback concerning efficiency: the cost of verification is often high, and this will be no exception. Indeed, verifying a proof

[2] Note that, due to Gödel's incompleteness theorem, we can never hope to prove that a proof assistant is fully correct inside itself; however, we can prove that its implementation corresponds to the underlying theory, a task that is still formidable, but at least possible.

[3] In this context, elementary means smaller than a tower of exponentials of fixed height.

of convertibility or non-convertibility often is almost as long as finding the proof itself,[4] at least concerning the costs of reduction. Thus, it will be almost as easy to prove the proof-generating program as to write a checker for its proofs. Besides, if the proof format was a generic one such as Coq, then Coq would need to be run to check every invocation of the convertibility test, defeating the objective of having a convertibility checker that could be used to verify parts of Coq!

Another possibility, slightly related, is generating a program that will output whether the two terms are convertible or not, and then use *translation validation* to check the resulting program will indeed output the correct result. With translation validation, the program transformation is again unverified, and can optionally output a proof that its inputs and the generated program behave in the same way. Then, a verified checker takes the inputs, the generated program and optionally the generated proof, and checks that the generated program will output the correct result.[5] Again, there is verification work that needs to be done for each convertibility test. However, depending on how the program is generated, the verification effort can be proportional to the size of the terms instead of proportional to the amount of work that has to be done to prove (non-)convertibility, so this is already a lot better! In our case however, convertibility and strong reduction need quite complex runtime mechanisms, which will have to be proved correct to ensure the generated program runs correctly. The verification effort is therefore not much more important if we want to prove correctness of the compilation part as well.

The possibility we implemented is to convert our terms to a program that will output the correct result, in a manner that is completely verified. With this solution, all verification is done only once: the program generator is proved correct according to the semantics of the output machine, and a convertibility test is then simply generating the program and running it. In our case, the machine is an abstract machine specifically designed for convertibility. It is implemented in OCaml, and we specify its semantics in Coq, so the implementation in OCaml has to be trusted. However, it follows quite faithfully the Coq semantics, so the trust in it can be relatively high. The reason why we implement it in OCaml instead of Coq is that the implementation maintains additional information to be able to perform scheduling, which is non-deterministic in the Coq semantics, where we prove that all executions return the correct answer.

Another question is about termination. Although proving termination properties of programs can be necessary for verification (the existence of a terminating program producing a result satisfying a given property means there exists such a result), in our goal of only producing correct answers, termination is not a very important property. Our convertibility checker always terminates when fed strongly-normalising terms, but we did not prove

[4] Although we will see that this is not completely the case with our efficient convertibility checker later, as it has a form of *choice points*! However, this is only for part of the proof search; the reduction steps themselves are a large part of the computation, and a checker will have to do the same amount of work than a prover here.

[5] As always in translation validation, the checker will use the fact that the generated program is not any program but a program generated from the input program, thus having the same shape, to avoid stepping into uncomputable territory.

this property, as it would require a lot of additional proof effort. Thus, what we proved about it is that *if* it terminates with a given result, then that result is correct. Likewise, we do not prove any theorem about complexity, since proving a complexity bound of a program implies proving its termination. However, we performed a pen-and-paper complexity analysis of our checker.

2. The λ-calculus

2.1. λ-terms and variables

The *λ-calculus* is one of the oldest models of computation, being first introduced by Church in the 1930s [Chu32; Chu33]. Unlike the Turing machine, which is only mostly useful for the theory of computation, the use of the λ-calculus is widespread in programming language theory. It forms the basis of most functional programming languages, and while bare, contains insights about the use and power of functions.

At the heart of the λ-calculus is the definition of *λ-terms*, which are defined by the following grammar:

$$t ::= x \mid \lambda x.t \mid t\ t$$

Here, the x are *variables*, which are names drawn from a (countably) infinite set. The term $\lambda x.t$ is called a *λ-abstraction*, or more simply an abstraction, and intuitively represents the function that associates t to x, written $x \mapsto t$ in mathematics. The term $t_1\ t_2$ is an *application* and represents the application of the function t_1 to the argument t_2. When writing terms, application associates to the left, so that $t_1\ t_2\ t_3$ is $(t_1\ t_2)\ t_3$ and not $t_1\ (t_2\ t_3)$. Another source of ambiguity are λ-abstractions, which extend as far to the right as possible, so that $\lambda x.t_1\ t_2$ is $\lambda x.(t_1\ t_2)$ and not $(\lambda x.t_1)\ t_2$.

For a given λ-term t, we can define its *free* variables, written $\mathbf{fv}(t)$, which are the variables in t which have at least one occurrence that is not under a λ-abstraction binding them. We will also call a *binder* any construction that creates bound variables, for now only λ-abstractions. Free variables are defined inductively by:

$$\begin{aligned} \mathbf{fv}(x) &= \{x\} \\ \mathbf{fv}(\lambda x.t) &= \mathbf{fv}(t) \setminus \{x\} \\ \mathbf{fv}(t_1\ t_2) &= \mathbf{fv}(t_1) \cup \mathbf{fv}(t_2) \end{aligned}$$

In the following, we will say that λ-terms containing free variables are *open*, while λ-terms without any free variable are *closed*.

For a given *representation* of a λ-term, we can speak about its *bound* variables, which are those that are named by a λ-abstraction. Note that, unlike the terminology suggests, a variable can be both bound and free at the same time, as in the given representation of the λ-term $x\ (\lambda x.x)$. Equality between λ-terms is slightly tricky to define, as it is equality up to renaming of bound variables, also called *α-equivalence*: for instance, $\lambda x.x$ and $\lambda y.y$ denote the same λ-term. The idea behind α-equivalence is that the semantics of a λ-term, or of a program, should not depend on the programmer-provided names of the bound variables. Thus, we want to be able to (carefully) rename bound variables, and we want α-equivalence to express the conservation of the underlying structure when renaming variables in this way. An informal method to define α-equivalence is to draw a line from each variable to the nearest enclosing λ-abstraction with the same name, then remove all variable names: two λ-terms are α-equivalent if the result is the same for both. More formally, we use an alternate representation of variables called *de Bruijn indices*: instead of variable names, we only remember how many abstractions need to be traversed to find the abstraction corresponding to each variable. The grammar of terms using de Bruijn indices is shown below:

$$t ::= n \mid \lambda.t \mid t\ t$$

In this form, all variable names become unnecessary, and two λ-terms are α-equivalent if and only if their representations using de Bruijn indices are identical. For instance, the λ-term $\lambda x.\lambda y.x\ y$ would be represented as $\lambda.\lambda.1\ 0$, while $\lambda x.\lambda y.y\ x$ would be represented as $\lambda.\lambda.0\ 1$, and $\lambda y.\lambda x.y\ x$, which is α-equivalent to the first term shown, would be represented as $\lambda.\lambda.1\ 0$, identical to the de Bruijn representation of that term.

When converting an open λ-term to de Bruijn indices, we need a way to associate numbers to such free variables, which do not have a corresponding binder. A simple way is to give them numbers greater than the number of enclosing binders, saying that such variables do not refer to a binder defined in the term, and we extend the definitions of open and closed terms accordingly. We show below a variant of de Bruijn indices called the *locally nameless* representation [Cha12], which avoids the problem of associating a number to free variables. In this representation, we have both named variables and de Bruijn indices, but the binders only apply to the variables with de Bruijn indices: named variables are automatically free. On the other hand, we constrain the indices to be smaller than the number of enclosing binders, so that a de Bruijn variable always refers to a binder in the term. We give a grammar of λ-terms in locally nameless representation below, followed by conversion functions from a λ-term with named variables to one in locally nameless representation.

$$t ::= n \mid x \mid \lambda.t \mid t\ t$$

In this definition, n is a nonnegative integer for bound variables, while x is for free variables. The λ-abstractions no longer carry a variable, since they are directly referred to by bound variables, using their position in the list of enclosing binders.

The conversion of λ-terms to the locally nameless representation is given by the two functions **index** and **db** below: $\mathbf{index}_\rho(x)$ locates the position of x in the *environment*

ρ, a list of variables, while $\mathbf{db}_\rho(t)$ converts t to the locally nameless representation using ρ as the list of variable names bound by the enclosing binders. The full conversion from a λ-term of its locally nameless representation is thus the function $\mathbf{db}_{[]}$.

$$\begin{aligned}
\mathbf{index}_{y::\rho}(x) &= 0 && \text{if } x = y \\
\mathbf{index}_{y::\rho}(x) &= 1 + \mathbf{index}_\rho(x) && \text{otherwise} \\[1em]
\mathbf{db}_\rho(x) &= \mathbf{index}_\rho(x) && \text{if } x \in \rho \\
\mathbf{db}_\rho(x) &= x && \text{otherwise} \\
\mathbf{db}_\rho(\lambda x.t) &= \lambda.\mathbf{db}_{x::\rho}(t) \\
\mathbf{db}_\rho(t_1\ t_2) &= \mathbf{db}_\rho(t_1)\ \mathbf{db}_\rho(t_2)
\end{aligned}$$

Equality among λ-terms becomes easy to define: t_1 is α-equivalent to t_2, written $t_1 =_\alpha t_2$ or $t_1 = t_2$ in the following, if $\mathbf{db}_{[]}(t_1) = \mathbf{db}_{[]}(t_2)$.

The main advantage of de Bruijn indices, or the locally nameless representation, is that α-equivalence becomes trivial: using de Bruijn indices, two terms are α-equivalent if and only if they are identical. When working with a proof assistant, this greatly simplifies the proofs.

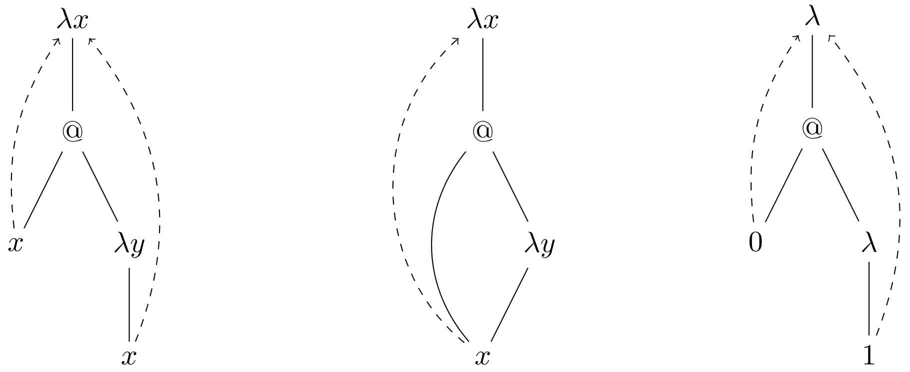
However, de Bruijn indices have a major drawback when we consider terms no longer as trees – their natural representation, since they are inductively defined –, but as directed acyclic graphs – i.e. trees where we identify some subtrees together – to express *sharing* of subterms. For instance, consider the term $\lambda x.x\ (\lambda y.x)$. This term can be represented as a graph by sharing both instances of x, like in Figure 2.1. However, in the de Bruijn representation, the variable nodes have difference indices and thus cannot be shared![1]

In this thesis, we will sometimes work with de Bruijn indices and sometimes with named variables, depending on whichever is most adapted for the task at hand. For instance, our Coq formalisation in chapter 6 takes terms with de Bruijn indices as inputs, but produces results with named variables in order to preserve sharing.

2.2. Substitution

A fundamental operation to define the semantics of the λ-calculus is *substitution*. The basic idea behind substitution is to replace all occurrences of a given variable in a term by another λ-term. However, variables must be handled with care, to avoid *capture*, that is, changing the binding site of a variable by accident. For example, $(\lambda x.x\ y)[y := x]$

[1] An alternate representation of variables is de Bruijn levels, where binders are enumerated from top to bottom instead of from bottom to top. However, while it allows sharing in this case, it prevents the sharing of $\lambda x.x$ in the case of $(\lambda x.x)\ (\lambda y.\lambda x.x)$. It is also possible to preserve sharing by inserting explicit renamings in the term, but this loses the unique representation property.



(a) Named representation, without sharing

(b) Named representation, with sharing

(c) De Bruijn representation, not shareable

Figure 2.1.: Different ways to represent the term $\lambda x.x\ (\lambda y.x)$. The term is read from top to bottom, where a λ node denotes an abstraction of the term represented by the child, a @ node denotes the application of the left subtree to the right subtree, and variable nodes denote the variables they contain. The backward edges from variables to λ nodes show which binder the variables refer to.

should not be equal to $\lambda x.x\ x$. Instead, we need to rename the bound variable x first to z, giving a result equal to $\lambda z.z\ x$.

The substitution of x by u in t, which we will write $t[x := u]$, is defined recursively as follows:

$$\begin{aligned}
x[x := u] &= u \\
y[x := u] &= y \quad \text{if } y \neq x \\
(t_1\ t_2)[x := u] &= (t_1[x := u])\ (t_2[x := u]) \\
(\lambda y.t)[x := u] &= \lambda y.(t[x := u]) \quad \text{if } y \neq x \wedge y \notin \mathbf{fv}(u)
\end{aligned}$$

Note that this definition is not total, as we require that y is neither x nor in $\mathbf{fv}(u)$ in the $\lambda y.t$ case. In the case where the result is not defined by the equations above, we perform an *α-renaming* on t, that is, we give the bound variables of t *fresh* names, different from x and not in $\mathbf{fv}(u)$, so that the result is well defined.

We can also perform *parallel substitution*, that is, substitution of multiple variables at once, without substituting under the values $\overline{u}$ being substituted. Here, $\overline{x}$ means any sequence of variables, and $\overline{u}$ is a sequence of λ-terms, with the same length as $\overline{x}$. The very similar definition is given below:

$$
\begin{aligned}
x_i[\overline{x} := \overline{u}] &= u_i \\
y[\overline{x} := \overline{u}] &= y \quad \text{if } y \notin \overline{x} \\
(t_1\ t_2)[\overline{x} := \overline{u}] &= (t_1[\overline{x} := \overline{u}])\ (t_2[\overline{x} := \overline{u}]) \\
(\lambda y.t)[\overline{x} := \overline{u}] &= \lambda y.(t[\overline{x} := \overline{u}]) \quad \text{if } y \notin \overline{x} \wedge y \notin \mathbf{fv}(\overline{u})
\end{aligned}
$$

To give a definition of substitution that is total, and thus, more easily used in formal proofs, we turn to de Bruijn indices again. However, the definition is trickier in this case, as the term u that is being substituted for is not necessarily closed.

Moreover, substitution needs to operate on open terms: for instance, we could need to replace all occurrences of the first unbound variable. Thus, the easiest way to define substitution on locally nameless or de Bruijn terms is to define parallel substitution first, and to deduce simple substitution from it.

A parallel substitution over de Bruijn terms is defined by a function σ from nonnegative integers to terms, meaning that the free variable at level i should be replaced by $\sigma(i)$. We also write $t\sigma$ for the parallel substitution of σ in t, which we define below:

$$
\begin{aligned}
n\sigma &= \sigma(n) \\
x\sigma &= x \\
(t_1\ t_2)\sigma &= (t_1\sigma)\ (t_2\sigma) \\
(\lambda.t)\sigma &= \lambda.t(\Uparrow \sigma)
\end{aligned}
$$

Here, $\Uparrow \sigma$ is the substitution mapping the variable 0 to 0, and the variable $i+1$ to $\uparrow (\sigma i)$, where $\uparrow t$ is t with all its free variables increased by 1. It is defined as $\uparrow_0 t$, where $\uparrow_k t$ increases all free variables at a level greater or equal than k by 1:

$$
\begin{aligned}
\uparrow_k n &= n && \text{if } n < k \\
\uparrow_k n &= n + 1 && \text{if } k \leq n \\
\uparrow_k x &= x \\
\uparrow_k (t_1\ t_2) &= (\uparrow_k t_1)\ (\uparrow_k t_2) \\
\uparrow_k (\lambda.t) &= \lambda.(\uparrow_{k+1} t)
\end{aligned}
$$

From this definition of parallel substitution, it is easy to define substitution of the first n free variables by terms of $\overline{u}$. To that end, we use the substitution $\sigma(i) = u_i$ for $i < n$ and $\sigma(i) = i - n$ otherwise, which replaces the first n unbound variables by the terms of $\overline{u}$ in that order, and considers the free variables starting from n to be the new unbound variables, by reducing them by n. The simple substitution of a single value is then defined by the above definition with $\overline{u}$ consisting in a single value.

2.3. β-reduction and convertibility

The fundamental operation in λ-calculus, which turns it into a computation model, is called *β-reduction*.[2] Intuitively, the application of a function $\lambda x.t$ to an argument u behaves like $t[x := u]$, namely, the *body* t of the function, where the *formal parameter* x is replaced by the *actual argument* u. β-reduction is a binary reduction relation between λ-terms, written $\rightarrow_\beta$. It is defined inductively by the following rules:

$$\dfrac{\textsc{AppAbs}}{(\lambda x.t)\ u \rightarrow_\beta t[x := u]} \qquad \dfrac{\overset{\textsc{CtxApp1}}{t_1 \rightarrow_\beta t_2}}{t_1\ u \rightarrow_\beta t_2\ u} \qquad \dfrac{\overset{\textsc{CtxApp2}}{t_1 \rightarrow_\beta t_2}}{u\ t_1 \rightarrow_\beta u\ t_2} \qquad \dfrac{\overset{\textsc{CtxAbs}}{t_1 \rightarrow_\beta t_2}}{\lambda x.t_1 \rightarrow_\beta \lambda x.t_2}$$

We will note $\rightarrow_\beta^*$ the *reflexive transitive closure* of $\rightarrow_\beta$, which expresses that a term t reduces in zero or more steps to a term u. We say that a λ-term t is in *normal form*, and note $t \not\rightarrow_\beta$, if there is no λ-term u such that $t \rightarrow_\beta u$. We will also say that t is *strongly normalising* if there is no infinite chain of reductions starting from t, and *weakly normalising* if there exists a term in normal form u so that $t \rightarrow_\beta^* u$.

While these rules seem very simple, they give rise to a rich model of computation. Most notably, the λ-calculus is Turing-complete, meaning it can express all computable functions. Consequently, deciding whether a λ-term is strongly or weakly normalising is equivalent to solving the halting problem, and therefore undecidable [Tur37].

We will call *β-equivalence*, and write $\equiv_\beta$, the reflexive transitive symmetric closure of $\rightarrow_\beta$. The *convertibility* problem is deciding whether two λ-terms are β-equivalent. Convertibility is an undecidable problem as well [Chu36], thus we need to reduce the scope of the problem if we want to have a decidable algorithm. Fortunately, the terms we are interested in are all strongly normalising, which is more than enough to get decidability, as we will see below.

The important property that allows us to get an algorithm is that $\rightarrow_\beta$ is *confluent*. We say that a reduction relation $\rightarrow$ is confluent if for all t, u, v, if $t \rightarrow^* u$ and $t \rightarrow^* v$, then there exists w such that $u \rightarrow^* w$ and $v \rightarrow^* w$. A way to graphically represent this definition is the diagram in Figure 2.2a, where the solid arrows indicate known reductions, and the dashed arrows express the existence of a value that satisfies them.

The confluence of the λ-calculus has several important consequences, the first being the uniqueness of normal forms. Indeed, if a term t has two normal forms u and v, then there exists w such that $u \rightarrow_\beta^* w$ and $v \rightarrow_\beta^* w$. However, u and v being normal forms, this implies $u = w = v$, so normal forms are indeed unique. Moreover, the question of convertibility can be significantly simplified: two terms u and v are β-equivalent if and only if there exists w such that $u \rightarrow_\beta^* w$ and $v \rightarrow_\beta^* w$. Indeed, the relation $u \equiv v$ defined

[2] There seems to be a tendency to name everything with Greek letters in λ-calculus: we have already seen λ-terms, α-equivalence, and β-reduction, and we will later see δ-reduction and η-expansion.

$$\begin{tikzcd}
t \arrow[r, "*"] \arrow[d, "*"'] & u \arrow[d, dashed, "*"] \\
v \arrow[r, dashed, "*"'] & w
\end{tikzcd}$$

(a) Confluence

$$\begin{tikzcd}
t \arrow[r] \arrow[d] & u \arrow[d, dashed, "*"] \\
v \arrow[r, dashed, "*"'] & w
\end{tikzcd}$$

(b) Local confluence

$$\begin{tikzcd}
t \arrow[r] \arrow[d] & u \arrow[d, dashed, "\leq 1"] \\
v \arrow[r, dashed, "\leq 1"'] & w
\end{tikzcd}$$

(c) Strong confluence

Figure 2.2.: Diagram for different notions of confluence

$$\begin{tikzcd}
x \arrow[dr, "*"] & & y \arrow[dl, "*"'] \arrow[dr, "*"] & & z \arrow[dl, "*"'] \\
& u \arrow[dr, dashed, "*"] & & v \arrow[dl, dashed, "*"'] & \\
& & w & &
\end{tikzcd}$$

Figure 2.3.: Visual proof of the transitivity of $\equiv$

as $\exists w, u \to_\beta^* w \land v \to_\beta^* w$ is trivially included in $\equiv_\beta$, but it is also reflexive, symmetric, containing $\to_\beta$ and proved transitive by Figure 2.3, so it is the same as $\equiv_\beta$.

From there, we can easily deduce an algorithm to check the convertibility of two strongly-normalising λ-terms: first, reduce the terms to a normal form, and then check whether the normal forms are equal. This algorithm terminates since the inputs are strongly normalising, and always returns the correct answer since normal forms are unique.

Computing a normal form by naïvely applying the rules of β-reduction is very inefficient; moreover, it is also possible to check convertibility without going all the way to normal forms. In this thesis, we will try to find better algorithms for those two questions. In the first part of this thesis, we will see how to compute the normal form of a λ-term more efficiently, while in the second part, we will look at a way to test convertibility of two terms without going all the way down to normal forms.

2.4. Contexts

Another way to specify β-reduction is to use *contexts*, which are terms with a *hole*, written $\square$. They are defined by the following grammar:

$$C ::= \square \mid \lambda x.C \mid C\ t \mid t\ C$$

Then, we define a *filling* operation $C[t]$, which takes a context and a term, and returns a term, which corresponds to C where the hole has been replaced by t:

$$\begin{aligned} \Box[t] &= t & (C\ t_2)[t_1] &= C[t_1]\ t_2 \\ (\lambda x.C)[t] &= \lambda x.C[t] & (t_1\ C)[t_2] &= t_1\ C[t_2] \end{aligned}$$

Once we have contexts, the definition of β-reduction becomes a lot simpler. We first define a rule for the reduction of a redex at the top of the term by:

$$(\lambda x.t_1)\ t_2 \mapsto_\beta t_1[x := t_2]$$

Then, we define $\to_\beta$ to be the *context closure* of $\mapsto_\beta$, that is:

$$C[t_1] \to_\beta C[t_2] \quad \text{if} \quad t_1 \mapsto_\beta t_2$$

The main advantage of contexts is that while the $\mapsto_\beta$ definition will always remain the same, we will see several variants of β-reduction where the definition of contexts is different. In these cases, we only have to specify the new contexts instead of all the rules for $\to_\beta$ given above, which simplifies a lot the definition of other variants.

2.5. Weak, open and strong β-reduction

In the rules given above for β-reduction, contexts allow reduction to happen anywhere inside a term. We call this version of $\to_\beta$ *strong* β-reduction. However, most functional programming languages do not reduce under λ-abstractions. One of the reasons is that this simplifies implementations, as if we see λ-abstractions as functions, compiling a term requires its code to stay the same throughout execution. If we remove the case of λ-abstractions from the definition of contexts as given below, we get another flavour of β-reduction called *open* β-reduction.

$$C ::= \Box \mid C\ t \mid t\ C$$

The most well-known version of β-reduction, which is actually the one used by most programming languages, is *weak* reduction. In this version, λ-abstractions do not appear in contexts like in open reduction, but we also require the term we reduce to be closed.[3] This way, when we perform a step of β-reduction $(\lambda x.t_1)\ t_2 \to_\beta t_1[x := t_2]$, t_2 is closed and thus we do not have to worry about its free variables when performing the substitution.

[3] It is easy to see that the property of being a closed term is preserved by β-reduction.

2.6. Reduction strategies

In the remainder of part I, we will consider weak reduction only, as strong reduction is a more complicated problem which will be studied in Parts II and III. The $\to_\beta$ relation we gave is non-deterministic, and there are several strategies to determine in what order the reductions are to be performed.

The simplest of those strategies is *call-by-name*. In call-by-name, we disallow reducing the argument of a function, that is, we further restrict the evaluation contexts to the following:

$$C ::= \square \mid C\ t$$

In an application $(\lambda x.t)\ u$, each copy of the argument u will then be further reduced after the β-reduction has been performed.

Another strategy is *call-by-value*. In call-by-value, we first define a subset of irreducible terms called *values*, here defined by $v ::= \lambda x.t$. Then, we perform reduction by restricting $\mapsto_\beta$ to a function applied to a value, that is: $(\lambda x.t)\ v \mapsto_\beta t[x := v]$. Furthermore, we may restrict the contexts to force (for instance) a function to be fully evaluated before reducing its argument:

$$C ::= \square \mid C\ t \mid v\ C$$

This choice of contexts ensure that call-by-value reduces the argument before substituting it in the function body. In contrast, call-by-name substitutes it first, which can cause repeated evaluation of the argument if the parameter occurs several times in the function body. For instance, writing $I' = I = \lambda x.x$, consider the term $t = (\lambda y.y\ y)\ (I'\ I)$. With call-by-value, the successive reduction steps will be $t \to (\lambda y.y\ y)\ I \to I\ I \to I$, while with call-by-name, they will be $t \to (I'\ I)\ (I'\ I) \to I\ (I'\ I) \to I'\ I \to I$, where we can see that the reduction of the argument $I'\ I \to I$ has been done twice.

On the other hand, call-by-value will sometimes do a lot of useless work if that argument is never used, while call-by-name will not do anything in that case. An extreme example is $(\lambda x.I)\ \Omega$, where Ω does not terminate: this term terminates in call-by-name with result I after a single reduction step, but does not terminate in call-by-value. More generally, call-by-name has a very strong property: if a λ-term is weakly normalising, then call-by-name will terminate on this term.

Finally, *call-by-need* is a hybrid approach that evaluates function arguments at most once, but only when they are needed. However, it is complicated to define in terms of pure λ-calculus, although it has been done by Ariola et al. [Ari+95] with big-step semantics, and Launchbury [Lau93] with operational semantics. A relatively simple way to define call-by-need is with an abstract machine, so we delay its definition to section 2.8.

2.7. Abstract and virtual machines

To give practical implementations of β-reduction or other computations while remaining at a high level and avoiding trivial details, we will use *abstract machines*. Formally, an abstract machine is simply a reduction relation $\rightsquigarrow$ on a set of *states*. A state represents a point in the computation in more details that can be expressed by a single term. We say that $\rightsquigarrow$ *simulates* a reduction relation $\to$ if we have:

- an *initialisation function* **init**, from terms to states,
- a *readback function* **read**, from states to terms, such that $\mathbf{read}(\mathbf{init}(t)) = t$ for all t,
- the *simulation property*, which states that if $s_1 \rightsquigarrow s_2$, then $\mathbf{read}(s_1) \to^* \mathbf{read}(s_2)$.

When trying to compute the number of reductions done by the machine compared to the number of reductions done in the original program, it is relevant to make the difference between so-called *administrative reductions*, which are reductions $s_1 \rightsquigarrow s_2$ for which $\mathbf{read}(s_1) = \mathbf{read}(s_2)$, and the other reductions, for which $\mathbf{read}(s_1) \to^+ \mathbf{read}(s_2)$. Indeed, there are at most as many non-administrative reductions as there are reductions in the original program, therefore evaluating the number of administrative reductions is necessary to get good complexity guarantees in the final program.

A well-known abstract machines is the *Krivine abstract machine* [Kri07] or KAM, which implements the weak call-by-name λ-calculus. The states of the Krivine machine are composed of three parts: the *code* which is a term; the *environment*, which is a map from variables to *closures*; and the *stack*, which is a list of closures; where a closure is a pair of a term and an environment.

The environment part of a closure is a delayed substitution, and we can read a closure back to a term by the following function:

$$\begin{aligned} \mathbf{readclos}(t, e) &= t[\overline{x} := \overline{u}] \\ &\text{where:} \\ &\qquad \overline{x} = \mathbf{dom}(e) \\ &\qquad \forall i, u_i = \mathbf{readclos}(e(x_i)) \end{aligned}$$

A state $\left(\begin{array}{c|c|c} \text{Code} & \text{Environment} & \text{Stack} \\ t & e & \overline{p} \end{array}\right)$ represents the application of the closure (t, e) to the arguments $\overline{p}$.

$$\begin{array}{c|c|ccc|c|c}
\text{Code} & \text{Environment} & \text{Stack} & & \text{Code} & \text{Environment} & \text{Stack} \\
t_1\ t_2 & e & \pi & \rightsquigarrow_a & t_1 & e & (t_2, e) :: \pi \\
x & e \star (x \mapsto (t, e')) & \pi & \rightsquigarrow_s & t & e' & \pi \\
\lambda x.t_1 & e & p :: \pi & \rightsquigarrow_\beta & t_1 & e \star (x \mapsto p) & \pi
\end{array}$$

Figure 2.4.: The Krivine abstract machine.

This leads to the following definitions of **init** and **read**:

$$\mathbf{init}(t) = \left(\begin{array}{c|c|c} \text{Code} & \text{Environment} & \text{Stack} \\ t & \emptyset & [] \end{array} \right)$$

$$\mathbf{read} \left(\begin{array}{c|c|c} \text{Code} & \text{Environment} & \text{Stack} \\ t & e & \overline{p} \end{array} \right) = \mathbf{readclos}(t, e)\ \overline{\mathbf{readclos}(p)}$$

The Krivine abstract machine is then defined by the union of the three transition relations in Figure 2.4, where each line represents a transition relation. For instance, the first line corresponds to the following transition relation:

$$\left(\begin{array}{c|c|c} \text{Code} & \text{Environment} & \text{Stack} \\ t_1\ t_2 & e & \pi \end{array} \right) \rightsquigarrow_a \left(\begin{array}{c|c|c} \text{Code} & \text{Environment} & \text{Stack} \\ t_1 & e & (t_2, e) :: \pi \end{array} \right)$$

It is easily seen that if $s_1 \rightsquigarrow_a s_2$ or $s_1 \rightsquigarrow_s s_2$, then $\mathbf{read}(s_1) = \mathbf{read}(s_2)$, and if $s_1 \rightsquigarrow_\beta s_2$, then $\mathbf{read}(s_1) \to_\beta \mathbf{read}(s_2)$, thus $\rightsquigarrow$ simulates $\to_\beta$ indeed.

One important property, which we will call the *subterm property*, of the Krivine machine is that, at each point, the code is a subterm of the initial term. There are thus only a finite number of possible values for the code, and we could replace the code with a compiled version. We first define a compiling function, from terms to machine instructions:

$$\begin{aligned}
[\![t_1\ t_2]\!]_\rho &= \mathtt{APP}([\![t_2]\!]_\rho); [\![t_1]\!]_\rho \\
[\![\lambda x.t]\!]_\rho &= \mathtt{ABS}; [\![t]\!]_{x::\rho} \\
[\![x]\!]_\rho &= \mathtt{VAR}(\mathbf{index}_\rho(x))
\end{aligned}$$

Once we have this compiling function, we can define a version of the Krivine abstract machine that works with compiled code:

$$\begin{array}{c|c|ccc|c|c}
\text{Code} & \text{Environment} & \text{Stack} & & \text{Code} & \text{Environment} & \text{Stack} \\
\mathtt{APP}(c_2); c_1 & e & \pi & \rightsquigarrow_a & c_1 & e & (c_2, e) :: \pi \\
\mathtt{VAR}(n) & \overline{p}^n :: (c', e') :: e & \pi & \rightsquigarrow_s & c' & e' & \pi \\
\mathtt{ABS}; c & e & p :: \pi & \rightsquigarrow_\beta & c & p :: e & \pi
\end{array}$$

	Code	Environment	Stack	Dump	Store
	$t_1\ t_2$	e	π	D	H
$\rightsquigarrow_a$	t_1	e	$a :: \pi$	D	$H \star (a \mapsto (t_2, e))$
	x	$e \star (x \mapsto a)$	π	D	$H \star (a \mapsto (t, e'))$
$\rightsquigarrow_l$	t	e'	$[]$	$(a, \pi) :: D$	$H \star (a \mapsto \square)$
	$\lambda x.t_1$	e	$a :: \pi$	D	H
$\rightsquigarrow_\beta$	t_1	$e \star (x \mapsto a)$	π	D	H
	$\lambda x.t_1$	e	$[]$	$(a, \pi) :: D$	$H \star (a \mapsto \square)$
$\rightsquigarrow_s$	$\lambda x.t_1$	e	π	D	$H \star (a \mapsto (\lambda x.t_1, e))$

Figure 2.5.: The lazy Krivine abstract machine.

The definitions are very similar, but we can see here that the analysis of the term is done by the compilation function, and that the compiled code never changes during execution, so that in practice, it could be represented by a pointer inside an array of machine instructions called *opcodes*. We will call such abstract machines *virtual machines*,[4] to show the importance of having static code.

2.8. Call-by-need reduction

As we said before, call-by-need is a reduction strategy that evaluates function arguments at most once, only if they are needed. There are two main ways of thinking about call-by-need: it can either be seen as a *lazy* version of call-by-value, or as a *memoizing* version of call-by-name. When we see call-by-need as a lazy version of call-by-value, instead of requiring the argument of a β-redex to be a value, we only promise we are going to reduce it to a value when it will be needed. Once it it first needed and we have computed the value, all places where the argument was used now have access to this value instead. When we see it as a memoizing version of call-by-name, we remember that the argument we substituted is shared at all the places where we substituted it. When we perform reductions in this argument, the reductions happen in all the places it was substituted at once.

Call-by-need combines the best of both worlds: it avoids computation as much as possible, and only performs computations that are needed to get the result.

[4] The definition of a virtual machine is often given as an abstract machine for which an interpreter exists. Besides being an inappropriate definition for theoretical purposes as it may change over time, most such implementations modify the abstract machine to use compiled code whenever possible for efficiency reasons. Thus, we choose this criterion as the definition of a virtual machine here, as it expresses why we want it. Abstract machines which satisfy the subterm property are often easy to convert into virtual machines, and we will sometimes call them such.

$$
\begin{aligned}
\mathbf{readclos}((t,e),H) &= t[\overline{x} := \overline{u}] \\
\text{where} & \\
\overline{x} &= \mathbf{dom}(e) \\
\forall i, u_i &= \mathbf{readclos}(H(x_i),H)
\end{aligned}
$$

$$
\mathbf{readstack}(t,\overline{p},H) = t\ \overline{\mathbf{readclos}(H(p),p)}
$$

$$
\begin{aligned}
\mathbf{readdump}(t,\pi,[],H) &= \mathbf{readstack}(t,\pi,H) \\
\mathbf{readdump}(t,\pi,(a,\pi') :: D, H \star (a \mapsto \Box)) &= \\
\mathbf{readdump}(\mathbf{readstack}(t,\pi,H \star (a \mapsto t)),\pi',D,H \star (a \mapsto t))
\end{aligned}
$$

$$
\mathbf{read}(t,e,\pi,D,H) = \mathbf{readdump}(\mathbf{readclos}((t,e),H),\pi,D,H)
$$

Figure 2.6.: Readback function for the LazyKAM.

	Code	Stack	Dump	Store	
	$(t_1\ t_2, e)_c$	π	D	H	
$\leadsto_a$	$(t_1, e)_c$	$a :: \pi$	D	$H \star (a \mapsto \mathtt{L}\ (t_2, e))$	
	$(x, e)_c$	π	D	H	
$\leadsto_l$	$e(x)$	π	D	H	
	$(\lambda x.t, e)_c$	π	D	H	
$\leadsto_\lambda$	$(\lambda x.t, e)$	π	D	H	
	a	π	D	$H \star (a \mapsto \mathtt{L}\ (t, e))$	
$\leadsto_{f_1}$	$(t, e)_c$	$[]$	$(a, \pi) :: D$	$H \star (a \mapsto \Box)$	
	a	π	D	$H \star (a \mapsto \mathtt{D}\ v)$	
$\leadsto_{f_2}$	v	π	D	$H \star (a \mapsto \mathtt{D}\ v)$	
	$(\lambda x.t_1, e)$	$a :: \pi$	D	H	
$\leadsto_\beta$	$(t_1, e \star (x \mapsto a))_c$	π	D	H	
	v	$[]$	$(a, \pi) :: D$	$H \star (a \mapsto \Box)$	
$\leadsto_s$	v	π	D	$H \star (a \mapsto \mathtt{D}\ v)$	$v \neq a'$

Figure 2.7.: The lazy Krivine abstract machine, with lazy values.

The easiest way to formally define it is to start with the KAM, the LazyKAM, which shares the evaluation when the code and environment correspond to a given closure (t, e). We then have to extend the definition of the machine to add two more parts: the *store*, which holds bindings from *memory locations* to values, and the *dump*, which remembers what is to be done once we finish computing the value corresponding to a given location.

The rules are shown in Figure 2.5, and are quite similar to those of the KAM: the rule $\leadsto_\beta$ has not changed at all, affecting neither the dump nor the store, while the only change to $\leadsto_a$ is the extra indirection that is added through the store. The more important change is the splitting of the rule $\leadsto_s$ into two rules, $\leadsto_l$ and $\leadsto_s$. Of these two rules, $\leadsto_l$ starts the evaluation of a closure stored on the store, but adds information to the dump to remember to update the value in the store once the evaluation of this term finishes on a value. On the other hand, the rule $\leadsto_s$ saves the result of the evaluation back on the store for when it is needed later, and restores the stack from before the evaluation of this variable.

We could give a readback function from states to terms to establish that the LazyKAM does simulate β-reduction, but it is actually easier to prove that it simulates the KAM itself! To go from a state for the LazyKAM to a state for the KAM, we simply concatenate all the parts of the stack that are seen in the dump, and we replace each memory location in the state of environment by the closure it held *at the moment it was first created*. We also have the property that for each contents (t', e') of a memory location, if it held (t, e) when it was created, then for all π, we have $(t, e, \pi) \leadsto^* (t', e', \pi)$ for the KAM, meaning that we memoized the result of a computation. With this invariant in mind, we can see that the LazyKAM simulates the KAM, and thus β-reduction. Besides, this really makes explicit that this machine is indeed a memoized version of call-by-need.

For this machine, the initial state for a term t is t as code, and an empty environment, stack, dump and store, while a final state is when the code is of the form $\lambda x.t$, and both the stack and the dump are empty. The readback function is a bit complex, and shown in Figure 2.6.

Another slightly different way to present the LazyKAM is by separating *lazy values* from already-evaluated values, in our case the λ-abstractions. This way is easier to extend, in particular to support the constructors extension.

In that case, the store contains either lazy values $\texttt{L}\ (t, e)$, or computed results $\texttt{D}\ v$, where a value v is either a memory address a or a λ-abstraction $(\lambda x.t, e)$. We also merge the code and environment together, with the code now being either a value v, or a pair between a term and en environment $(t, e)_c$.[5] The resulting machine is presented in Figure 2.7. By separating the lazy values from the others, we can clearly see how each lazy value is only evaluated once.

[5] We use $(t, e)_c$ instead of (t, e) to be able to make the distinction when the value itself is a λ-abstraction $(\lambda x.t, e)$.

2.9. Weak head reduction

Weak head reduction is a subset of β-reduction, aimed at exposing the shape of the normal form of a term. Weak head reduction is simply defined by the H-closure of $\mapsto_\beta$, with the following definition for the *weak head contexts* H:

$$H ::= \square \mid H\ t$$

A term is said to be in *weak head normal form* if it is a normal form for weak head reduction. Terms in weak head normal form are either abstractions $\lambda x.t$, or a free variable x applied to any number of arguments $x\ \overline{t}$. Weak head reduction has contexts identical to call-by-name reduction shown above, but it is treated differently because we want to consider it as a separate kind of reduction, and not as simply an evaluation strategy, so that we can combine it with other kinds of reductions on open terms. Moreover, weak head reduction has a property concerning convertibility which will be crucial in part III. Indeed, when two terms are in weak head normal form, they are convertible if and only if one of the following is true:

- both terms are of the form $\lambda x.t$ and $\lambda x.u$ (up to α-conversion), and t and u are convertible,
- both terms are of the form $x\ \overline{t}^n$ and $x\ \overline{u}^n$, with the same variable x and the same number of arguments n, and for all i, t_i and u_i are convertible.

Thus, we can see why we say that weak head reductions exposes the shape of the normal form of a term: once a term is in weak head normal form, its global shape (either an abstraction or a free variable applied to some number of arguments) can no longer change.

2.10. Extensions of the λ-calculus

2.10.1. Defined constants

The λ-calculus with defined constants is one which will be very important in this work. Here, we extend the terms with *constants*:

$$t ::= \cdots \mid c$$

We have a finite number of constants c, and a static definition d_c for each constant. Besides, defined constant must be closed, and they must form an acyclic graph (each constant can only reference other constants defined before it).

We add a new reduction rule $\rightarrow_\delta$, defined by the context-closure of $\mapsto_\delta$ defined as $c \mapsto_\delta d_c$ for each constant c, calling *unfolding* the constant c.

Thus, each constant can be unfolded to its definition whenever it appears, allowing to modularise the code of a λ-calculus program.

2.10.2. Constructors and pattern matching

We can also extend the λ-calculus with *data constructors* and *pattern matching* (more precisely, shallow pattern matching). Here, the terms are extended as follows:

$$t ::= \cdots \mid T\ \bar{t} \mid \textbf{match}\ t\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow t}\ \textbf{end}$$

Constructors have a tag T and any number of subterms, while pattern matching matches one term depending on its constructor. When a constructor has more than one subterm, we usually group them inside parentheses, with commas separating them, while the several cases in pattern matching are separated by |.

Contexts and head contexts are likewise extended:

$$\begin{aligned} C &::= \cdots \mid T\ (\bar{t}, C, \bar{t}) \mid \textbf{match}\ C\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow t}\ \textbf{end} \\ &\quad\ \mid \textbf{match}\ t\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow t} \mid T\ \overline{x} \Rightarrow C \mid \overline{T\ \overline{x} \Rightarrow t}\ \textbf{end} \\ H &::= \cdots \mid \textbf{match}\ H\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow t}\ \textbf{end} \end{aligned}$$

As usual, the reduction rule $\rightarrow_\iota$ is the context-closure of $\mapsto_\iota$ defined below:

$$\textbf{match}\ S\ \overline{u}^n\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow t} \mid S\ \overline{y}^n \Rightarrow v \mid \overline{T\ \overline{x} \Rightarrow t}\ \textbf{end} \mapsto_\iota v[\overline{y} := \overline{u}]$$

This reduction rule selects the case with the correct tag, and replaces the variables $\overline{y}$ by the arguments of the constructor $\overline{u}$. It also requires $\overline{y}$ and $\overline{u}$ to have the same length, for the substitution to be defined.

With the introduction of these terms, *stuck* terms can also appear. These terms allow no reduction to happen, but are not in the form we expect for values. To avoid them, we will assume our terms are *well-typed*. Normally, this is in reference to some specific type system, but we will not consider this here. Instead, we will simply assume that if we have a term u and $u \rightarrow^* v$, then none of the following *forbidden subterms* appear in v:

$$\begin{aligned} F ::= {} & (T\ \bar{t})\ u \mid \textbf{match}\ \lambda x.t\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow u}\ \textbf{end} \\ & \mid \textbf{match}\ S\ \bar{t}\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow u}\ \textbf{end} && (S \notin \overline{T}) \\ & \mid \textbf{match}\ S\ \overline{u}^n\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow t} \mid S\ \overline{y}^m \Rightarrow v \mid \overline{T\ \overline{x} \Rightarrow t}\ \textbf{end} && (n \neq m) \end{aligned}$$

Each of these subterms corresponds to an operation the type system should have prevented: applying a constructor, matching on a λ-abstraction, or a match operation where the constructor being matched and the pattern matching case do not make the same assumptions about the name of the constructor or its arity.

2.10.3. Fixpoints

The third extension we will consider is the presence of *fixpoints*. It is an extension of constructors and allows us to perform computations over recursive data structures in a typed setting.[6] We extend terms and contexts as shown below, with n a *positive* integer:

$$\begin{aligned} t &::= \cdots \mid \mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end} \\ C &::= \cdots \mid \mathbf{fix}_n\ f\ \overline{x}^n := C\ \mathbf{end} \\ H &::= \cdots \mid (\mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end})\ \overline{u}^{n-1}\ H \end{aligned}$$

We also extend $\mapsto_\iota$ with the following rule:

$$\begin{aligned} &(\mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end})\ \overline{u}^{n-1}\ (T\ \overline{v}) \\ \mapsto_\iota\ &(\lambda \overline{x}^n.t[f := \mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end}])\ \overline{u}^{n-1}\ (T\ \overline{v}) \end{aligned}$$

The idea behind this rule is to replace f with the definition of the fixpoint inside its body, but only perform this substitution if the n-th argument has a constructor as its head. With the *guard condition* imposed by Coq, which ensures recursive calls are decreasing in their n-th arguments, this is enough to make such fixpoint definitions strongly normalising. Worthy of note is also the way we have extended head contexts to account for reduction in the n-th argument of a fixpoint in order to get a constructor as the head.

We also need to extend the forbidden subpatterns to account for the typing of fixpoints, since the last argument of a fixpoint needs to be a constructor, and partially-applied fixpoints are not constructors either:

$$\begin{aligned} F ::=\ & \ldots \\ & \mid \mathbf{match}\ (\mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end})\ \overline{u}^m\ \mathbf{with}\ \overline{T\ \overline{y} \Rightarrow v}\ \mathbf{end} && (m < n) \\ & \mid (\mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end})\ \overline{u}^{n-1}\ (\lambda y.v) \\ & \mid (\mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end})\ \overline{u}^{n-1}\ ((\mathbf{fix}_m\ f\ \overline{y}^m := v\ \mathbf{end})\ \overline{w}^p) && (p < m) \end{aligned}$$

[6] In an untyped setting, a fixpoint combinator can be defined, such as Curry's Y combinator $\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$

3. Formal verification and Coq

3.1. A first Coq proof

In the introduction, we mentioned type theory, a possible axiomatisation of mathematics, and Coq, a proof assistant based on the calculus of constructions, an extension of type theory. Let us first consider what a proof assistant is. A proof assistant is a software tool which allows one to formalise and prove mathematical[1] theorems, with the computer being able to verify that the user-supplied proofs are correct.

For instance, the example below defines the divisibility relation `divides p n` over $\mathbb{N}$, which corresponds to $p \mid n$ in mathematical notation, as $\exists q \in \mathbb{N}.\ n = qp$. It then shows that $\forall n \in \mathbb{N}, n \mid n$ and that $\forall p, n, m \in \mathbb{N}, p \mid n \Rightarrow p \mid mn$.

```
Definition divides (p: nat) (n: nat) := exists q: nat, n = q * p.

Theorem divides_refl:
  forall n, divides n n.
Proof.
  intros. exists 1. simpl. auto.
Qed.

Theorem divides_multiple:
  forall p n m, divides p n -> divides p (m * n).
Proof.
  intros p n m D. destruct D as [q EQ]. subst n.
  exists (m * q). apply Nat.mul_assoc.
Qed.
```

Knowing this, what does it mean for the proof assistant? First, that we need to state the theorems and properties we want to prove. This is done by communicating with the software in a given language, called the *term language* in Coq; it is the language used to write definitions and the statements of theorems. Unlike Coq, in some proof assistants based on *first-order logic*, there is a clear distinction between the expressions, designating objects, and the propositions, used to state theorems. Yet, both these languages are

[1] When we say mathematical here, we mean anything that is both purely abstract and can be obtained from formal deduction rules as mentioned in the introduction, which includes most of logic, mathematics and computer science, but even some things in the domain of philosophy: for instance, Gödel's ontological proof of existence of God has been formalised in Coq [BWP17].

closely related: except in rare cases, propositions are statements about expressions, and it makes little sens to say that they are different languages entirely. Therefore, we will unify them even in this case and call the union of them the term language. For instance, in the above example, `divides p n`, `m * n` and `forall n, divides n n` are all part of the term language.

Second, as the user supplies the proof in proof assistants (unlike theorem provers, which try to automatically prove the statements that are given to them), we need to be able to communicate them to the proof assistant. In Coq, as in all proof assistants based on *pure type systems*, a particular kind of type theory, proofs can be written directly in the term language. However, it is quite hard to directly write proofs in this form, so Coq, like many proof assistants,[2] exposes another language called the *tactic language*, which allows the user to write proofs in an interactive manner, and even to write some code which automates repetitive parts of the proofs,[3] to write tactics performing uninteresting computations, or even to use external theorem provers and to import their proofs back into the proof assistant! In Coq, `Proof.` introduces a proof written in the tactic language and `Qed.` ends it: for instance, `intros.` or `destruct D as [q EQ].` are both part of the tactic language.

In Coq, the term language is called Gallina, while the tactic language is called Ltac, with a newer version of it called Ltac2. Being based on the calculus of inductive constructions, an extension of pure type systems, Coq terms can express computation, allowing to formalise facts about programs that can be run within Coq.[4] For instance, we can define the factorial function as a recursive function, show that $\forall n, m, n \leq m \to n! \mid m!$, and even compute the value of $5!$:

```
Fixpoint factorial (n: nat) : nat :=
  match n with
  | O => 1
  | S n' => n * factorial n'
  end.

Theorem divides_factorial:
  forall n m, n <= m -> divides (factorial n) (factorial m).
Proof.
  Local Opaque mult.
  induction m as [ | m']; intros LE.
  - (* Case m = 0 *)
```

[2] But not all: for instance, Agda does not, and instead supplies interactive tools for the user to directly write their proof in the term language.

[3] In fact, some proof assistants such as HOL4 even use a general-purpose programming language as the tactic language for this exact reason.

[4] However, it is important to note that Gallina is *not* a Turing-complete language, as all Gallina programs terminate. This is necessary to ensure the correctness of Coq, and does not prevent proving facts about non-terminating programs, although these programs have to be encoded in some way (fuel, monads, deep embeddings, ...) to express them.

```
    assert (n = 0) by lia. subst n. exists 1; reflexivity.
  - (* Case m = S m' *)
    apply Nat.le_succ_r in LE. destruct LE.
    + (* Case n <= m' *)
      simpl. apply divides_multiple. apply IHm'. auto.
    + (* Case n = m *)
      subst n. apply divides_refl.
Qed.

Compute (factorial 5). (* = 120 : nat *)
```

3.2. Proofs by computational reflection

A crucial part of Coq is that terms and propositions exist only up to computation: for instance, $2 + 2 = 4$ and $4 = 4$ are exactly the same theorem, and reflexivity of equality $(\forall x, x = x)$ is a valid proof of $2+2 = 4$. This also means that proofs that can be fully done by computation are simple, and all the steps of the computation need not to be encoded inside the proof itself, as we can see in the following Coq example: while we could give a standard proof using various theorems, a simpler proof is to use `reflexivity`, which will produce a proof term simply equal to `eq_refl 4`, the proof of `4 = 4`, which is also a proof of `2 + 2 = 4` up to computation.

```
Remark two_plus_two: 2 + 2 = 4.
Proof.
  (* By deduction *)
  rewrite Nat.add_succ_l. rewrite Nat.add_succ_l. rewrite Nat.add_0_l. auto.
Restart.
  (* By computation *)
  reflexivity.
Qed.

Print two_plus_two. (* = eq_refl : 2 + 2 = 4 *)
```

This is akin to the distinction between verification and proof expressed by Poincaré more than a century ago [Poi94]: "La « vérification » diffère précisément de la véritable démonstration, parce qu'elle est purement analytique et parce qu'elle est stérile." — *verification differs from true demonstration, because it is purely analytical and sterile.* By "sterile", Poincaré meant that no ideas are necessary in computation: it is a simple series of steps. In modern terms, we might say that it is a decidable operation, and, furthermore, by a simple algorithm.

Formally, this is encoded as the *convertibility rule*: if t has type A, and if A and B are convertible, then t has type B as well. This leads to a whole new sort of proofs, related to computer-aided proofs in mathematics, called *proofs by reflection*.

For those proofs, we want to prove a statement P x, where P is a property of a typically large object x; for instance, graph 4-colourability in the proof of the 4-colour theorem. Instead of directly doing that, as it would require writing and having the proof assistant check a large proof, we instead write a program prove_P, and establish that forall x, prove_P x = true -> P x. Being independent of x, this property is typically simpler to prove than P x would be directly. However, we can then use it to establish it is enough to prove prove_P x = true, and directly prove this by reflexivity, the left-hand side performing the computation and proof for us, without needing to check anything again! A simple example, for divisibility, is shown below. Note how, in particular, we did not need to compute the divisibility witness 17 ourselves in the proof by reflection: it comes from the proof of dec_divides_sound.

```
Definition dec_divides (p: nat) (n: nat) : bool := n mod p =? 0.

Lemma dec_divides_sound: forall p n, dec_divides p n = true -> divides p n.
Proof.
  intros p q H. apply Nat.eqb_eq in H. apply Nat.Div0.div_exact in H.
  exists (q / p). lia.
Qed.

Remark divides_13_221: divides 13 221.
Proof.
  (* By deduction *)
  exists 17; auto.
Restart.
  (* By reflection *)
  apply dec_divides_sound. reflexivity.
Qed.

Print divides_13_221. (* = dec_divides_sound 13 221 eq_refl *)
```

Proofs by reflection can involve large amounts of computation. Although more narrowly-scoped methods of proving convertibility exist for this special case, this still explains why we care about the efficiency of convertibility. Besides, they are far from the only use case for an efficient convertibility test! Indeed, almost every proof step will require some convertibility checking to ensure it is correct; and, more importantly, proof search by Ltac or other tactic languages require convertibility tests, to know what is possible to do and what is not. Besides, in this case, it is important to fail as fast as possible, to ensure proof search remains efficient: we no longer only care about checking that two terms are convertible, but we want to decide if they are.

Thus, we can see how convertibility tests are a critical part of proof search and verification; moreover, it is a well-defined and isolated part of it, and as such is very amenable to being formally verified.

3.3. Trusted code base

As we discussed concerning the tactic language, we can go all the way to using external tools to prove theorems. However, a question remains: what code do we need to trust to be correct if we want to be sure a given theorem, whose proof was checked by a proof assistant, is true? First, we need to trust that our theorem is correctly stated, and that the definitions used in its statement are correct. While this seems obvious, there might already be errors there: if we prove that $2 + 3 \times 4 = 14$, but the user incorrectly thinks this should be understood as $(2 + 3) \times 4 = 14$, we already have a problem.

Another thing we need to trust is the system of axioms and deduction rules of the proof assistant, since consistency proofs for these logics are difficult. As some systems are inconsistent and allow to prove any theorems, our proof would still be correct inside the axiom system used; but it wouldn't state anything useful. Following the chain of implementation, we need to trust the critical part of the proof assistant, which is often the part that checks that proofs are correct. Crucially, this is not the entire proof assistant: for instance, almost nothing in the tactic language needs to be trusted, as it simply translates the language used to write the proof to internal proof terms.

Further down the chain, we also need to trust the implementation of the programming language used to write the proof assistant (and even its compiled binary, as shown by Thompson [Tho84]), the operating system it runs on, the firmware and the hardware of the machine we use to run it, even that the hardware is sufficiently shielded from external influence such as cosmic rays...

As outlined above, there are many parts that need to be relied on, and only a few directly related to the proof assistant itself. Therefore, when trying to formally verify a proof assistant, we need to decide what the scope of the trusted code base is. In our case, the only part we are really interested in is the trusted code base of the proof assistant itself, and not everything the assistant relies on to run![5]

[5] However, there are other projects which are interested in formally verifying other things, for instance CompCert and CakeML for programming languages, SEL4 for operating systems, or Kami for hardware; which do not directly interest us.

Part II.

Strong call-by-need

4. Big-step semantics for strong call-by-need

In this part, we aim for an implementation of strong call-by-need for the λ-calculus. However, call-by-need semantics are often more complicated than corresponding call-by-value or call-by-name semantics, due to them often needing mutable state.

Besides, there are two ways of seeing call-by-need as variations of other evaluation orders of λ-calculus: one is a lazy call-by-value, the other is a memoizing call-by-name. In our case, we will express our call-by-need semantics by memoizing call-by-name semantics. Thus, we start with strong call-by-name semantics for the λ-calculus, in extension of the weak call-by-name semantics we saw in section 2.6.

4.1. Starting with call-by-name

The first step towards having a strong call-by-name semantics for the λ-calculus is specifying the normal forms. A term is in normal form if, and only if, it does not contain a subterm of the form $(\lambda x.t)\ u$. Thus, it is either a variable x, a λ-abstraction $\lambda x.t$, in which case t must also be in normal form, or an application $t\ u$, where t and u must be in normal form, and t must not be a λ-abstraction.

We thus get a grammar for normal forms, with i the *inert* normal forms, those which are not λ-abstractions, and r the general normal forms.

$$\begin{aligned} \text{Normal forms } r &::= i \mid \lambda x.r \\ \text{Inert forms } i &::= x \mid i\ r \end{aligned}$$

Now that we know what normal forms are, we can get a small-step semantics for strong call-by-name.

In call-by-name, we only reduce the right-hand side of applications, if the left-hand side is already in normal form and is not a λ-abstraction. Besides, if the left-hand side is a λ-abstraction, we reduce the redex instead of reducing under the abstraction. Indeed, if we first reduced below the abstraction, we might reduce some redexes that would otherwise disappear by reducing the outer redex first.[1] Thus, we can restrict the reduction contexts

[1] Consider for instance the following term (where t is any term containing a redex): $(\lambda x.x\ t)\ (\lambda y.z)$. If we reduced under the λ-abstraction before reducing the outermost redex, we would be able to

to the following, where C_r is the reduction context, and C_i the inert contexts, those which are not λ-abstractions. We see that these contexts are very similar in structure to the definitions of normal forms. This is intended, as we have chosen those contexts precisely to get the unique decomposition theorem (since we wanted to specify an evaluation order), and to prove this unique decomposition theorem, naturally by induction on the structure of the term, the definitions have to be close enough so that we can use the induction hypothesis.

$$\begin{aligned} C_r &::= C_i \mid \lambda x.C_r \\ C_i &::= \Box \mid C_i\ t \mid i\ C_r \end{aligned}$$

From this definition, we can check that each term is either a normal form r, or can be written in a unique way as $C_r[(\lambda x.t)\ u]$, so the small-step semantics we get by restricting β-reduction to those contexts C_r is deterministic. This reduction order is known as the *normal evaluation order*, also called the *leftmost-outermost evaluation order*: at each step, we reduce the leftmost redex, and in the case of overlapping redexes, we reduce the outermost one. This evaluation order is also famous because it is normalising: reduction always terminates if the original term is weakly normalising[Bar67].

Next, we can derive big-step semantics from this small-step semantics. We will define two mutually recursive relations, $\Downarrow_{\mathtt{s}}$ and $\Downarrow_{\mathtt{d}}$. In $\Downarrow_{\mathtt{s}}$, which we call *shallow* reduction, the possible values are $i \mid \lambda x.t$, and in $\Downarrow_{\mathtt{d}}$, which we call *deep* reduction, they are $r = i \mid \lambda x.r$. Note how only the λ-abstractions differ between both kinds of values and inert terms are shared: this will allow us to share parts of the rules below between them.

These relations actually express that v is the normal form for t in the C-closure of $\mapsto_\beta$, where C is C_i for $\Downarrow_{\mathtt{s}}$ and C_r for $\Downarrow_{\mathtt{d}}$.

The big-step semantics we obtain that way are as follows, where f means either $\mathtt{s}$ or $\mathtt{d}$:[2]

$$\textsc{Var}\ \frac{}{x \Downarrow_f x} \qquad \textsc{Lam-S}\ \frac{}{\lambda x.t \Downarrow_{\mathtt{s}} \lambda x.t} \qquad \textsc{Lam-D}\ \frac{t \Downarrow_{\mathtt{d}} r}{\lambda x.t \Downarrow_{\mathtt{d}} \lambda x.r} \qquad \textsc{App-}\lambda\ \frac{t_1 \Downarrow_{\mathtt{s}} \lambda x.t_3 \qquad t_3[x := t_2] \Downarrow_f v}{t_1\ t_2 \Downarrow_f v}$$

$$\textsc{App-I}\ \frac{t_1 \Downarrow_{\mathtt{s}} i \qquad t_2 \Downarrow_{\mathtt{d}} r}{t_1\ t_2 \Downarrow_f i\ r}$$

To show a bit more how we obtain those rules, consider the relation $\Downarrow_{\mathtt{s}}$, where we want to get a normal form for the C_i-closure of $\mapsto_\beta$. Consider a term t, and let us see the

reduce redexes occurring inside t, as x is itself in normal form. However, by reducing the outer redex first, the term becomes $(\lambda y.z)\ t$, where the only reduction allowed gives us z where t has completely disappeared, thus making any reduction inside t eventually useless.

[2] f stands for *flag*, as in a Boolean flag that means whether to reduce shallowly or deeply.

possible ways we have to write it. If t is an abstraction $\lambda x.t$ or a variable x, then it is already a normal form, and the normal form of t is equal to t, as seen in the Var and Lam-S rules above. Otherwise, t is an application $t_1\ t_2$. Since, by definition of C_i, we can reduce t_1 until it is in normal form, and since we have uniqueness of the decomposition into a redex and a context while t_1 is not a normal form, there exists a way to decompose t as a context and a redex inside t_1, meaning no other reduction is possible in t. Thus, t has exactly the same normal form as $u\ t_2$, where u is the normal form of t_1 (and t has no normal form if t_1 has none). There are two possibilities for u: either it is of the form $\lambda x.t_3$, in which case the only possible reduction is the reduction of the $(\lambda x.t_3)\ t_2$ redex, and the normal form of t is the same as the one of the result of this reduction, and we get the App-λ rule. The other possibility is for u to be of the form i, and in this case the only possible reductions in t are those inside t_2 until it is of the form r. At this point, t becomes of the form $i\ r$ and is a normal form, resulting in the rule App-I. We can perform a similar analysis for the $\Downarrow_{\mathsf{d}}$ case, where the only difference is the handling of λ-abstractions, under which we perform reductions, thus leading to the presentation above where we share the rules for the two reduction relations when they are similar.

Next, we try to avoid handling substitutions in the application rule; instead, we will use an environment mapping variables to terms, themselves with environments, as in the KAM. Sometimes, a variable does not correspond to an argument of a function but is instead a free variable (due to the Lam-D rule), and we represent this in the environment by mapping the variable to a variable name.

$$\begin{aligned} e &::= \emptyset \mid e \star (x \mapsto b) \\ b &::= x \mid (t, e) \end{aligned}$$

We can interpret a binding b as a term using the following readback functions, as in abstract machines:

$$\begin{aligned} \mathbf{read}(x) &= x \\ \mathbf{read}((t,e)) &= t[\overline{x} := \overline{u}] \\ \overline{x} &= \mathbf{dom}(e) \\ \forall i, u_i &= \mathbf{read}(e(x_i)) \end{aligned}$$

We now define a new evaluation relation $e \vdash t \Downarrow_f v$ with the property that if $e \vdash t \Downarrow_f v$, then $\mathbf{read}(t,e) \Downarrow_f v$. The definition is straightforward from the definition of $\Downarrow_f$, and we obtain the rules in Figure 4.1. We can also see that rule Lam-D maps x to a fresh variable y instead of using x, as x might be free in the environment e and we need to avoid accidental capture.

$$\textsc{Var-F}\ \frac{e(x) = y}{e \vdash x \Downarrow_f y} \qquad \textsc{Var-C}\ \frac{e(x) = (t, e') \qquad e' \vdash t \Downarrow_f v}{e \vdash x \Downarrow_f v}$$

$$\textsc{Lam-S}\ \frac{}{e \vdash \lambda x.t \Downarrow_{\mathtt{s}} (\lambda x.t, e)} \qquad \textsc{Lam-D}\ \frac{e \star (x \mapsto y) \vdash t \Downarrow_{\mathtt{d}} r \qquad y \text{ fresh}}{e \vdash \lambda x.t \Downarrow_{\mathtt{d}} \lambda y.r}$$

$$\textsc{App-}\lambda\ \frac{e \vdash t_1 \Downarrow_{\mathtt{s}} (\lambda x.t_3, e') \qquad e' \star (x \mapsto (t_2, e)) \vdash t_3 \Downarrow_f v}{e \vdash t_1\ t_2 \Downarrow_f v} \qquad \textsc{App-I}\ \frac{e \vdash t_1 \Downarrow_{\mathtt{s}} i \qquad e \vdash t_2 \Downarrow_{\mathtt{d}} r}{e \vdash t_1\ t_2 \Downarrow_f i\ r}$$

Figure 4.1.: Rules for $e \vdash t \Downarrow_f v$.

4.2. From call-by-name to call-by-need

From our big-step semantics of strong call-by-name, we can express call-by-need semantics using memoization. However, with two reduction relations, naïve memoization of the same derivations is not sufficient. For instance, consider the following term t, where u is a term depending on the value of x and taking a long time to compute:

$$t = \lambda x.(\lambda y.\lambda z.y)\ u$$

Here, computing $t\ v$ returns a constant function that always returns the value of evaluating u in the environment $(x \mapsto v)$. However, this takes a long time to compute; and with the current semantics, we will independently memoize the results r_1 given by $(x \mapsto v) \vdash u \Downarrow_{\mathtt{s}} r_1$ and r_2 given by $(x \mapsto v) \vdash u \Downarrow_{\mathtt{d}} r_2$. This is unfortunate, because these values are very closely related: in particular, if r_1 is inert, then $r_2 = r_1$.

Our strategy to deal with this is to give explicit *transfer lemmas* relating the result of shallow and deep reductions. One of these lemmas is related to inert terms:

Lemma 4.1 (Transfer lemma, inert terms) *For all terms t, environments e and* inert *terms i, we have:*

$$(e \vdash t \Downarrow_{\mathtt{s}} i) \Leftrightarrow (e \vdash t \Downarrow_{\mathtt{d}} i).$$

This lemma is straightforward to prove by induction over both derivations, and allows the memoization to work independently of which reduction relation was used. However, the case where the results are λ-abstractions is more complicated. We still easily compute the deep form from the shallow form, as shown in the following lemma:

Lemma 4.2 (Transfer lemma, λ-abstractions, shallow-to-deep) *For all terms t, u, environments e, e', if $e \vdash t \Downarrow_{\mathtt{s}} (\lambda x.u, e')$, then for all values v we have:*

$$(e \vdash t \Downarrow_{\mathtt{d}} v) \Leftrightarrow (e' \vdash \lambda x.u \Downarrow_{\mathtt{d}} v).$$

In other words, once we have the result from shallow reduction, we only have to reduce it using deep evaluation to get the normal form, instead of starting from the beginning!

Unfortunately, computing the shallow form from the deep one does not work so well. Indeed, when we have $e \vdash t \Downarrow_{\mathtt{d}} \lambda x.r$, the original form of the abstraction is lost and only the normal form $\lambda x.r$ remains.[3] However, we have already computed it, so the only change we need is to remember it!

Thus, we modify the rules of our strong-call-by-name semantics to preserve this information, by making the results of deep reduction be of the form $r \mid (\lambda x.t, e, \lambda x.r)$, with the first two elements being the original λ-abstraction and its environment. For this, we need to redefine the two rules Lam-D and App-I, and we define the following function **nf** which extracts the original result of deep reduction from the new results:

$$\begin{aligned} \mathbf{nf}(i) &= i \\ \mathbf{nf}(\lambda x.t, e, \lambda x.r) &= \lambda x.r \end{aligned}$$

$$\textsc{Lam-D'}\quad \frac{e \star (x \mapsto y) \vdash t \Downarrow_{\mathtt{d}} r \qquad y \text{ fresh}}{e \vdash \lambda x.t \Downarrow_{\mathtt{d}} (\lambda x.t, e, \lambda y.\mathbf{nf}(r))}$$

$$\textsc{App-I'}\quad \frac{e \vdash t_1 \Downarrow_{\mathtt{s}} i \qquad e \vdash t_2 \Downarrow_{\mathtt{d}} r}{e \vdash t_1\ t_2 \Downarrow_{f} i\ (\mathbf{nf}(r))}$$

Now that this change is done, we have the remaining transfer lemma:

Lemma 4.3 (Transfer lemma, λ-abstractions, deep-to-shallow) *For all terms t, u, normal forms r, environments e, e', if $e \vdash t \Downarrow_{\mathtt{d}} (\lambda x.u, e', \lambda x.r)$, then we have:*

$$e \vdash t \Downarrow_{\mathtt{s}} (\lambda x.u, e').$$

Once this is done, we only have to extend the semantics to express memoization explicitly. For this, we use a store containing lazy values or their result (either shallow or deep), and function application allocates a new lazy value. Thus, the value of a memory location can be one of:

[3] While it would be possible to use the normal form instead of the original form, this is not compatible with our condition of never reducing under an abstraction before applying it allowing compatibility with compiled code. Besides, even if we dropped this requirement and wanted to use the normal form, we would need to handle terms with sharing inside the *input* instead of the output only, as sharing introduced by the reduction under the abstraction would be otherwise lost!

App-λ
$$\frac{e, m_1 \vdash t_1 \Downarrow_{\mathtt{s}} \mathtt{S}\ (\lambda x.t_3, e'), m_2 \qquad e' \star (x \mapsto a), (a \mapsto \mathtt{L}\ (t_2, e)) \star m_2 \vdash t_3 \Downarrow_f v, m_3}{e, m_1 \vdash t_1\ t_2 \Downarrow_f v, m_3}$$

App-I
$$\frac{e, m_1 \vdash t_1 \Downarrow_{\mathtt{s}} \mathtt{I}\ i, m_2 \qquad e, m_2 \vdash t_2 \Downarrow_{\mathtt{d}} r, m_3}{e, m_1 \vdash t_1\ t_2 \Downarrow_f \mathtt{I}\ (i\ (\mathbf{nf}(r))), m_3}$$

Lam-S
$$\frac{}{e, m \vdash \lambda x.t \Downarrow_{\mathtt{s}} \mathtt{S}\ (\lambda x.t, e), m}$$

Lam-D
$$\frac{e \star (x \mapsto a), (a \mapsto \mathtt{I}\ x) \star m_1 \vdash t \Downarrow_{\mathtt{d}} r, m_2}{e, m_1 \vdash \lambda x.t \Downarrow_{\mathtt{d}} \mathtt{D}\ (\lambda x.t, e, \lambda x.(\mathbf{nf}(r)))), m_2}$$

Var
$$\frac{m_1 \vdash e(x) \rightsquigarrow_f v, m_2}{e, m_1 \vdash x \Downarrow_f v, m_2}$$

Force-Lazy
$$\frac{e, (a \mapsto \square) \star m_1 \vdash t \Downarrow_f v, (a \mapsto \square) \star m_2}{(a \mapsto (t, e)) \star m_1 \vdash a \rightsquigarrow_f v, (a \mapsto v) \star m_2}$$

Force-I
$$\frac{m(a) = \mathtt{I}\ i}{m \vdash a \rightsquigarrow_f \mathtt{I}\ i, m}$$

Force-DS
$$\frac{m(a) = \mathtt{D}\ (\lambda x.t, e', \lambda x.r)}{m \vdash a \rightsquigarrow_{\mathtt{s}} \mathtt{S}\ (\lambda x.t, e'), m}$$

Force-DD
$$\frac{m(a) = \mathtt{D}\ (\lambda x.t, e', \lambda x.r)}{m \vdash a \rightsquigarrow_{\mathtt{d}} \mathtt{D}\ (\lambda x.t, e', \lambda x.r), m}$$

Force-SS
$$\frac{m(a) = \mathtt{S}\ (\lambda x.t, e')}{m \vdash a \rightsquigarrow_{\mathtt{s}} \mathtt{S}\ (\lambda x.t, e'), m}$$

Force-SD
$$\frac{e, (a \mapsto \square) \star m_1 \vdash \lambda x.t \Downarrow_{\mathtt{d}} v, (a \mapsto \square) \star m_2}{(a \mapsto \mathtt{S}\ (\lambda x.t, e)) \star m_1 \vdash a \rightsquigarrow_{\mathtt{d}} v, (a \mapsto v) \star m_2}$$

Figure 4.2.: Big-step rules for strong call-by-need.

- $\mathtt{L}\ (t, e)$, a lazy value,
- $\mathtt{I}\ i$, an inert value,
- $\mathtt{S}\ (\lambda x.t, e)$, the value of a λ-abstraction whose normal form has not yet been computed,
- $\mathtt{D}\ (\lambda x.t, e, \lambda x.r)$, the value of a λ-abstraction with normal for $\lambda x.r$,
- and $\square$, a special symbol meaning the value is currently being computed.

We also introduce an inductive relation $m_1 \vdash a \rightsquigarrow_f v, m_2$ which computes the actual value v corresponding to the memory location a, forcing delayed computations if needed, in order to deduplicate the rules concerning variables. The rules concerning applications and λ-abstractions are mostly unchanged, while the rules concerning variables apply the transfer lemmas if necessary. The rules thread the store through the evaluations, and are shown in Figure 4.2.

Here, we can see that the rules App-I and Lam-S remain the same, except for the explicit tagging of each type of values and the threading of the store. The rules App-λ and Lam-D allocate a fresh cell for the variable besides adding it to the environment, but the rules stay otherwise identical as well. The rules that really change are the ones concerned with variables. Var factorises the next rules along the memory location, which is the part common to all rules. Force-Lazy replaces the old Var-C rule, by evaluating the lazy value, and then updating the environment to mark the value as evaluated. During computation, the value is replaced by a marker $\square$ to explicitly show that this value cannot be accessed during that time, which we can prove by showing our store always remains acyclic. After the value is computed, the environment is updated to store the new value and the value is returned. The rule Force-I subsumes the previous rule Var-F, by handling both free variables and results of lazy values which were inert. The four remaining rules handle getting the result of shallow or deep reduction of a variable when the store already contains a shallow or deep result. Two of these rules, Force-SS and Force-DD are very simple, and simply return the stored value when it corresponds to what is asked. The rule Force-DS simply extracts the shallow part of a deep value (which is possible thanks to the extended values). Finally, the rule Force-SD asks the deep value of a variable for which we only have the shallow value. To do this, we evaluate the λ-abstraction in a deep setting in the corresponding environment (using the rule Lam-D), and then replace the stored value with the new deep result, to memoize the computation for future uses.

4.3. An alternate presentation: eval/deepen

Our presentation of strong call-by-need (and call-by-name) semantics above has two evaluation modes, shallow and deep. Extracting a shallow value from a deep value is straightforward and does not perform any additional computation, but there is computation to transform a shallow value into a deep value.

We can give an alternate presentation, where the evaluation always computes a shallow value, and we have another conversion, called *deepening*, from shallow to deep values that can perform additional computation, which is of course memoized. There is now two evaluation relations: $e, m_1 \vdash t \Downarrow v, m_2$, which reduces t to value v in environment e, changing memory from m_1 to m_2, and $m_1 \vdash v \Downarrow\!\!\!\Downarrow r, m_2$, which reduces value v to normal form r, changing memory from m_1 to m_2.

For λ-abstractions, we now only use $\mathtt{D}\ (\lambda x.t, e, a)$, with the change that a is now a lazy reference to the normal form instead of the normal form. To do this, we need to introduce lazy values for the normal form of a λ-abstraction $\mathtt{L}_\lambda\ (\lambda x.t, e)$, lazy values which have already been computed $\mathtt{N}\ r$, and a forcing relation $m_1 \vdash a \Rrightarrow r, m_2$. The complete rules are shown in Figure 4.3.

The rules App-λ, App-I, Var and Force-Lazy are very similar to the ones existing in Figure 4.2 for $\Downarrow_{\mathtt{s}}$, with $e, m_2 \vdash t_2 \Downarrow v, m_3$ and $m_3 \vdash v \Downarrow\!\!\!\Downarrow r, m_4$ replacing the previous

App-λ

$$\dfrac{e_1, m_1 \vdash t_1 \Downarrow \mathtt{D}\ (\lambda x.t_3, e_2, a), m_2 \qquad e_2 \star (x \mapsto a), (a \mapsto \mathtt{L}\ (t_2, e_1)) \star m_2 \vdash t_3 \Downarrow v, m_3}{e_1, m_1 \vdash t_1\ t_2 \Downarrow v, m_3}$$

App-I

$$\dfrac{e, m_1 \vdash t_1 \Downarrow \mathtt{I}\ i, m_2 \qquad e, m_2 \vdash t_2 \Downarrow v, m_3 \qquad m_3 \vdash v \Ddownarrow r, m_4}{e, m_1 \vdash t_1\ t_2 \Downarrow \mathtt{I}\ (i\ r), m_4}$$

Lam

$$\dfrac{}{e, m \vdash \lambda x.t \Downarrow \mathtt{D}\ (\lambda x.t, e, a), (a \mapsto \mathtt{L}_\lambda\ (\lambda x.t, e)) \star m}$$

Var

$$\dfrac{m_1 \vdash e(x) \leadsto v, m_2}{e, m_1 \vdash x \Downarrow v, m_2}$$

Force-Lazy

$$\dfrac{e, (a \mapsto \Box) \star m_1 \vdash t \Downarrow v, (a \mapsto \Box) \star m_2}{(a \mapsto \mathtt{L}\ (t, e)) \star m_1 \vdash a \leadsto v, (a \mapsto v) \star m_2}$$

Force-Val

$$\dfrac{m(a) = v \qquad v \text{ is not a lazy value}}{m \vdash a \leadsto v, m}$$

ForceDeep-λ

$$\dfrac{\begin{array}{c} e \star (x \mapsto b), (a \mapsto \Box) \star (b \mapsto \mathtt{I}\ y) \star m_1 \vdash t \Downarrow v, (a \mapsto \Box) \star m_2 \\ (a \mapsto \Box) \star m_2 \vdash v \Ddownarrow r, (a \mapsto \Box) \star m_3 \end{array}}{(a \mapsto \mathtt{L}_\lambda\ (\lambda x.t, e)) \star m_1 \vdash a \Rrightarrow \lambda y.r, (a \mapsto \mathtt{N}\ (\lambda y.r)) \star m_3}$$

ForceDeep-V

$$\dfrac{m(a) = \mathtt{N}\ r}{m \vdash a \Rrightarrow r, m}$$

Deepen-I

$$\dfrac{}{m \vdash \mathtt{I}\ i \Ddownarrow i, m}$$

Deepen-λ

$$\dfrac{m_1 \vdash a \Rrightarrow r, m_2}{m_1 \vdash \mathtt{D}\ (\lambda x.t, e, a) \Ddownarrow r, m_2}$$

Figure 4.3.: Big-step strong call-by-need in eval-deepen presentation.

$e, m_2 \vdash t_2 \Downarrow_{\mathtt{d}} r, m_4$, which is what we do in general instead of $\Downarrow_{\mathtt{d}}$. The Lam rule allocates a fresh variable and a fresh location for the lazy body of the normal form, and the rule Force-Val factorises the rules Force-I, Force-SS and Force-DS. Finally, ForceDeep-λ and ForceDeep-V force the body of a λ-abstraction and deepens it to return its normal form, or returns an already-computed result, while Deepen-I and Deepen-λ extract the normal form from shallow values, forcing it if necessary.

4.4. Constructors and pattern matching

When introducing constructors and pattern matching, we extend the normal forms as well. Using the well-typedness property, there are still only two sorts of normal forms, with the following definitions:

$$\begin{aligned} r &::= \cdots \mid T\ \overline{r} \\ i &::= \cdots \mid \textbf{match}\ i\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow r}\ \textbf{end} \end{aligned}$$

Likewise, we can extend the contexts to handle these new terms:

$$\begin{aligned} C_r &::= \ \cdots \mid T\ (\overline{r}, C_r, \overline{t}) \\ C_i &::= \ \ldots \\ &\quad \mid \textbf{match}\ C_i\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow r}\ \textbf{end} \\ &\quad \mid \textbf{match}\ i\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow r} \mid S\ \overline{y} \Rightarrow C_r \mid \overline{T\ \overline{x} \Rightarrow t}\ \textbf{end} \end{aligned}$$

Here, we use $T\ (\overline{r}, C_r, \overline{t})$ instead of $T\ (\overline{t}, C_r, \overline{t})$ for C_r (and likewise for cases of a **match**), to ensure we keep the unique decomposition as a context and a redex; this causes evaluation to happen left-to-right.

Once this is done, we can extract big-step call-by-name semantics as before, whose rules are shown in Figure 4.4. We can see that the handling of constructors is very similar to the handling of λ-abstractions, while for **match**, it is very similar to applications. This is not surprising, since constructors happen in the same kind of normal forms r and contexts C_r as λ-abstractions, and likewise for **match** and applications appearing in inert normal forms i and contexts C_i.

The definition of call-by-name semantics with environments is straightforward, so we directly jump to the definition of call-by-need semantics. For this, we have two new types of values, $\mathtt{B}\ (T, \overline{a})$, the shallow value for a constructor T with lazy values as arguments, and $\mathtt{C}\ (T, \overline{a}, T\ \overline{r})$, the deep value for this constructor, carrying the normal form with it as well as the shallow value. The additional rules for our call-by-need semantics are quite natural given the existing rules, and are given in Figure 4.5.

Constr-S

$$\frac{}{T\ \bar{t} \Downarrow_{\mathtt{s}} T\ \bar{t}}$$

Constr-D

$$\frac{\forall i, t_i \Downarrow_{\mathtt{d}} r_i}{T\ \bar{t} \Downarrow_{\mathtt{d}} T\ \bar{r}}$$

Switch-C

$$\frac{t_1 \Downarrow_{\mathtt{s}} S\ \overline{u}^n \qquad t_2[\overline{y}^n := \overline{u}^n] \Downarrow_f v}{\textbf{match}\ t_1\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow w} \mid S\ \overline{y}^n \Rightarrow t_2 \mid \overline{T\ \overline{x} \Rightarrow w}\ \textbf{end} \Downarrow_f v}$$

Switch-I

$$\frac{t \Downarrow_{\mathtt{s}} i \qquad \forall k, u_k \Downarrow_{\mathtt{d}} r_k}{\textbf{match}\ t\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow u}\ \textbf{end} \Downarrow_f \textbf{match}\ i\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow r}\ \textbf{end}}$$

Figure 4.4.: Big-step call-by-name rules for constructors and pattern matching.

Constr-S

$$\frac{}{e, m \vdash T\ \bar{t} \Downarrow_{\mathtt{s}} \mathtt{B}\ (T, \overline{a}), \overline{(a \mapsto \mathtt{L}\ (t, e))} \star m}$$

Constr-D

$$\frac{m_0 = \overline{(a \mapsto \mathtt{L}\ (t, e))} \star m \qquad \forall i, e, m_{i-1} \vdash t_i \Downarrow_{\mathtt{d}} r_i, m_i}{e, m \vdash T\ \bar{t}^n \Downarrow_{\mathtt{d}} \mathtt{C}\ (T, \overline{a}, T\ \overline{\mathbf{nf}(r)}), m_n}$$

Switch-C

$$\frac{e, m_1 \vdash t_1 \Downarrow_{\mathtt{s}} \mathtt{B}\ (S, \overline{a}^n), m_2 \qquad e \star \overline{(y \mapsto a)}, m_2 \vdash t_2 \Downarrow_f v, m_3}{e, m_1 \vdash \textbf{match}\ t_1\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow u} \mid S\ \overline{y}^n \Rightarrow t_2 \mid \overline{T\ \overline{x} \Rightarrow u}\ \textbf{end} \Downarrow_f v, m_3}$$

Switch-I

$$\frac{e, m \vdash t \Downarrow_{\mathtt{s}} \mathtt{I}\ i, m_0 \qquad \forall k, e \star \overline{(x_k \mapsto a_k)}, \overline{(a_k \mapsto \mathtt{I}\ y_k)} \star m_{k-1} \vdash u_k \Downarrow_{\mathtt{d}} r_k, m_k}{e, m \vdash \textbf{match}\ t\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow u}^n\ \textbf{end} \Downarrow_f \mathtt{I}\ (\textbf{match}\ i\ \textbf{with}\ \overline{T\ \overline{y} \Rightarrow \mathbf{nf}(r)}\ \textbf{end}), m_n}$$

Force-ConstrDS

$$\frac{m(a) = \mathtt{C}\ (T, \bar{b}, T\ \overline{r})}{m \vdash a \rightsquigarrow_{\mathtt{s}} \mathtt{B}\ (T, \bar{b}), m}$$

Force-ConstrDD

$$\frac{m(a) = \mathtt{C}\ (T, \bar{b}, T\ \overline{r})}{m \vdash a \rightsquigarrow_{\mathtt{d}} \mathtt{C}\ (T, \bar{b}, T\ \overline{r}), m}$$

Force-ConstrSS

$$\frac{m(a) = \mathtt{B}\ (T, \bar{b})}{m \vdash a \rightsquigarrow_{\mathtt{s}} \mathtt{B}\ (T, \bar{b}), m}$$

Force-ConstrSD

$$\frac{\forall i, (a \mapsto \square) \star m_{i-1} \vdash b_i \rightsquigarrow_{\mathtt{d}} r_i, (a \mapsto \square) \star m_i}{(a \mapsto \mathtt{B}\ (T, \bar{b}^n)) \star m_0 \vdash a \rightsquigarrow_{\mathtt{d}} \mathtt{C}\ (T, \bar{b}^n, T\ \overline{\mathbf{nf}(r)}), (a \mapsto \mathtt{C}\ (T, \bar{b}^n, T\ \overline{\mathbf{nf}(r)})) \star m_n}$$

Figure 4.5.: Big-step call-by-need rules for constructors and pattern matching.

$$\textsc{Constr}\quad \frac{}{e, m \vdash T\ \overline{t} \Downarrow \mathtt{C}\ (T, \overline{a}, b), \overline{(a \mapsto \mathtt{L}\ (t, e))} \star (b \mapsto \mathtt{L}_C\ (T, \overline{a})) \star m}$$

$$\textsc{Switch-C}\quad \frac{e, m_1 \vdash t_1 \Downarrow \mathtt{C}\ (S, \overline{a}^n, b), m_2 \qquad e \star \overline{(y \mapsto a)}, m_2 \vdash t_2 \Downarrow v, m_3}{e, m_1 \vdash \textbf{match}\ t_1\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow u} \mid S\ \overline{y}^n \Rightarrow t_2 \mid \overline{T\ \overline{x} \Rightarrow u}\ \textbf{end} \Downarrow v, m_3}$$

$$\textsc{Switch-I}\quad \frac{\begin{array}{c} e, m \vdash t \Downarrow \mathtt{I}\ i, m_0 \\ \forall k, e \star \overline{(x_k \mapsto a_k)}, \overline{(a_k \mapsto \mathtt{I}\ y_k)} \star m_{k-1} \vdash u_k \Downarrow v_k, m'_k \qquad m'_k \vdash v_k \Ddownarrow r_k, m_k \end{array}}{e, m \vdash \textbf{match}\ t\ \textbf{with}\ \overline{T\ \overline{x} \Rightarrow u}^n\ \textbf{end} \Downarrow \mathtt{I}\ (\textbf{match}\ i\ \textbf{with}\ \overline{T\ \overline{y} \Rightarrow r}\ \textbf{end}), m_n}$$

$$\textsc{ForceDeep-C}\quad \frac{\forall i, (a \mapsto \square) \star m_{i-1} \vdash b_i \rightsquigarrow v_i, (a \mapsto \square) \star m'_i \qquad (a \mapsto \square) \star m'_i \vdash v_i \Ddownarrow r_i, (a \mapsto \square) \star m_i}{(a \mapsto \mathtt{L}_C\ (T, \overline{b}^n)) \star m_0 \vdash a \Rrightarrow T\ \overline{r}, (a \mapsto \mathtt{N}\ (T\ \overline{r})) \star m_n}$$

$$\textsc{Deepen-C}\quad \frac{m_1 \vdash b \Rrightarrow r, m_2}{m_1 \vdash \mathtt{C}\ (T, \overline{a}, b) \Ddownarrow r, m_2}$$

Figure 4.6.: Eval-deepen rules for constructors and pattern matching.

Like in section 4.3, we can also give an eval/deepen presentation for this. To do that, we only need to introduce lazy deep constructor values $\mathtt{L}_C\ (T, \overline{a})$, the rest of the machinery having already been introduced. The rules are given in Figure 4.6.

4.5. Fixpoints

As with constructors, we have to extend the normal forms and the contexts, and we use the well-typedness property to restrict the possibilities. The new normal forms and contexts are shown in Figure 4.7. We can see that due to the complexity generated by fixpoints forcing their n-th argument before being unfolded, the contexts become quite complex.

Then, once again, we extract big-step call-by-name semantics, shown in Figure 4.8. Due to the way **fix** values are used, most of the new rules are rules for reduction of an application. The two rules Fix-S and Fix-D simply reduce **fix** terms to **fix** values, whereas the following rules deal with how these values are applied to arguments. The rules App-Fix-S and App-Fix-D simply state that under-applied **fix** values remain **fix** values, while App-Fix-C and App-Fix-I force the last argument applied to a **fix** value.

$$
\begin{aligned}
r &::= \;\cdots \mid \mathbf{fix}_n\, f\; \overline{x}^n := r \;\mathbf{end}\; \overline{r}^m && (m < n)\\
i &::= \;\cdots \mid \mathbf{fix}_n\, f\; \overline{x}^n := r \;\mathbf{end}\; \overline{r}^{n-1}\; i\\
C_r &::= \;\dots\\
&\quad\mid \mathbf{fix}_n\, f\; \overline{x}^n := C_r \;\mathbf{end}\; \overline{r}^m && (m < n)\\
&\quad\mid \mathbf{fix}_n\, f\; \overline{x}^n := t \;\mathbf{end}\; \overline{r}^m\; C_r\; \overline{t}^p && (m + p + 1 < n)\\
C_i &::= \;\dots\\
&\quad\mid \mathbf{fix}_n\, f\; \overline{x}^n := t \;\mathbf{end}\; \overline{t}^{n-1}\; C_i\\
&\quad\mid \mathbf{fix}_n\, f\; \overline{x}^n := C_r \;\mathbf{end}\; \overline{r}^{n-1}\; i\\
&\quad\mid \mathbf{fix}_n\, f\; \overline{x}^n := t \;\mathbf{end}\; \overline{r}^m\; C_r\; \overline{t}^{n-2-m}\; i
\end{aligned}
$$

Figure 4.7.: Normal forms and contexts for fixpoints.

Fix-S

$$\frac{}{\mathbf{fix}_n\, f\; \overline{x}^n := t \;\mathbf{end} \Downarrow_{\mathtt{s}} \mathbf{fix}_n\, f\; \overline{x}^n := t \;\mathbf{end}}$$

Fix-D

$$\frac{t \Downarrow_{\mathtt{d}} r}{\mathbf{fix}_n\, f\; \overline{x}^n := t \;\mathbf{end} \Downarrow_{\mathtt{d}} \mathbf{fix}_n\, f\; \overline{x}^n := r \;\mathbf{end}}$$

App-Fix-S

$$\frac{t_1 \Downarrow_{\mathtt{s}} \mathbf{fix}_n\, f\; \overline{x}^n := t_3 \;\mathbf{end}\; \overline{u}^m \qquad m < n-1}{t_1\; t_2 \Downarrow_{\mathtt{s}} \mathbf{fix}_n\, f\; \overline{x}^n := t_3 \;\mathbf{end}\; \overline{u}^m\; t_2}$$

App-Fix-D

$$\frac{t_1 \Downarrow_{\mathtt{s}} \mathbf{fix}_n\, f\; \overline{x}^n := t_3 \;\mathbf{end}\; \overline{u}^m \qquad m < n-1 \qquad t_3 \Downarrow_{\mathtt{d}} r \qquad \forall i, u_i \Downarrow_{\mathtt{d}} r_i \qquad t_2 \Downarrow_{\mathtt{d}} r'}{t_1\; t_2 \Downarrow_{\mathtt{d}} \mathbf{fix}_n\, f\; \overline{x}^n := r \;\mathbf{end}\; \overline{r}^m\; r'}$$

App-Fix-C

$$\frac{\begin{array}{c} t_1 \Downarrow_{\mathtt{s}} \mathbf{fix}_n\, f\; \overline{x}^n := t_3 \;\mathbf{end}\; \overline{u}^{n-1} \\ t_2 \Downarrow_{\mathtt{s}} T\; \overline{w} \qquad t_3[f, \overline{x}^n := \mathbf{fix}_n\, f\; \overline{x}^n := t_3 \;\mathbf{end}, \overline{u}^{n-1}, T\; \overline{w}] \Downarrow_f v \end{array}}{t_1\; t_2 \Downarrow_f v}$$

App-Fix-I

$$\frac{t_1 \Downarrow_{\mathtt{s}} \mathbf{fix}_n\, f\; \overline{x}^n := t_3 \;\mathbf{end}\; \overline{u}^{n-1} \qquad t_2 \Downarrow_{\mathtt{s}} i \qquad t_3 \Downarrow_{\mathtt{d}} r \qquad \forall i, u_i \Downarrow_{\mathtt{d}} r_i}{t_1\; t_2 \Downarrow_f \mathbf{fix}_n\, f\; \overline{x}^n := r \;\mathbf{end}\; \overline{r}^{n-1}\; i}$$

Figure 4.8.: Big-step call-by-name rules for fixpoints.

If it reduces to a constructor, the function is unfolded, while if it does not, the normal forms of the body and arguments are computed and an inert term is created.

There are no new ideas used to extract the call-by-need semantics from the call-by-name semantics, so we will only show them for completeness in Figure 4.9. They are in the eval/deepen presentation, as it is the simplest presentation, even if it is already quite complex.

$$\textsc{Fix}\quad \frac{m' = (a \mapsto \mathtt{L}_{\mathbf{fix}}\ (\mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end}, e)) \star m}{e, m \vdash \mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end} \Downarrow \mathtt{F}\ (\mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end}, e, [], a), m'}$$

$$\textsc{App-Fix}\quad \frac{\begin{array}{c} e, m_1 \vdash t_1 \Downarrow \mathtt{F}\ (\mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end}, e_2, \overline{a}^p, b), m_2 \\ p < n-1 \qquad m_3 = (c \mapsto \mathtt{L}\ (t_2, e)) \star m_2 \end{array}}{e, m_1 \vdash t_1\ t_2 \Downarrow \mathtt{F}\ (\mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end}, e_2, \overline{a}^p :: c, b), m_3}$$

$$\textsc{App-Fix-C}\quad \frac{\begin{array}{c} e, m_1 \vdash t_1 \Downarrow \mathtt{F}\ (\mathbf{fix}_n\ f\ \overline{x}^{n-1}\ y := t\ \mathbf{end}, e_2, \overline{a}^{n-1}, b), m_2 \qquad e, m_2 \vdash t_2 \Downarrow \mathtt{C}\ (T, \overline{c}, d), m_3 \\ m_4 = (b' \mapsto \mathtt{F}\ (\mathbf{fix}_n\ f\ \overline{x}^{n-1}\ y := t\ \mathbf{end}, e_2, [], b)) \star (d' \mapsto \mathtt{C}\ (T, \overline{c}, d)) \star m_3 \\ e_2 \star (f \mapsto b') \star \overline{(x \mapsto a)} \star (y \mapsto d'), m_4 \vdash t \Downarrow v, m_5 \end{array}}{e, m_1 \vdash t_1\ t_2 \Downarrow v, m_5}$$

$$\textsc{App-Fix-I}\quad \frac{\begin{array}{c} e, m \vdash t_1 \Downarrow \mathtt{F}\ (\mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end}, e_2, \overline{a}^{n-1}, b), m' \qquad e, m' \vdash t_2 \Downarrow \mathtt{I}\ i, m_0 \\ \forall k, m_{k-1} \vdash a_k \rightsquigarrow v_k, m'_k \qquad m'_k \vdash v_k \Ddownarrow r_k, m_k \qquad m_{n-1} \vdash b \Rrightarrow r, m'' \end{array}}{e, m \vdash t_1\ t_2 \Downarrow \mathtt{I}\ (r\ \overline{r}^{n-1}\ i), m''}$$

$$\textsc{ForceDeep-Fix}\quad \frac{\begin{array}{c} e \star (f \mapsto b) \star \overline{(x \mapsto c)}, (a \mapsto \square) \star (b \mapsto \mathtt{I}\ g) \star \overline{(c \mapsto \mathtt{I}\ y)} \star m_1 \vdash t \Downarrow v, (a \mapsto \square) \star m_2 \\ (a \mapsto \square) \star m_2 \vdash v \Ddownarrow r, (a \mapsto \square) \star m_3 \qquad r' = \mathbf{fix}_n\ g\ \overline{y}^n := r\ \mathbf{end} \end{array}}{(a \mapsto \mathtt{L}_{\mathbf{fix}}\ (\mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end}, e)) \star m_1 \vdash a \Rrightarrow r', (a \mapsto \mathtt{N}\ r') \star m_3}$$

$$\textsc{Deepen-Fix}\quad \frac{\forall k, m_{k-1} \vdash a_k \rightsquigarrow v_k, m'_k \qquad m'_k \vdash v_k \Ddownarrow r_k, m_k \qquad m_p \vdash b \Rrightarrow r, m}{m_0 \vdash \mathtt{F}\ (\mathbf{fix}_n\ f\ \overline{x}^n := t\ \mathbf{end}, e_2, \overline{a}^p, b) \Ddownarrow r\ \overline{r}^p, m}$$

Figure 4.9.: Eval-deepen rules for fixpoints.

5. Implementing strong call-by-need

5.1. OCaml implementation

To implement strong call-by-need evaluation, we first need to define the type of terms, and the type of normal forms. Their definition are two simple inductive OCaml types:

```
type var = string

type term =
| Var of var
| Abs of var * term
| App of term * term

type nf =
| NfVar of var
| NfAbs of var * nf
| NfApp of nf * nf
```

Note that `nf` is isomorphic to terms (defining a type that can only contain normal forms is a bit harder and shown at the end of this section), but we use two distinct types to distinguish input terms, that we have not yet examined, and output terms, necessarily in normal form.

Now, we need to chose which semantics we are going to use for writing the evaluator. The simplest one for this is the eval/deepen semantics, which we are going to use. We also need environments, mapping variables to value references (needed for laziness), and a fresh variable name generator.[1] Thus, the type of values is as follows:

```
module VarMap = Map.Make(struct
  type t = var
  let compare = compare
end)

type value =
```

[1] The fresh variable name generator in the code below can generate variables that conflict with existing ones. However, since only the variables generated by this generator appear in the normal forms, there is no soundness issue, and we could even use separate types for variables in terms and in normal forms. If we wanted to be sure variable names were fresh, we could also use a type for variables such as `type var = Named of string | Fresh of int`.

```
| Inert of nf
| Lazy of term * env
| Blackhole
| Closure of env * var * term * nf option ref

and env = value ref VarMap.t

let fresh =
  let r = ref 0 in
  fun () -> incr r; "_" ^ string_of_int !r
```

Note that we have done a slight simplification here in the `Closure` case, where instead of having a lazy value corresponding to $\mathtt{L}_\lambda$ $(\lambda x.t, e)$, we simply use `None`, since all the information we would need is in the closure already. With the preceding definitions, it is now easy to write the evaluator, simply following the big-step semantics. The main difference with the eval/deepen semantics is that in the code of **`deepen`**, we use the information stored in the closure instead of information inside **`a`** to compute the normal form if it has not already been computed.

```
let makelazy t env = ref (Lazy (t, env))

let rec eval t env =
  match t with
  | Var x -> force (VarMap.find x env)
  | Abs (x, t) -> Closure (env, x, t, ref None)
  | App (t1, t2) ->
    match eval t1 env with
    | Lazy _ | Blackhole -> assert false
    | Closure (env1, x, body, _) ->
      eval body (VarMap.add x (makelazy t2 env) env1)
    | Inert i -> Inert (NfApp (i, deepen (eval t2 env)))

and deepen v =
  match v with
  | Lazy _ | Blackhole -> assert false
  | Inert i -> i
  | Closure (env, x, t, a) ->
    match !a with
    | Some nf -> nf
    | None ->
      let y = fresh () in
      let nenv = VarMap.add x (ref (Inert (NfVar y))) env in
      let nf = NfAbs (y, deepen (eval t nenv)) in
      a := Some nf; nf
```

```
and force v =
  match !v with
  | Blackhole -> assert false
  | Lazy (t, env) ->
    v := Blackhole;
    let r = eval t env in
    v := r; r
  | v -> v
```

Note that it is also possible to statically ensure the result is in normal form, by changing the definition of type `nf` to only allow normal forms; the type of values has to be changed as well to ensure inert values are inert normal forms.

```
type _ nf =
| NfVar : var -> [`inert] nf
| NfAbs : var * _ nf -> [`lambda] nf
| NfApp : [`inert] nf * _ nf -> [`inert] nf

type value =
| Inert of [`inert] nf
| Lazy of term * env
| Blackhole
| Closure of env * var * term * [`lambda] nf option ref
```

Unfortunately, due to limitations of the OCaml type system, we have to make a couple of modifications to the code. These are however quite simple and do not modify the structure of the code as a whole: we introduce a new type `any_nf`, which can hold values of both type ``[`inert] nf`` and ``[`lambda] nf``. Besides, we introduce smart constructors `mkNfAbs` and `mkNfApp` which work with values of type `any_nf` as input.

```
type any_nf = AnyNf : _ nf -> any_nf [@@unboxed]
let mkNfAbs (x, AnyNf t) = NfAbs (x, t) [@@inline]
let mkNfApp (i, AnyNf t) = NfApp (i, t) [@@inline]
```

With this change, we only have to replace the calls to `NfAbs` and `NfApp` by their corresponding smart constructor, and wrap the result of `deepen` inside an `AnyNf` constructor to get a version of the code producing terms which are guaranteed to be normal forms by their type definition.

5.2. Exploiting laziness from the host language

If the host language (OCaml in our case) supports lazy evaluation, we can use it to implement a purely functional evaluator, which is even simpler than our previous implementation. Indeed, the memory references in our eval/deepen implementation are used precisely as lazy values. Thus, we can simply use the laziness provided by the host language to avoid having to perform side effects.

The resulting interpreter is in the following form:

```
type value =
| Inert of nf
| Closure of env * var * term * nf Lazy.t

and env = value Lazy.t VarMap.t

let rec eval t env =
  match t with
  | Var x -> Lazy.force (VarMap.find x env)
  | Abs (x, t) ->
    Closure (env, x, t, lazy (
      let y = fresh () in
      let nenv = VarMap.add x (Lazy.from_val (Inert (NfVar y))) env in
      NfAbs (y, deepen (eval t nenv))))
  | App (t1, t2) ->
    match eval t1 env with
    | Closure (env1, x, body, _) ->
      eval body (VarMap.add x (lazy (eval t2 env)) env1)
    | Inert i -> Inert (NfApp (i, deepen (eval t2 env)))

and deepen v =
  match v with
  | Inert i -> i
  | Closure (_, _, _, nf) -> Lazy.force nf
```

As we can see, this implementation is a lot simpler than the previous one! Besides, using the host implementation of lazy values is more efficient than using our own custom-made lazy values, as the former are more optimised (for instance, the garbage collector can short-circuit already-computed lazy values).

5.3. Implementing extensions of the λ-calculus

As previously, we start by adding constructors and pattern matching as extensions. To give the simplest implementation, we will extend the last one we showed, where we exploit the laziness of the host language.

To add constructors, we need to extend the terms, normal forms, and values to handle them.

The new definitions of terms and normal forms are straightforward:

```
type term = ...
| Constr of int * term list
| Match of term * (var list * term) list

type nf = ...
| NfConstr of int * nf list
| NfMatch of nf * (var list * nf) list
```

Concerning the values, we need to add constructors, which are lazy, and which also lazily compute a normal form. Thus, we add the following case to the values:

```
type value = ...
| Block of int * value Lazy.t list * nf Lazy.t
```

Finally, the only changes to `eval` and `deepen` are to support the new terms and values, as well as failing due to a type error if we try to apply a `Block`. The new cases are shown in Figure 5.1.

Once we have constructors and pattern matching, we can add fixpoints as well. As previously, we need to extend the terms, normal forms, and values.

```
type term = ...
| Fix of var list * term

type nf = ...
| NfFix of var list * nf

type value = ...
| PartialFix of
    var list * term * env *
    value Lazy.t list * int *
    nf Lazy.t * nf Lazy.t
```

The new value `PartialFix` has a lot of arguments, so it is worth explaining what they do. The first three are the list of variables from the fixpoint, the term under the fixpoint, and the environment in which it is evaluated. The next two arguments are respectively the values currently applied to the fixpoint (there are always less such values than the arity of the fixpoint), and the number of remaining arguments. Finally, the last two values are respectively a lazy value of the normal form of the fixpoint, and a lazy value of the normal form of the fixpoint applied to all the current arguments.

The case for a `Fix (args, body)` term in the `eval` function, as well as the extension of the function `deepen`, are relatively straightforward and shown in Figure 5.2.

The real complexity lies in the new case for the application of a `PartialFix` value, as shown in Figure 5.3. When applying a fixpoint, there are two cases. The first, easy case is if there still arguments to apply, in which case we simply add the argument to the list of currently unapplied arguments, and we extend the lazy normal form correspondingly.

```
let extend_env env names values =
  List.fold_left2 (fun env name value ->
    VarMap.add name value env) env names values

let inert_var x = Lazy.from_val (Inert (NfVar x))

let force_deepen v = deepen (Lazy.force v)

let rec eval t env =
  match t with
  [...]
  | Constr (tag, l) ->
    let vl = List.map (fun t -> lazy (eval t env)) l in
    Block (tag, vl, lazy (
      NfConstr (tag, List.map force_deepen vl)))
  | Match (t, cases) ->
    match eval t env with
    | Closure _ -> failwith "type error"
    | Block (tag, args, _) ->
      if List.length cases <= tag then
        failwith "type error";
      let (names, body) = List.nth cases tag in
      if List.length args <> List.length names then
        failwith "type error";
      let env = extend_env env names args in
      eval body env
    | Inert i ->
      let nfcases = List.map (fun (names, body) ->
        let names2 = List.map (fun _ -> fresh ()) names in
        let env = extend_env env names (List.map inert_var names2) in
        (names2, deepen (eval body env))
      ) cases
      in
      Inert (i, nfcases)

and deepen v =
  match v with
  [...]
  | Block (_, _, nf) -> Lazy.force nf
```

Figure 5.1.: Rules for constructors and pattern-matching.

```
let rec eval t env =
  match t with
  [...]
  | Fix (args, body) ->
    let nf = lazy (
      let nargs = List.map (fun _ -> fresh ()) args in
      let nenv = extend_env env args (List.map inert_var nargs) in
      NfFix (nargs, deepen (eval body nenv)))
    in
    PartialFix (args, body, env, [], List.length args - 1, nf, nf)

and deepen v =
  match v with
  [...]
  | PartialFix (_, _, _, _, _, _, nf) -> Lazy.force nf
```

Figure 5.2.: Rule for evaluating fixpoints.

```
let rec eval t env =
  match t with
  [...]
  | App (t1, t2) ->
    match eval t1 env with
    [...]
    | PartialFix (args, body, env1, vals, i, nffix, nf) ->
      if i > 1 then
        (* Simple case: simply add the new argument to the list *)
        let nval = lazy (eval t2 env) in
        let nf = lazy (NfApp (Lazy.force nf, force_deepen nval)) in
        PartialFix (args, body, env1, nval :: vals, i - 1, nffix, nf)
      else
        (* Complicated case: i = 1, application happens here *)
        match eval t2 env with
        | PartialFix _ | Closure _ -> failwith "type error"
        | Inert i -> Inert (NfApp (Lazy.force nf, i))
        | (Block _) as nval ->
          let self = Lazy.from_val (PartialFix (args, body, env1,
              [], List.length args - 1, nffix, nffix)) in
          let vals = self :: List.rev (nval :: vals) in
          eval body (extend_env env args vals)
```

Figure 5.3.: Rule for applying fixpoints

The second case is more complicated, and happens when we add the final argument of the fixpoint. In this case, the argument needs to be forced, and it must be a `Block` if application is to happen. In this case, we have to reconstruct a value to be substituted for the fixpoint inside its body as well as the arguments, and we evaluate the body in this new environment. If the argument is inert instead, we simply return an inert term which we obtain by applying the normal form to the argument.

6. Coq proof

6.1. Overview

We performed a complete Coq proof that the semantics given in chapter 4 for strong call-by-need (with constructors and pattern matching, but without fixpoints) are correct with respect to the small-step semantics given by $\to_{\beta\iota}$. The semantics which is proved correct is the first version with the two forms for values of λ-abstractions (with and without the normal form), not the eval/deepen presentation. To prove it is correct, we established simulations between the successive semantics presented in chapter 4: small-step semantics, strong call-by-name, strong call-by-name with environments, and the call-by-need semantics as a memoized version of the call-by-name semantics.

All our big-step semantics are expressed in pretty-big-step form [Cha13]. Indeed, pretty-big-step allows for a more graceful handling of run-time errors (type errors, which would be the appearance of a forbidden pattern) and of divergence, as well as reducing duplication efforts in the proofs. Although we have only proved preservation of divergence for the first semantics (strong call-by-name without environments), the structure is there if we wish to extend the proof. In our case, the pretty-big-step version of the App-I and App-λ rules shown in section 4.1 would be decomposed as follows:

$$\begin{array}{lll}
\textsc{App} & \textsc{App-I} & \textsc{App-}\lambda \\
\dfrac{t_1 \Downarrow_{\mathtt{s}} v \qquad \textsc{App1}(v, t_2) \Downarrow_f v'}{t_1\ t_2 \Downarrow_f v'} &
\dfrac{t_2 \Downarrow_{\mathtt{d}} r}{\textsc{App1}(i, t_2) \Downarrow_f i\ r} &
\dfrac{t_1[x := t_2] \Downarrow_f v}{\textsc{App1}(\lambda x.t_1, t_2) \Downarrow_f v}
\end{array}$$

6.2. λ-terms and β-reduction

One crucial question before beginning the write a proof is to choose how binders are represented. The two main possibilities are de Bruijn indices and named variables, although a lot of other solutions are possible too, such as the locally nameless approach. In our case, we chose to use de Bruijn indices for the terms, because this allows us to avoid having to care about fresh variables, especially in substitutions. However, we used named variables for normal forms, because de Bruijn indices do not work with sharing (or need explicit renaming nodes, once again breaking the uniqueness of representations), and we want to be able to exploit sharing in our results.

The definition of terms is as follows:

```
Inductive term :=
| var : nat -> term
| dvar : nat -> term
| abs : term -> term
| app : term -> term -> term
| constr : nat -> list term -> term
| switch : term -> list (nat * term) -> term.
```

Here, the constructor `var` is for variables with de Bruijn indices, while `dvar` is reserved for future defined variables. The constructors `abs` and `app` handle abstractions and application, respectively, while `constr` is a data constructor (using integers as tags), and `switch` is for pattern matching, where each tag corresponds to the corresponding position in the list, and the integer is the number of bound variables (no names are necessary, since we are using de Bruijn indices).

Since the induction principle generated by Coq is not powerful enough for the `list term` nested subterms, we have to generate it by hand using a `Fixpoint` definition. The definition is simply a recursive function over the term:

```
Fixpoint term_ind2 (P : term -> Prop)
         (Hvar : forall n, P (var n))
         (Hdvar : forall n, P (dvar n))
         (Habs : forall t, P t -> P (abs t))
         (Happ : forall t1 t2, P t1 -> P t2 -> P (app t1 t2))
         (Hconstr : forall tag l, Forall P l -> P (constr tag l))
         (Hswitch : forall t m, P t ->
               Forall (fun '(p, t2) => P t2) m -> P (switch t m))
         (t : term) : P t :=
  match t with
  | var n => Hvar n
  | abs t =>
    Habs t (term_ind2 P Hvar Hdvar Habs Happ Hconstr Hswitch t)
  | constr tag l =>
    Hconstr tag l
      ((fix H (l : _) : Forall P l :=
        match l with
        | nil => @Forall_nil _ _
        | cons t l =>
          @Forall_cons _ _ t l
            (term_ind2 P Hvar Hdvar Habs Happ Hconstr Hswitch t)
            (H l)
        end) l)
  [...]
  end.
```

With this definition, we can define renamings and substitution. Renamings are increasing functions from integers to integers (with a few more conditions), that are used to change the values of de Bruijn indices in a term when we substitute terms under binders. With this, we can first define renaming the variables inside a given term, and then parallel substitution inside a term.

```
Fixpoint ren_term (r : renaming) t :=
  match t with
  | var n => var (renv r n)
  | abs t => abs (ren_term (lift r) t)
  | constr tag l => constr tag (map (ren_term r) l)
  [...]
  end.

Definition lift_subst us :=
  scons (var 0) (comp (ren_term (plus_ren 1)) us).
Definition liftn_subst p us n :=
  if p <=? n then ren_term (plus_ren p) (us (n - p)) else var n.

Fixpoint subst us t :=
  match t with
  | var n => us n
  | dvar n => dvar n
  | abs t => abs (subst (lift_subst us) t)
  | app t1 t2 => app (subst us t1) (subst us t2)
  | constr tag l => constr tag (map (subst us) l)
  | switch t l =>
      switch (subst us t) (map (fun pt2 =>
        (fst pt2, subst (liftn_subst (fst pt2) us) (snd pt2))) l)
  end.
```

We can see that both functions are very similar, with the main difference being in the variable case, where in the first case the renaming only returns the new index of the variable, where in the second case it returns a complete term. Unfortunately, we cannot simply implement `ren_term` in terms of `subst`, since lifting the substitutions in the recursive calls of `subst` necessitates to be able to rename terms! However, we have an equality theorem relating the two:

```
Definition ren r := fun n => var (renv r n).

Lemma ren_term_is_subst :
  forall t r, ren_term r t = subst (ren r) t.
```

The definition of terms and substitution allows us to finally define β-reduction.[1] As it

[1] Actually, what we call β-reduction in the code is $\beta\iota$-reduction.

is non-deterministic (we do not restrict ourselves to any fixed reduction strategy), it is defined as as inductive predicate relating two terms.

```
Inductive beta : term -> term -> Prop :=
| beta_app1 : forall t1 t2 t3, beta t1 t2 -> beta (app t1 t3) (app t2 t3)
| beta_app2 : forall t1 t2 t3, beta t1 t2 -> beta (app t3 t1) (app t3 t2)
| beta_abs : forall t1 t2, beta t1 t2 -> beta (abs t1) (abs t2)
| beta_redex : forall t1 t2, beta (app (abs t1) t2) (subst1 t2 t1)
| beta_constr : forall tag t1 t2 l1 l2, beta t1 t2 ->
    beta (constr tag (l1 ++ t1 :: l2)) (constr tag (l1 ++ t2 :: l2))
| beta_switch1 : forall t1 t2 l, beta t1 t2 ->
    beta (switch t1 l) (switch t2 l)
| beta_switch2 : forall t p t1 t2 l1 l2, beta t1 t2 ->
    beta (switch t (l1 ++ (p, t1) :: l2)) (switch t (l1 ++ (p, t2) :: l2))
| beta_switch_redex : forall l t l1 l2,
    beta (switch (constr (length l1) l) (l1 ++ (length l, t) :: l2))
         (subst (read_env l) t).
```

6.3. The call-by-name, big-step semantics

With the definition of beta done, we can move to defining the different semantics which we will prove equivalent to it. However, we first need to define the type of normal forms:

```
Inductive nfval :=
| nvar : nat -> nfval
| napp : nfval -> nfval_or_lam -> nfval
| nswitch : nfval -> list (nat * nfval_or_lam) -> nfval

with nfval_or_lam :=
| nval : nfval -> nfval_or_lam
| nlam : nfval_or_lam -> nfval_or_lam
| nconstr : nat -> list nfval_or_lam -> nfval_or_lam.
```

Here, binders are still in de Bruijn notation, as it is easier that way. However, we will soon switch binders to support sharing.

First, we define our substitution-based call-by-name big-step semantics, which will be presented in pretty-big-step [Cha13] format to simplify some parts of the proofs, for instance by allowing us to share parts of the proof between the different rules for function application. We first need to define a type for the results of evaluation: either a result, or something indicating divergence, to which we could add as future work a result indicating a runtime error.

```
Inductive out t :=
| out_ret : t -> out t
| out_div : out t.
```

We can then define a type of values for the results of computations, and of extended terms: either a normal term to be reduced, or an intermediate step in a reduction rule, like `ext_app` for the intermediate step when evaluating an application where the function has been reduced, but neither the argument nor the β-redex. Both of them come in two flavours, for shallow or deep evaluation.

```
Inductive deep_flag := shallow | deep.

Inductive val : deep_flag -> Type :=
| vals_nf : nfval -> val shallow
| vals_abs : term -> val shallow
| vald_nf : nfval_or_lam -> val deep
[...].

Inductive ext : deep_flag -> Type :=
| ext_term : forall df, term -> ext df
| ext_app : forall df, out (val shallow) -> term -> ext df
| ext_appnf : forall df, nfval -> out (val deep) -> ext df
| extd_abs : out (val deep) -> ext deep
[...].
```

Next, to propagate divergent (or error) states, we define the functions `get_out_abort` and `get_abort`, which lift any `out_div` out of an extended term:

```
Definition get_out_abort {t1 t2} (o : out t1) : option (out t2) :=
  match o with
  | out_div => Some out_div
  | _ => None
  end.

Definition get_abort {df t} (e : ext df) : option (out t) :=
  match e with
  | ext_term _ => None
  | ext_app o _ => get_out_abort o
  | ext_appnf _ o => get_out_abort o
  [...]
  end.
```

The core of the definition of the semantics is then a large ($\approx$ 60 lines) inductive definition with open recursion. We show the cases concerning the core λ-calculus below:

```
Inductive red_ (rec : forall df, ext df -> out (val df) -> Prop) :
    forall df, ext df -> out (val df) -> Prop :=
| red_var : forall df n,
    red_ rec df (ext_term (var n)) (out_ret (val_nf (nvar n)))
| red_abs_shallow : forall t,
    red_ rec shallow (ext_term (abs t)) (out_ret (vals_abs t))
| red_abs_deep : forall t o1 o2,
    rec deep (ext_term t) o1 ->
    rec deep (extd_abs o1) o2 ->
    red_ rec deep (ext_term (abs t)) o2
| red_abs1 : forall v,
    red_ rec deep
      (extd_abs (out_ret (vald_nf v)))
      (out_ret (vald_nf (nlam v)))
| red_app : forall df t1 o1 t2 o2,
    rec shallow (ext_term t1) o1 ->
    rec df (ext_app o1 t2) o2 ->
    red_ rec df (ext_term (app t1 t2)) o2
| red_app1_nf : forall df v o1 t2 o2,
    rec deep (ext_term t2) o1 ->
    rec df (ext_appnf v o1) o2 ->
    red_ rec df (ext_app (out_ret (vals_nf v)) t2) o2
| red_app1_abs : forall df t1 t2 o,
    rec df (ext_term (subst1 t2 t1)) o ->
    red_ rec df (ext_app (out_ret (vals_abs t1)) t2) o
| red_appnf : forall df v1 v2,
    red_ rec df
      (ext_appnf v1 (out_ret (vald_nf v2)))
      (out_ret (val_nf (napp v1 v2))
[...]
| red_abort : forall df e o,
    get_abort e = Some o -> red_ rec df e o.
```

The open recursion is closed afterwards, giving both inductive and coinductive definitions, so that it is possible to reason about termination: `red df e o` expresses that `e` terminates and reduces to `o`, while `cored df e o` expresses that `e` either diverges or reduces to `o`; in particular, `cored df e out_div` expresses that `e` diverges.

```
Inductive red : forall df, ext df -> out (val df) -> Prop :=
| mkred : forall df e o, red_ red df e o -> red df e o.

CoInductive cored : forall df, ext df -> out (val df) -> Prop :=
| mkcored : forall df e o, red_ cored df e o -> cored df e o.
```

Moreover, this scheme of defining the semantics via open recursion which we close at the end has a second advantage. Indeed, we implemented a small library which allows us to generate stronger induction principles for such datatypes, thus further simplifying the proofs.

With these definitions in hand, we write readback functions and prove our first semantics preservation theorem:

```
Fixpoint read_nfval v :=
  match v with
  | nvar n => var n
  | napp v1 v2 => app (read_nfval v1) (read_nfval_or_lam v2)
  | nswitch v l =>
    switch (read_nfval v)
      (map (fun pt2 => (fst pt2, read_nfval_or_lam (snd pt2))) l)
  end

with read_nfval_or_lam v :=
  match v with
  | nval v => read_nfval v
  | nlam v => abs (read_nfval_or_lam v)
  | nconstr tag l => constr tag (map read_nfval_or_lam l)
  end.

[...]

Lemma red_star_beta :
  forall df e o, red df e o ->
    forall t1 t2,
      read_ext e = Some t1 ->
      read_out o = Some t2 ->
      star beta t1 t2.
```

This theorem states that if we start with a state `e` and get a result `o`, both of which can be read back to λ-terms, then we have a sequence of reductions from the first term to the second. Note that this is a `forall t1 t2` and not `forall t1, exists t2`, as we might not get a result, in case of divergence or run-time errors. We however prove a second theorem, which entails that we do not introduce divergence in a strongly-normalising λ-term:

```
Lemma red_div_beta :
  forall t df e o, cored df e o ->
    read_ext e = Some t ->
    costar beta t (read_out o).
```

Taking `o = out_div`, we obtain that `cored df e out_div` implies divergence of `t`.

However, due to the inherent difficulties of working with coinductive types in Coq, we only proved this theorem of preservation of termination for this semantics, and not the following ones.[2] Moreover, we did not formalise the notion of forbidden subpatterns at all, so we have no proof of the absence of run-time errors.

6.4. Environment semantics

The next step is the semantics using environments, which we call `redE` in the code. The status of this semantics is a bit peculiar: as it is a form of intermediate semantics, there are some parts of it which are solely to simplify the proofs.

The definition of the semantics starts with a definition of the values inside the environment: either a variable name, or a term (with de Bruijn indices), together with its environment, itself a list of values. In the second case, we also keep a list of variable names which is a superset of the list of free variables of the term and its environment. This list of variables has no impact on the semantics, and is only there to help with the proof.

```
Inductive clo :=
| clo_var : freevar -> clo
| clo_term : list freevar -> term -> list clo -> clo.
```

We also define a type of normal forms with named variables:

```
Inductive nfvalx :=
| nxvar : freevar -> nfvalx
| nxapp : nfvalx -> nfvalx_or_lam -> nfvalx
| nxswitch : nfvalx -> list (list freevar * nfvalx_or_lam) -> nfvalx

with nfvalx_or_lam :=
| nxval : nfvalx -> nfvalx_or_lam
| nxlam : freevar -> nfvalx_or_lam -> nfvalx_or_lam
| nxconstr : nat -> list nfvalx_or_lam -> nfvalx_or_lam.
```

We then define `valE` and `extE` as previously, before defining `redE`. It is again defined using open recursion, and takes an additional input which is the environment. It also takes three(!) additional lists of variables as inputs, named `xs`, `fvs` and `dom`.

- `xs` is a list of variables which must not be used for new variable names,
- `dom` is a list of variables containing `xs`, which must contain all new variable names,
- `fvs` is a list of variables such that, if the free variables of the input are contained in `fvs`, so are the free variables of the output.

The interesting property of `fvs` is that it is only there to help with the proof, as shown by the following theorem:

[2] Some tools such as the `paco` library [Hur+13] might have helped; however, since we are mainly interested in proving correctness and not completeness, we did not pursue this further.

```
Lemma redE_fvs_any :
  forall df xs dom fvs e o, redE df xs dom fvs e o ->
    forall fvs2, redE df xs dom fvs2 e o.
```

The definition of `redE` itself is similar to the definition of `red`. However, there are a few interesting points concerning the handling of the list of free variables in `clo_term`:

```
Inductive redE_
  (rec : forall df, list freevar -> list freevar -> list freevar ->
    extE df -> out (valE df) -> Prop) :
  forall df, list freevar -> list freevar -> list freevar ->
    extE df -> out (valE df) -> Prop :=
| redE_clo_term : forall df xs xs2 dom fvs t env o,
    xs2 ⊆ dom ->
    rec df xs2 dom (list_inter xs2 fvs) (extE_term env t) o ->
    redE_ rec df xs dom fvs (extE_clo (clo_term xs2 t env)) o
| redE_app1_abs : forall df xs xs2 dom fvs env env2 t1 t2 o,
    xs ⊆ xs2 -> xs2 ⊆ dom ->
    rec df xs dom fvs (extE_term
      (match env_get_maybe_var env t2 with
       | Some c => c | _ => clo_term xs2 t2 env
       end :: env2) t1) o ->
    redE_ rec df xs dom fvs
      (extE_app env (out_ret (valEs_abs t1 env2)) t2) o
[...]
```

Here, in the definition of `redE_app1_abs`, the function call `env_get_maybe_var env t2` looks at whether `t2` is a variable, in which case it will return `Some c`, with `c` the value of that variable in the environment, otherwise it returns `None`. Note that a version of `env_get_maybe_var` which would always return `None` would be correct at well: it is only a small optimisation, which avoids creating a `clo_term` value that would only get evaluated to the value of this variable when we already know the result. Since it adds almost nothing to the proof effort, we included this optimisation here.

Here we can see that when adding a `clo_term` value to the environment, we use any list of variables between `xs` and `dom`. We could have used only `xs` instead, but this allows us to have monotonicity of our definition for both `xs` and `dom`.

Moreover, when evaluating a `clo_term` value, we reset the value of `xs` back to `xs2` which was held inside the `clo_term`! Indeed, since the free variables of the term are a subset of `xs2` (since they were at creation), there is no risk of accidental capture when using the result somewhere else in `redE_clo_term`. This is critical, since this is what allows us to memoize the evaluation in different contexts, which might not have the same `xs`, since they could be used under different numbers of binders.

As hinted above, `redE` is monotonic in both `xs` and `dom`, although in different directions, as shown by the following two theorems:

```
Lemma redE_xs_mono :
  forall df xs dom fvs e o, redE df xs dom fvs e o ->
    forall xs2, xs2 ⊆ xs -> redE df xs2 dom fvs e o.

Lemma redE_dom_mono :
  forall df xs dom fvs e o, redE df xs dom fvs e o ->
    forall dom2, dom ⊆ dom2 -> redE df xs dom2 fvs e o.
```

The third list of variables, `fvs`, is only there for the proof of the semantics preservation theorem. Unfortunately, it is not possible to remove it from the definition, and write a function that would yield `fvs` from the input, since it would lead to an universe inconsistency, due to `fvs` inhabiting in `Type`, while the semantics itself is an inductive `Prop`. It would be possible to change the semantics to reside in `Type`, but that would contaminate all other semantics of the proof, which would need to be in `Type` to avoid other universe inconsistencies.

With `redE` defined, we can prove it is correct with respect to `red`. First, we define a readback function from closures to terms, and a readback function from normal forms to terms.

```
Fixpoint read_clo (xs : list freevar) (c : clo) : term :=
  match c with
  | clo_var x => var (index xs x)
  | clo_term _ t l =>
    let nl := map (read_clo xs) l in
    subst (read_env nl) t
  end.

Fixpoint read_nfvalx xs v :=
  match v with
  | nxvar x => nvar (index xs x)
  | nxapp v1 v2 => napp (read_nfvalx xs v1) (read_nfvalx_or_lam xs v2)
  | nxswitch t m => [...]
  end

with read_nfvalx_or_lam xs v :=
  match v with
  | nxval v => nval (read_nfvalx xs v)
  | nxlam x v => nlam (read_nfvalx_or_lam (x :: xs) v)
  | nxconstr tag l => [...]
  end.
```

We can then use these to read `valE` and `extE`, using `read_clo` on the inputs and `read_nfvalx` for the outputs, mapping each `extE` constructor to its equivalent `ext` constructor and reading each of the arguments back to `terms`. Finally, with the help of some simple helper lemmas, we can prove the semantics preservation theorem:

```
Lemma redE_red :
  forall df xs dom fvs e o,
    redE df xs dom fvs e o ->
    fvs ⊆ xs /\ extE_closed_at e fvs ->
    forall fvs2,
      fvs ⊆ fvs2 ->
      red df (read_extE fvs2 e) (out_map (read_valE fvs2) o).
```

Here we can see the hypothesis `fvs ⊆ xs /\ extE_closed_at e fvs`, which uses `fvs` as we said earlier. It expresses the fact that the free variables of `e` are a subset of `fvs`, which themselves are a subset of `xs`. This property, being true at toplevel, is true for all recursive calls to the inductive predicate in the definition of `redE`. This theorem is proved independently, and our library for stronger induction principles allow us to derive an induction principle here where we don't have to prove again that this property is preserved. We thus get the result that if we have a reduction by `redE` from `e` to `o`, we have a reduction from their readbacks by `red` (and thus by `star beta`, using the other preservation theorem that we proved).

6.5. Transfer lemmas and memoized semantics

Before giving the definition of `redM`, the memoized semantics, we first prove the transfer lemmas we showed in section 4.2. Once again, we need to take care of free variables, such as the domain. For instance, in the case of λ-abstractions, we have the following dual lemmas:

```
Lemma redE_deep_shallow_abs :
  forall xs dom fvs e t env v,
    redE deep xs dom fvs e (out_ret (valEd_abs t env v)) ->
    forall e2,
      extE_deep_to_shallow e = Some e2 ->
      redE shallow xs dom fvs e2 (out_ret (valEs_abs t env)).

Lemma redE_shallow_deep_abs :
  forall xs dom fvs e t env,
    redE shallow xs dom fvs e (out_ret (valEs_abs t env)) ->
    forall o dom2, xs ⊆ dom -> dom ⊆ dom2 ->
      redE deep dom dom2 fvs (extE_term env (abs t)) o ->
      redE deep xs dom2 fvs (extE_shallow_to_deep e) o.
```

As we can see, in the second lemma, we require that the second reduction happens with a set of forbidden variables equal to all variables that could have been defined in the first reduction. That way, there is no risk of collision between both sets of variables.

Once we have those transfer lemmas, we can move on to the definition of the memoized semantics `redM`.

To define them, we need to introduce a memory, which stores the results of the evaluation of each term of type clo in the previous semantics. Thus, whereas we had a value of type clo, we now have a memory reference (freevar), and the memory contains results of the evaluation of such memory references, or the eiM_lazy constructor to indicate that it has not yet been evaluated.

```
Inductive eiM :=
| eiM_lazy : term -> list freevar -> eiM
| eiM_abs1 : term -> list freevar -> eiM
| eiM_abs2 : term -> list freevar -> nfvalx_or_lam -> eiM
| eiM_constr1 : nat -> list freevar -> eiM
| eiM_constr2 : nat -> list freevar -> nfvalx_or_lam -> eiM
| eiM_val : nfvalx -> eiM.
Definition memM := list (freevar * eiM).
```

The previous clo_term is now replaced by eiM_lazy, which has not been evaluated yet (but which can have references to terms that have been evaluated). Where we had clo_var x previously, we now have eiM_val (nxvar x), where the increased generality is used to store results of terms. All the other cases represent results of evaluation, with two versions, one for shallow and one for deep evaluation. Thanks to the transfer lemmas, we can extract the shallow version from the deep version, and the deep version from the shallow version after some more computation.

At this point we can define valM and extM with definitions close to valE and extE, replacing the type clo by memory references freevar. Then, we can define a predicate update_result which takes the result of a computation and will update it inside the memory at a given location (with eiM_abs1, eiM_constr1 and eiM_val used if it was a shallow evaluation, and eiM_abs2, eiM_constr2 and eiM_val if it was deep). Finally, we can define redM like for redE, with new rules instead of redE_clo_term:

```
Inductive redM_ (rec : forall df, extM df -> outM (valM df) memxM -> Prop) :
  forall df, extM df -> outM (valM df) memxM -> Prop :=
| redM_clo_abs1_shallow : forall m x t env,
    env_find (fst m) x = Some (eiM_abs1 t env) ->
    redM_ rec shallow (extM_clo m x) (outM_ret (valMs_abs t env) m)
| redM_clo_abs1_deep : forall m x t env o1 o2,
    env_find (fst m) x = Some (eiM_abs1 t env) ->
    rec deep (extM_term env m (abs t)) o1 ->
    update_result deep x o1 o2 ->
    redM_ rec deep (extM_clo m x) o2
| redM_clo_abs2_shallow : forall m x t env body,
    env_find (fst m) x = Some (eiM_abs2 t env body) ->
    redM_ rec shallow (extM_clo m x) (outM_ret (valMs_abs t env) m)
| redM_clo_abs2_deep : forall m x t env body,
    env_find (fst m) x = Some (eiM_abs2 t env body) ->
    redM_ rec deep (extM_clo m x) (outM_ret (valMd_abs t env body) m)
```

```
| redM_clo_lazy : forall df m x t env o1 o2,
    env_find (fst m) x = Some (eiM_lazy t env) ->
    rec df (extM_term env m t) o1 ->
    update_result df x o1 o2 ->
    redM_ rec df (extM_clo m x) o2
[...]
.
```

Here, we can see that `redM_clo_lazy` simply evaluates a lazy term, then stores it result before returning. On the other hand, we can see four versions of `redM_clo_abs`, for both shallow or deep evaluation, depending on whether a shallow or deep version was stored. If the version stored is the same as evaluation, we can simply return the version that was stored, but if not, we can use the transfer lemmas to either extract or compute the version we need, possibly updating the stored version from shallow to deep.

The correctness of this is quite complex to prove. Indeed, we need a readback relation `res` between a `clo` and a memory element `eiM`, given the state of the memory. As what can be stored in the memory can be an evaluated version of the `clo_term`, we need to specify that it is indeed the expected result, and thus we need to call `redE` there.[3] This really is the core of the proof, that the results that have been memoized are indeed the correct results.

Thus, the definition of `read_eiM` is as follows:

```
Inductive read_eiM res dom : eiM -> clo -> Prop :=
| read_eiM_lazy : forall t ys vs xs,
    map Some vs = map res ys ->
    read_eiM res dom (eiM_lazy t ys) (clo_term xs t vs)
| read_eiM_abs1 : forall t ys u ws vs xs fvs,
    map Some vs = map res ys ->
    redE shallow xs dom fvs (extE_term ws u) (out_ret (valEs_abs t vs)) ->
    read_eiM res dom (eiM_abs1 t ys) (clo_term xs u ws)
| read_eiM_abs2 : forall t ys u ws v vs xs fvs,
    map Some vs = map res ys ->
    redE deep xs dom fvs (extE_term ws u) (out_ret (valEd_abs t vs v)) ->
    read_eiM res dom (eiM_abs2 t ys v) (clo_term xs u ws)
[...]
| read_eiM_var : forall y,
    read_eiM res dom (eiM_val (nxvar y)) (clo_var y).
```

[3] This is better than `redM` because it is pure and does not depend on the state of the memory. If we used `redM`, it would be unclear which memory state we would need, as we could use the current state, or the state when the lazy value was reduced, or again the state when it was produced. Whatever the state chosen, we would need to prove that other choices work as well. With the pure `redE`, we do not have this problem at all.

We can see the readback is no longer a function from one kind of values to those of the previous semantics, but instead a relation, making it less practical to work with, but able to express more powerful invariants concerning the memory. `res` then acts like an oracle giving what the contents of the memory should correspond to, and the various theorems will then assume the memory is compatible with this expected result.

Now that we have a way to read the memory, we can write the specifications for reading `extM` and `valM` states. These simply make a correspondence with the matching `extE` or `valE` constructor, using `res` as the oracle to read the memory references to produce the corresponding closures, and ensuring the memory state is consistent with the expected readback `res`.

Once this is done, we define a notion of compatibility between two readbacks of states of the memory `res1` and `res2`, specifying that `res2` is defined on more memory addresses than `res1`, and on those in common, both are read the same way; i.e. the memory was extended between `res1` and `res2`. We can then prove that the readback of `eiM` or `valM` are unaffected in this extended memory, establishing we only need to prove that the new addresses that were defined in the memory were read correctly. Besides, we also establish that updating a memory location to something else that has the same readback is correct, which is used when we update a lazy value to its computed version.

Finally, we can prove our preservation theorem from `redM` to `redE`:

```
Lemma redM_redE :
  forall df e o,
    redM df e o ->
    forall m res e2,
      get_ext_memxM e = Some m ->
      read_extM df res e e2 ->
      exists m2 o2 res2,
        get_out_memxM o = Some m2 /\
        redE df (snd m) (snd m2) nil e2 o2 /\
        read_outM df res2 o o2 /\
        compat_res res res2.
```

It says that if there is a reduction from `e` to `o`, `e` contains a memory state `m` (i.e. it is neither an error nor divergence) and that we can read `e` as `e2` with some corresponding readback of memory `res`, then the result `o` contains some memory `m2`, which is read in a compatible readback `res2` to `o2`, and that there is a reduction `redE` from `e2` to `o2`, which is the preservation theorem we wanted.

Thus, each of these semantics was proved correct with respect to the previous semantics, establishing that our strong call-by-need semantics is indeed correct with respect to the small-step semantics of the λ-calculus.[4] As said before, we have not proved the preservation of normalisation, as the handling of coinduction in Coq is complex.

In a previous attempt, we had a full proof for a small-step strong call-by-need semantics, but only for the pure λ-calculus, without constructors. As the proof was already very large (more than 10000 lines of Coq) and relied on extremely complex invariants (specifying two terms were equal when a variable was substituted in some places and not substituted in other places, keeping a well-founded order on variables, etc.), it was too difficult to scale to a proof for the extended λ-calculus and we wrote a proof for big-step semantics instead.

6.6. Extending induction principles with invariant properties

The types shown above for reduction relations are very large, and often require proving that some invariants hold when doing an inductive proof. In particular, a common pattern is to prove inductively:

```
Lemma example : forall (a : T), P a -> Q a.
```

In this case, we need to prove `P b` for each `b` which is a “subterm” of `a`. Such a proof can be done independently, and then we can have a stronger induction principle for instantiations of the above for various values of `Q`, without repeating the tedious step of proving `P b` in each case.

However, it is quite tricky to define what a subterm is, especially when the definitions come from an already large term. Thankfully, open recursion and higher-order definitions can help us a lot to avoid writing two very similar definitions and keeping them in sync.

Thus, suppose we have an inductive proposition `P` of one argument, defined by the open recursion `G`:

```
Context (G : (A -> Prop) -> (A -> Prop)).
Inductive P : A -> Prop := mkP : forall (x : A), G P x -> P x.
```

First, to abstract this into a library, we need to be able to specify what the inductive definition means without using an inductive definition. In our case, we need three things:

- We have the equivalent of the constructor : `forall x, G P x -> P x.`
- The induction hypothesis holds:
  `forall Q, (forall x, G Q x -> Q x) -> forall x, P x -> Q x.`
  When defining a new inductive type in Coq, Coq automatically derives such an induction theorem for us.

[4] Technically speaking, we would need to write a readback relation from the `redM` terms to `term` to prove that, and compose all the theorems together. We have not done this, as at the point the proof was finished, we focused on the more interesting convertibility problem instead.

- The definition is *positive*:
  `forall Q R, (forall x, Q x1 -> R x1) -> forall x, G Q x -> G R x.`
  Positivity is required to prove that if we can prove some property on our terms, we can prove a stronger one. It is also necessary to define new inductive types in Coq, so this property will always hold for inductively defined types.

These three conditions are a bit more general than inductive types: for instance, Coq inductive types require *strict positivity* as well, meaning no subterms can happen to the left of an arrow, even in covariant positions. However, we can prove the induction principles that interest us without it.

The stronger induction principles we want to prove have an invariant, which is a property `Q`, such that if `Q` is true on a term `x`, then it is true on all subterms of `x`. The naïve version would be to define it in the following way:

```
Definition preserved_down Q :=
  forall x, G P x -> Q x -> G (fun x => P x /\ Q x) x.
```

However, if there are several constructors for `G`, then there might be a proof of `G (fun x => P x /\ Q x) x`, but which did not correspond to the initial proof of `G P x` we had. Thus, we specify a stronger requirement for `preserved_down`, where the presence of `R` will ensure we have the same proof:

```
Definition preserved_down Q :=
  forall R x, G (fun x => P x /\ R x) x -> Q x ->
    G (fun x => P x /\ Q x /\ R x) x.
```

We can also specify what it means that a property `Q` is inductive, assuming we have invariant `R`. It simply means that we can assume that `R` is true on `x` and all its subterms, and we need to prove `Q` from it.

```
Definition preserved_with_inv Q R :=
  forall x, G (fun x => P x /\ Q x /\ R x) x -> R x -> Q x.
```

From this, we can prove the induction-with-invariants theorem:

```
Lemma preserved_with_inv_rec :
  forall Q R, preserved_down R -> preserved_with_inv Q R ->
    forall x, P x -> R x -> Q x.
```

The advantage of this induction theorem is that we can prove `preserved_down R` once, and then instantiate the theorem with various values for `Q`!

In practice, we use a small OCaml program to generate a version of this for each number of arguments, as it would be quite complicated to write the above in a way that is parametric in the number of arguments. We also write a tactic to prove that a inductive type satisfies our definitions, to avoid proving them by hand each time.

7. Experimental evaluation

7.1. Methodology

For experimental evaluation, we relied on our OCaml implementation of the strong call-by-need evaluator, which includes the handling of fixpoints additionally to what has been verified in Coq. Variables in the input terms were represented by the type `string`, which comes with additional cost compared to Coq's internal de Bruijn indices. On the Coq side, we instrumented Coq's convertibility checker so that it prints the time taken (this is much more precise than just relying on Coq's `Time` command, which also accounts for other aspects such as typechecking). We used Coq 8.15.2, extended with these changes to the reduction engine. Moreover, both Coq and our implementation were compiled using OCaml 4.12.1, and the experiments were made on a laptop with an 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz CPU and 2x 16GiB SODIMM DDR4 Synchronous 3200 MHz (0.3 ns) RAM, running Linux 5.15.74 with NixOS 22.05.

7.2. Testcases

We measured time taken by both Coq and our implementation to compute the normal forms of various λ-terms, especially with computations in Peano or Church arithmetic.

The testcases used are the same as in the benchmarks used by Grégoire and Leroy [GL02], corresponding to computing the normal forms of different computations on integers, both using Peano and Church integers. The first three tests with Peano integers only use weak reduction, but all other tests need strong reduction to compute the normal form. The results are summarised in Figure 7.1.

Testcase (Peano)	Coq	Ours
`test1p`	0.40s	1.84s
`test2p`	1.08s	0.74s
`test3p`	28ms	44ms
`test4p`	15ms	13ms

Testcase (Church)	Coq	Ours
`test1c`	1.18s	2.09s
`test2c`	1.11s	0.98s
`test3c`	1.7ms	0.3ms
`test4c`	2.5ms	0.4ms

Figure 7.1.: Timings, in seconds, for the computation of normal forms of different terms. Left column uses Peano integers, right column uses Church integers. The tests correspond to, in this order, `factorial 9`, `is_even (factorial 9)`, `256 * 64` and `(fun x y => (128 + x) * (128 + y))`.

Concerning Church integers, since our evaluator is untyped while Coq is typed, we chose to use Church integers specialised to a given type as well as definitions of the operations used (in particular, factorial) so that there are no type abstractions or evaluations during the reduction to keep the comparison as fair as possible.

We see here that the time taken by Coq and by our evaluator is on the same order of magnitude. Since our evaluator relies on a formally verified evaluation semantics, this is a victory: we were able to formally verify an evaluation strategy that behaves close to the one used by Coq.

In the next part of this thesis, we will focus on our original problem, a convertibility test. We will see there that, by using a more complex strategy, we are able to significantly outperform Coq in some cases.

8. Future and related work

8.1. Related work

8.1.1. A compiled implementation of strong reduction

In [GL02], Grégoire and Leroy present a virtual machine (which we will here call the OpenZAM), which is a modified version of the ZAM used in OCaml's implementation, allowing to perform open call-by-value reduction. With a procedure consisting in open head reduction followed by iterative reduction under the λ-abstractions that appear in the normal form, they obtain a machine for strong call-by-value reduction. While this machine is efficient with minimal changes to the readback procedure to preserve sharing, and is indeed used by Coq's `vm_compute` strategy, it is still call-by-value reduction, and will sometimes reduce subterms that are never needed, potentially even resulting in non-termination. However, it also has the merit of being a compiled implementation. As we ensured our own semantics respected the subterm property, we could actually mimic the OpenZAM to obtain efficient compilation of λ-terms if we so wished.

8.1.2. Crégut's KN and KNL machines

In [Cré07], Crégut presents two abstract machines, KN and KNL, which are variations on the KAM for strong call-by-need reduction. While the KN machine satisfies the subterm property, it is unable to share the computation of normal forms of the same λ-abstraction in different contexts, leading to exponential complexity in some cases. On the other hand, KNL tries to share reductions, but loses the subterm property by reducing under λ-abstractions before they are applied, and having a potentially quadratic slowdown in the worst case, as it can end up reducing a normal form that has already been computed to normal form repeatedly. Besides, normal forms are expressed using de Bruijn levels; as we mentioned in section 2.1, this makes it impossible to use some kinds of sharing, causing normal forms to grow larger.

8.1.3. Full laziness

In [Bal12], Balabonski studies *full laziness*, a stronger form of call-by-need. With full laziness, subexpressions inside a λ-abstraction which do not depend of the value of the variable being abstracted on are shared among all the applications of the λ-abstraction, and thus only evaluated once instead of each time the λ-abstraction is applied to an argument. For instance, consider the λ-abstraction $\lambda x.x + \mathbf{exp2}\ 30$. Here, $\mathbf{exp2}\ 30$ is expensive to compute but does not depend on x, and we would like to compute it only once, and not each time our λ-abtraction is applied. On the other hand, we would like it not to be evaluated if we are never going to apply our λ-abstraction, to avoid paying the cost of something we are never going to use.

There have been many previous definition of full laziness in the literature (for instance in [Jon87]); Balabonski shows they are all equivalent in the sense that they all perform the same β-reductions. While the semantics we gave do not have the property of full laziness, one standard way to obtain it is to apply a transformation called *λ-lifting* to the input program, so that the resulting program will behave under our semantics as the original program, except that subexpressions not depending on the argument of the λ-abstraction are shared. The idea behind λ-lifting is simply to enforce sharing of all these subexpressions by replacing them with variables that are let-bound outside the λ-abstraction.

More precisely, if t is of the form $\lambda x.C[u]$, where u does not depend on x, then we replace it with $(\lambda y.\lambda x.C[y])\ u$, which has the same semantics as t since it directly reduces to t. However, when we reduce the outermost redex, we create a lazy binding for u, causing it to be computed once the first time me apply the inner λ-abstraction, and the result of the computation to be reused each time we apply it again. For instance, in the previous example, we would get $(\lambda y.\lambda x.x + y)$ (**exp2** 30), and the computation of **exp2** 30 will only happen at most once while reducing.

More generally, we identify under every λ-abstraction every subterm which is not a variable and does not depend on the parameter of the λ-abstraction, and we replace each of them by a variable, which we bind to its definition outside all λ-abstractions with a parameter that does not appear inside the definition.

With this transformation, both weak and strong normalisation are preserved, as the new redexes we introduce have a very specific form: no redex is created at runtime that did not exist in the original term. Thus, termination is preserved with call-by-name and call-by-need, as both these strategies terminate if and only if the term is weakly normalising. However, termination is not guaranteed when using specific evaluation strategies. Indeed, with call-by-value, lifting a term that does not terminate outside a function that is never applied will cause this term to be evaluated before the application of the function to an argument, causing non-termination, while the original program never applied this function and was thus able to terminate. This can for instance be seen on the following term: $(\lambda x.w)\ (\lambda y.\Omega)$, where Ω does not terminate.

8.1.4. Automatically converting small-step semantics to big-step semantics

In [Cio13], Ciobâcă examines a transformation that can be used to convert small-step semantics to big-step semantics. The transformation proposed there could probably be used to produce our big-step call-by-name semantics, by adapting it to our formalisation with contexts instead of an inductively-defined reduction relation, and adapting it to the fact that we have two different evaluation contexts and thus two big-step reduction relations to consider. In any case, our big-step semantics are simple enough that we obtained them quite naturally by hand, before noticing while writing this thesis that there was a simple, almost-mechanical process to extract them from the small-step semantics.

8.1.5. Balabonski, Barenbaum, Bonelli, and Kesner's λ_c calculus

In [Bal+17], Balabonski, Barenbaum, Bonelli, and Kesner present a strong call-by-need calculus λ_c. This calculus is an explicit substitution calculus, and designed to be an extension of weak call-by-need calculi, by never reducing inside a λ-abstraction that will later be applied. As such, it is similar to our own semantics, although our semantics keep sharing only implicitly in the output (with the sharing of pointers in the representation), while they keep sharing explicit even in the results. Another difference is that since they do not keep the two versions of λ-abstractions like we do, they must disallow reducing under λ-abstractions until after all their uses have been seen, which is complex to implement without at least re-traversing the terms. While we only prove correction of our calculus, they also prove completeness and conservativity over the weak call-by-need calculus. However, they do not provide an abstract machine, nor a formal proof of correctness. Still, we expect our semantics to be similar to theirs, using exactly the same number of β-reductions to reach a normal form.

8.1.6. Balabonski, Lanco, and Melquiond's strong call-by-need calculus

In [BLM21], Balabonski, Lanco, and Melquiond present a strong call-by-need calculus, and an abstract machine for it. This calculus is an extension of λ_c, which allows reduction under λ-abstractions before applying them if it identifies that the normal form of the λ-abstraction will be needed, and is as such able to perform fewer β-reduction steps than our semantics or λ_c. However, it can also cause the terms being reduced to grow from their initial size; having to be analysed again when the λ-abstraction is applied, potentially causing quadratic complexity of the abstract machine, although further study would be needed to identify if this actually happens.[1] Besides, we lose the subterm property again, which we wanted to enforce to make sure compilation to a virtual machine or even machine code was possible.

8.1.7. Accattoli, Condoluci, and Coen's SCAM

In [ACC21], Accattoli, Condoluci, and Coen present the SCAM, an abstract machine for strong call-by-value reduction. Whereas Grégoire and Leroy's focus was on practical efficiency, their focus is on theoretical complexity, allowing them to prove a bilinear

[1] This is not unlike Lévy's optimal reduction, which makes an optimal number of β-reductions among all possible reduction strategies, but comes at a potentially non-elementary administrative cost to find which redex needs to be reduced.

$O(mn)$ complexity for the SCAM, where m is the number of β-reduction steps and n the size of the initial term, instead of the usual quasibilinear $O(mn\mathbf{log}n)$ complexity due to environment lookups. It shows the subtleties and pitfalls of efficient open reduction. The general focus on complexity in Accattoli's papers is something we kept in mind while designing our semantics and abstract machines, in both this part and the next, to ensure our machines do not introduce unnecessary inefficiencies.

8.1.8. The RKNL machine

In [BCD22], Biernacka, Charatonik, and Drab present the RKNL machine, a simple and efficient abstract machine for strong call-by-need calculus. While developed independently, this machine is actually isomorphic to our big-step semantics for strong call-by-need, and can (and is) obtained mechanically by CPS-transformation from a program almost identical to our eval-deepen presentation in section 5.2, which they describe as a modified version of Crégut's KN machine. They perform further analysis of the obtained machine, including proofs of soundness, correctness and complexity; resulting in the quasibilinear complexity we expected. On the other hand, all proofs are paper proofs only, making our Coq proof give increased confidence in the RKNL machine.

8.1.9. Verified compilation of functional languages

Closely related, there has been a lot of work on verified compilation of functional languages. Dargaye [Dar09] verified a compiler for a small subset of ML to Cminor, an intermediate language used in CompCert, but without verifying the code of the garbage collector. McCreight, Chevalier, and Tolmach [MCT10] established a framework for verified compilation of garbage-collected languages by providing an intermediate language, GCminor, which includes primitives for allocation and specifying GC roots, and compiling it to Cminor. The CakeML project [Kum+14] is a verified optimising compiler for a large subset of Standard ML directly to machine code, which is also able to bootstrap itself; later extended with a second frontend, PureCake [Kan+23], for a lazy, purely-functional language. A bit earlier, Stelle and Stefanovic [SS18] were the first to verify compilation of call-by-need semantics, from the λ-calculus to a virtual machine. Swierstra [Swi12], for the KAM, and Kunze, Smolka, and Forster [KSF18], for call-by-value, derived machines for the λ-calculus by refining successive machines directly from the small-step semantics, related to how we obtained our strong call-by-need semantics by refining successive big-step semantics. There has also been work on verified compilation of Coq, in particular the CertiCoq project [Ana+17], a compiler from Coq to Cminor, or the work on OpenZAM by Grégoire and Leroy [GL02], compiling Coq to a virtual machine, and the only work of this list doing open, and not just weak, reduction.

8.1.10. Graph reduction for the λ-calculus

The idea of using graph reduction as a λ-calculus evaluation strategy allowing sharing is not new, and could already be found in Wadsworth's PhD thesis [Wad71]. An introduction to graph reduction and its implementation can be found in Jones's book [Jon87]. The spineless tagless G-machine [Jon92] is one efficient implementation among others of graph reduction. However, graph reduction was often viewed as an implementation detail and waited a long time before being formally studied [SW10; BLM05]. Another use of graph reduction has been for the implementation of optimal reduction by Lamping [Lam90], later made simpler and more efficient by Gonthier, Abadi, and Lévy [GAL92]. Asperti, Giovanetti, and Naletto [AGN96] created an abstract machine for these algorithms. These algorithms perform a minimal number of β-reductions, by sharing not only subterms, but contexts. However, this comes at a steep cost: Lawall and Mairson [LM96] first proved that the cost of bookkeeping of these algorithms was at least exponential; Asperti, Coppola, and Martini [ACM04] later proved that the cost was even non-elementary in the number of β-reductions. While this shows the optimal number of β-reductions is not a good cost model of the λ-calculus, the picture is not as bleak as it appears: it is conjectured by [Asp17] that this cost is mainly due to necessary duplication, and the actual overhead is only polynomial.

8.2. Future work

8.2.1. A virtual machine for strong call-by-need reduction

While the semantics we gave in section 4.3 lead quite straightforwardly to the OCaml implementation described in chapter 5, compiling the terms depends on having a deepening function, which operates on dynamically-known terms only, preventing the mechanical transformation to a virtual machine we showed at the beginning. The idea amounts to having a *runtime*, as in most programming languages, which would provide this function. We would then add a single opcode `DEEPEN`, which would call this function, examining the term on which it is called, and recursively calling itself as needed to deepen the inner subterms. We have however not fully written the rules of such a virtual machine yet, and this remains as future work.

8.2.2. Complexity analysis

Analysing the complexity of this strong call-by-need semantics (or its corresponding OCaml code) is not obvious. While we can quite easily give a bound depending on the number of β-reductions that happened at runtime, it is harder to quantify this number. We tentatively want to say that the number of β-reductions is optimal under the constraint that we never reduce under a term $\lambda x.t$ before applying it, but properly defining what this means and proving it are still future work.

Part III.

Convertibility

9. An efficient strategy for convertibility

9.1. Motivations

9.1.1. Early failure

In the first part of this thesis, we approached the question of proving the convertibility of two λ-terms as a question of computing their normal forms, and then comparing them. However, we sometimes can conclude that t_1 and t_2 are not convertible much earlier. Indeed, if t_1 is x and t_2 is f (**exp2** 100) (with x, f free variables), then we can immediately conclude that they are not convertible, without computing[1] the normal form of **exp2** 100.

To do that, we simply need to compute the head-normal form of both terms, which we can compare before further computation.

9.1.2. Early success: controlling the unfolding of constants

When encountering a convertibility problem of the form $f\ t_1 \overset{?}{\equiv} f\ t_2$, where f is a defined constant, the naïve solution is to unfold f to get its definition d, and to continue with the problem $d\ t_1 \overset{?}{\equiv} d\ t_2$. However, sometimes we can conclude about convertibility much faster! Indeed, if $t_1 \equiv t_2$, then $f\ t_1 \equiv f\ t_2$, and we do not need to unfold f to be able to conclude this. When f is an expensive function to compute, this can make the convertibility test significantly faster, as can be seen on the problem **exp2** (**add** 99 1) $\overset{?}{\equiv}$ **exp2** 100.

However, if $t_1 \not\equiv t_2$, then we are not able to conclude anything about whether $f\ t_1$ and $f\ t_2$ are convertible without unfolding f. Indeed, if f is the identity function, they are not convertible, while if f is a constant function, they are convertible, so we need to know the definition of f to be able to conclude.

This gives us a first strategy, which is the one implemented in Coq: when trying to prove $f\ t_1 \equiv f\ t_2$, first try to prove the convertibility of t_1 and t_2, and if they are not convertible, then fall back to unfolding the definition of f. We will first show a way to implement this strategy in section 9.2 to introduce the idea behind our more efficient machine. However, while this is a good heuristic, it can lead to performing expensive

[1] Here and in the following, we assume we use Peano integers, which are represented in unary, so computing n takes time at least $O(n)$. As such, **exp2**, the function that associates 2^n to n, is the archetypal expensive function: when run on inputs of size $O(n)$, it takes time $O(2^n)$ to complete.

additional work. Indeed, if f is a constant function such as **always0** (defined as $\lambda x.0$), then **always0** (**exp2** 99) $\equiv$ **always0** (**exp2** 100), but trying to check the convertibility of **exp2** 99 and **exp2** 100 will take a very long time, while unfolding f and performing (lazy) β-reductions will prove convertibility instantaneously.

9.1.3. Parallel computation

At first the solution to the problem outlined above could be to find a good heuristic about whether we should unfold f or prove convertibility of its inputs. If we had an untrusted oracle for convertibility, we could simply ask the oracle whether t_1 and t_2 are convertible, and if the oracle answers "yes", check that t_1 and t_2 are indeed convertible, and if it answers "no", unfold f. Without such an oracle, it looks like our only option is to find a "good enough" heuristic. However, this relies on an incorrect premise that we need to either unfold f or to prove convertibility of its arguments first. Our approach tries to get the best of these ideas by trying to do both of these things *in parallel.* If one of the parallel computations is able to prove convertibility, we can abort the other and report that $f\ t_1 \equiv f\ t_2$. If both computations run to their end and are unable to prove convertibility, then the terms are not convertible and we have our answer as well.[2] Likewise, if we are considering the problem $x\ t_1\ t_2 \overset{?}{\equiv} x\ u_1\ u_2$, we examine in parallel the problems $t_1 \overset{?}{\equiv} u_1$ and $t_2 \overset{?}{\equiv} u_2$, and if one of them proves non-convertibility, then we know the initial terms were not convertible either, without having to decide which of these problems to examine first. A machine implementing these ideas will be presented in the remainder of this chapter, beginning in section 9.3.

9.2. Reduction to weak head normal form

We first show a well-known strategy, similar to the one used by Coq, for mixing convertibility and evaluation, and design an abstract machine that can be used for it. With such an abstract machine in mind, we will move on to a novel approach reducing terms in parallel, allowing to get faster results in many cases. In this basic strategy, we reduce terms to their head-normal form for $\to_\beta$ (and not $\to_{\beta\delta}$), and then compare both head-normal forms. If both head-normal forms have the same shape and the same head constant, for instance $f\ t_1 \overset{?}{\equiv} f\ t_2$, we can try to prove that corresponding arguments are convertible. If not all of them are convertible, or if the terms have a different shape and at least one of them has a head constant, we need to unfold the head constant and continue reducing. Except if we could prove convertibility earlier because we had the same head constant and all the arguments were convertible, we will finally end in a state where both head

[2] Technically speaking, we only need to prove non-convertibility when f has been unfolded, since $f\ t_1$ and $f\ t_2$ are convertible if and only if they are after f has been unfolded, while we have only one direction of implication if we consider the convertibility of t_1 and t_2. However, by the time we noticed this, the implementation was written and its Coq proof was almost finished, which is why we decided not to fix this minor inefficiency. Adapting the code and the proof should require no novel ideas, but we did not have the time to do so in the Coq proof.

will be variables, and we are in weak head normal form for $\to_{\beta\delta}$. In this case, both terms are convertible if and only if they have the same shape (either both λ-abstractions, or both inert terms with the same head variable), and all the corresponding subterms are convertible (note there can be none at all, which is the base case of our recursion): for instance, if x and y are both variables, $x\ t_1 \equiv y\ t_2$ if and only if $x = y$ and $t_1 \equiv t_2$.

In order to be able to do this efficiently, we need to reduce the terms to their head-normal form, while preserving the sharing of computations between consecutive runs. For instance, if the weak head normal forms are $x\ t_1$ and $x\ t_2$, we need to be able to share the computations done to compute these weak head normal forms with those for t_1 and t_2. This will be done using a global machine state with lazy values, that are updated after being computed to avoid recomputation, and shared between one computation and the next, so that any lazy value that was evaluated during the computation of $x\ t_1$ remains evaluated when computing the weak head-normal form of t_1. The reduction rules will be similar to the reductions to normal form, but with the reductions of arguments delayed to when they are needed. The machine has the property that once it has produced a weak head normal form, we can restart it with other terms (like t_1 and t_2 in our case), while preserving the store (which holds the results of computation of lazy values), to keep the sharing of computations between those successive runs.

This abstract machine has four components, like the LazyKAM shown in section 2.8. They serve the same purpose, so let us recall what they mean:

- the code: either a value or a pair of a term and an environment,
- the stack: a list of values,
- the dump: a list of computations waiting for the current computation to be forced,
- and the store: a mapping from references to lazy or computed values.

The values in the store are either:

- a lazy value $\mathtt{L}\ (t, e)$, corresponding to a term t that will need to be evaluated in environment e,
- a lazy application of a value to a stack, $\mathtt{A}\ (v, \pi)$: the result of applying v to π,
- a completed computation $\mathtt{D}\ v$ with result v,
- a marker that the value is being computed $\square$.

The values themselves can be one of:

- a store reference a, used when a lazy value is created so that we can update its result later,
- a λ-abstraction $(\lambda x.t, e, y, v)$, given by a term which is a λ-abstraction, an environment, a free variable name, and a value corresponding to the body of the λ-abstraction after reduction,

- an inert term $(x, \pi, \emptyset)$ or (c, π, v), which is given by a free variable or a constant, a stack, and in the case of a constant, a value corresponding to the result after unfolding the constant.

The last case leads us to store a constant c expanding to v as $(c, [], v)$. Other presentations use instead a global map from constants to their definitions. However, the global map approach is inferior to our approach when some constants are not in normal form, for instance if there are constants of the form $(\lambda n.\lambda x.t)$ (**exp2** 30), While unusual in user-written constants, this is very likely to occur as the result of λ-lifting. With a simple global map from constants to their definitions, we will either end up reducing the constant each time it is applied, or reducing the constant when we encounter it first, even if its weak head normal form is never needed.

Besides, we want to be able to share computation between partial applications of defined constants. For instance, consider $(\lambda u.f\ (u\ y)\ (u\ z))\ (c\ x)$. Here, $c\ x$ might make arbitrary computations, which we do not want to be duplicated when we evaluate independently $u\ y$ and $u\ z$. The value corresponding to $c\ x$ will then be of the form $(c, x :: [], a)$ with a being a reference to $\mathtt{A}\ (v, x :: [])$, whose computation will be shared among both partial applications.

The initial state of the machine for a given term t has the stack, dump and store empty, and code (t, e), where e is an environment binding the name of each constant c to its lazy normal form $\mathtt{L}\ (u, e')$, with u the body of the constant and e' the environment binding all previous constants (this takes only linear space thanks to sharing).

The final states of the machine, having computed a weak head normal form, correspond to the code being a value which is not lazy, and the stack and dump being empty. The weak head normal form corresponds to that value; and from there, the machine can be restarted with other (lazy) values to compute their weak head normal forms, simply by taking these values as the code of the machine, using an empty stack and dump, and, crucially to avoid recomputation, reusing the store.

The full rules of the machine are shown in Figure 9.1. Here is the explanation of what these rules do:

- $\rightsquigarrow_a$ handles application by pushing a lazy value corresponding to the argument on the stack;
- $\rightsquigarrow_l$ loads the value corresponding to variable x from the environment;
- $\rightsquigarrow_{f_1}$ and $\rightsquigarrow_{f_2}$ force a lazy value corresponding to a store reference by pushing the status of the stack to the dump and replacing the store reference by a marker $\square$;
- $\rightsquigarrow_{f_3}$ loads a lazy value that has already been computed;
- $\rightsquigarrow_\lambda$ creates a value from a λ-abstraction by creating a lazy value for the normal form of its body;

	Code	Stack	Dump	Store	
	$(t_1\ t_2, e)$	π	D	$\mathcal{E}$	
$\rightsquigarrow_a$	(t_1, e)	$a :: \pi$	D	$(a \mapsto \mathtt{L}\ (t_2, e)) \star \mathcal{E}$	
	(x, e)	π	D	$\mathcal{E}$	
$\rightsquigarrow_l$	$e(x)$	π	D	$\mathcal{E}$	
	a	π	D	$(a \mapsto \mathtt{L}\ (t, e)) \star \mathcal{E}$	
$\rightsquigarrow_{f_1}$	(t, e)	$[]$	$(a, \pi) :: D$	$(a \mapsto \square) \star \mathcal{E}$	
	a	π	D	$(a \mapsto \mathtt{A}\ (v, \pi_2)) \star \mathcal{E}$	
$\rightsquigarrow_{f_2}$	v	π_2	$(a, \pi) :: D$	$(a \mapsto \square) \star \mathcal{E}$	
	a	π	D	$(a \mapsto \mathtt{D}\ v) \star \mathcal{E}$	
$\rightsquigarrow_{f_3}$	v	π	D	$(a \mapsto \mathtt{D}\ v) \star \mathcal{E}$	
	$(\lambda x.t_1, e)$	π	D	$\mathcal{E}$	
$\rightsquigarrow_\lambda$	$(\lambda x.t_1, e, y, a)$	π	D	$(a \mapsto \mathtt{L}\ (t_1, e \star (x \mapsto (y, [], \emptyset)))) \star \mathcal{E}$	y fresh
	v	$[]$	$(x, \pi) :: D$	$(x \mapsto \square) \star \mathcal{E}$	
$\rightsquigarrow_s$	v	π	D	$(x \mapsto \mathtt{D}\ v) \star \mathcal{E}$	$v \neq a$
	$(\lambda x.t_1, e, y, v_1)$	$v_2 :: \pi$	D	$\mathcal{E}$	
$\rightsquigarrow_\beta$	$(t_1, e \star (x \mapsto v_2))$	π	D	$\mathcal{E}$	
	$(x, \pi_1, \emptyset)$	π_2	D	$\mathcal{E}$	
$\rightsquigarrow_{n_1}$	$(x, \pi_1 +\!\!+ \pi_2, \emptyset)$	$[]$	D	$\mathcal{E}$	if $\pi_2 \neq []$
	(c, π_1, v)	π_2	D	$\mathcal{E}$	
$\rightsquigarrow_{n_2}$	$(c, \pi_1 +\!\!+ \pi_2, a)$	$[]$	D	$(a \mapsto \mathtt{A}\ (v, \pi_2)) \star \mathcal{E}$	if $\pi_2 \neq []$

Figure 9.1.: Machine for reduction to weak head normal form

- $\rightsquigarrow_s$ records the result of a forced computation (when it is not itself a store reference) and resumes the previous computation;
- $\rightsquigarrow_\beta$ performs β-reduction by popping a value from the stack and adding it to the environment of the closure;
- $\rightsquigarrow_{n_1}$ and $\rightsquigarrow_{n_2}$ capture the stack when applying a neutral value, and, in the case of $\rightsquigarrow_{n_2}$, creates a new lazy value corresponding to the result after unfolding the constant.

The machine has computed a weak head normal form when the code is a value which is not a store reference, and both the stack and the dump are empty. We can then simply compare the results, using the strategy detailed above, re-running the machine if necessary to compute further head normal forms. When running the machine again, we simply reuse the store from the previous run, initialize the code with the value we want to reduce, an empty stack and an empty dump.

To compare weak head normal forms, we use a crucial property: two weak $\rightarrow_{\beta\delta}$ head normal forms that do not have a constant as the head are convertible if and only if they both have the same shape, and the subterms at the same positions are convertible. Formally, two weak head normal forms are convertible if and only if one of the following conditions is true:

- both head normal forms are λ-abstractions $\lambda x.t$ and $\lambda y.u$, and $t[x := z] \equiv u[y := z]$, where z is a fresh variable;
- both head normal forms are $x\ t_1\ \ldots\ t_n$ and $x\ u_1\ \ldots\ u_n$, with x a free variable, and for all i, $t_i \equiv u_i$;
- one head normal form has a constant as head, and after unfolding the constant and reducing again to head normal form, one of the three conditions is true.[3]

9.3. Parallel convertibility

Our approach for a more efficient machine relies on trying several convertibility problems in parallel. In order to do that, we have a set of *convertibility threads*, where each thread is a convertibility problem between two (partially reduced) terms. Each of these threads works in the same way as the convertibility test we have seen in the previous section, by reducing both terms to weak head normal form, comparing their shapes, and continuing with the arguments if the shapes match. The main difference is that instead of performing successive convertibility tests after reducing to weak head normal form, we now perform them in parallel by creating new convertibility threads, allowing us to sometimes give a result before all of these threads finish. For instance, if we consider $x\ t_1\ u_1 \overset{?}{\equiv} x\ t_2\ u_2$, we create two new convertibility threads for $t_1 \overset{?}{\equiv} t_2$ and $u_1 \overset{?}{\equiv} u_2$, and we run those

[3] This is well-defined, because there is no infinite chain of reductions, so this case can only happen a finite number of times.

in parallel: if either proves non-convertibility, we can conclude about non-convertibility without waiting for the other one.

9.3.1. Avoiding recomputation

When naïvely performing convertibility problems in parallel such as $t_1 \overset{?}{\equiv} t_2$ and $d\ t_1 \overset{?}{\equiv} d\ t_2$, there is duplicate work that is done when performing reductions inside t_1 or t_2. To avoid this problem, we add another kind of thread, called *reduction threads*. These threads perform reductions to find the weak head normal form of a term, and convertibility threads has exactly two reduction subthreads working to compute the weak head normal forms of the terms being tested. When forcing a lazy value, we start a new reduction thread for the computation of the lazy value, and wait for that reduction thread to return a result before resuming. If another thread wants to force the value as well, it will pause itself as well and wait for the thread that is running. When a reduction thread has finished running, all threads that were waiting for it (both convertibility and reduction threads) are resumed and can perform additional work based on the weak head normal form of the thread.

9.3.2. The thread structure

We therefore have a two-kinded structure for threads. When the machine is in a given state, we can draw the threads as an acyclic graph, where a thread p has an edge to a thread q if p is waiting for a result from q. Since reduction threads will never have a convertibility subthread, we can divide the threads into a group of convertibility threads and a group of reduction threads, with edges only inside a group or a convertibility thread to a reduction thread. Besides, convertibility threads have at most one parent (and exactly one if they are not the convertibility thread trying to solve the initial problem), while reduction threads have at most one child (another thread they are waiting for to continue their computation). A representation of the thread structure can be seen in Figure 9.2.

9.4. An abstract machine

9.4.1. Reduction threads

We first focus on reduction threads, since we can fully specify them independently of conversion threads. A reduction thread performs reduction to weak head normal form, and is a pair consisting in:

- the code, which is either a pair (t, e) of a term and an environment (mapping variables to values), or a value v,
- and a stack π, a list of values.

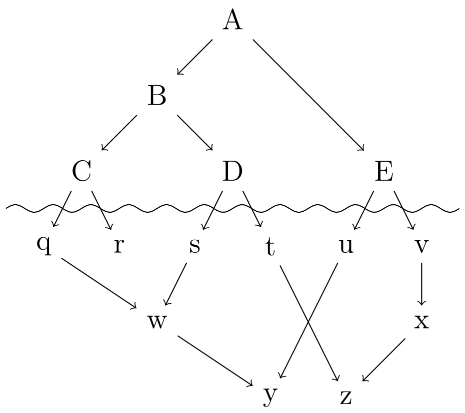


Figure 9.2.: The thread structure. The nodes above the middle line (uppercase letters) correspond to convertibility threads, while those below (lowercase letters) correspond to reduction threads.

Here, a value is one of:

- a thread identifier r, where the value is given by the (future) result of the thread,
- a λ-abstraction $(\lambda x.t, e, y, v)$, composed of a term which is a λ-abstraction, an environment, a free variable name, and a value corresponding to the body of the λ-abstraction after reduction,
- or an inert term $(x, \pi, \emptyset)$ or (c, π, v), composed of a free variable or a constant, a stack, and in the case of a constant, a value corresponding to the result after unfolding the constant,

Notice how we no longer have a concept of lazy values. Indeed, we use *thread values* instead, with lazy values encoded as thread values. Therefore, in the reduction rules below, we write **lazy**(t, e) to denote a thread reference r, where r is a new thread defined by:

Code	Stack
(t, e)	$[]$

We also note **finished**(r, v), and say that thread r *finished* with value v, if v is not a thread and thread r is equal to:

Code	Stack
v	$[]$

Finally, we write **apply**(v, π) to denote a thread reference r where r is a new thread defined by:

Code	Stack
v	π

Code	Stack		Code	Stack	
$(t_1\ t_2, e)$	π	$\rightsquigarrow_a$	(t_1, e)	$\mathbf{lazy}(t_2, e) :: \pi$	
(x, e)	π	$\rightsquigarrow_l$	$e(x)$	π	
$(\lambda x.t_1, e)$	π	$\rightsquigarrow_\lambda$	$(\lambda x.t_1, e, y, v)$ where $v = \mathbf{lazy}(t_1,$ $e \star (x \mapsto (y, [], \emptyset)))$	π	y fresh
r	π	$\rightsquigarrow_s$	v	π	if $\mathbf{finished}(r, v)$
$(\lambda x.t_1, e, y, v_1)$	$v_2 :: \pi$	$\rightsquigarrow_\beta$	$(t_1, e \star (x \mapsto v_2))$	π	
$(x, \pi_1, \emptyset)$	π_2	$\rightsquigarrow_{n_1}$	$(x, \pi_1 +\!\!+ \pi_2, \emptyset)$	$[]$	if $\pi_2 \neq []$
(c, π_1, v)	π_2	$\rightsquigarrow_{n_2}$	$(c, \pi_1 +\!\!+ \pi_2, \mathbf{apply}(v, \pi_2))$	$[]$	if $\pi_2 \neq []$

Figure 9.3.: Single-thread reduction rules of the abstract machine.

$$\begin{array}{rcll}
\bar{r}\langle(t_1\ t_2, e), \pi\rangle & \rightsquigarrow_a & \nu s.\bar{r}\langle(t_1, e), s :: \pi\rangle \mid \bar{s}\langle t_2, e\rangle & \\
\bar{r}\langle(x, e), \pi\rangle & \rightsquigarrow_l & \bar{r}\langle e(x), \pi\rangle & \\
\bar{r}\langle(\lambda x.t_1, e), \pi\rangle & \rightsquigarrow_\lambda & \nu s.\bar{r}\langle(\lambda x.t_1, e, y, s), \pi\rangle \mid \bar{s}\langle t_1, e \star (x \mapsto (y, [], \emptyset))\rangle & y \text{ fresh} \\
\bar{r}\langle s, \pi\rangle \mid \bar{s}\langle v, []\rangle & \rightsquigarrow_s & \bar{r}\langle v, \pi\rangle \mid \bar{s}\langle v, []\rangle & v \text{ not a thread} \\
\bar{r}\langle(\lambda x.t_1, e, y, v_1), v_2 :: \pi\rangle & \rightsquigarrow_\beta & \bar{r}\langle(t_1, e \star (x \mapsto v_2)), \pi\rangle & \\
\bar{r}\langle(x, \pi_1, \emptyset), \pi_2\rangle & \rightsquigarrow_{n_1} & \bar{r}\langle(x, \pi_1 +\!\!+ \pi_2, \emptyset), []\rangle & \text{if } \pi_2 \neq [] \\
\bar{r}\langle(c, \pi_1, v), \pi_2\rangle & \rightsquigarrow_{n_2} & \nu s.\bar{r}\langle(c, \pi_1 +\!\!+ \pi_2, s), []\rangle \mid \bar{s}\langle v, \pi_2\rangle & \text{if } \pi_2 \neq []
\end{array}$$

Figure 9.4.: Reduction rules of the abstract machine, with a π-calculus inspired syntax.

The reduction rules we obtain for a single thread are show in Figure 9.3. Once again, we have a machine satisfying the subterm property, and we will see in the next chapter how it can be transformed into a virtual machine.

Notice that there is almost always a reduction rule that applies, in which case we say our thread is *running*, except in the following two cases:

- code is r and thread r is not finished, in which case our thread is *waiting* on thread r to perform more computations,
- code is v and stack is $[]$, in which case our thread is finished and the other threads waiting on it can resume their work.

We can easily show that threads containing references to other threads form an acyclic graph, therefore it is also the case for threads waiting on other threads. Besides, since a thread by definition cannot be waiting on a finished thread, either all threads are finished or there is at least one running thread.

We can also notice that this presentation of the machine sometimes creates useless threads, such as the one for the normal form of a λ-abstraction when immediately applying it. In practice, we can add a combined rule so this does not happen:

$$\begin{array}{c|c c c|c} \text{Code} & \text{Stack} & & \text{Code} & \text{Stack} \\ (\lambda x.t_1, e) & v :: \pi & \rightsquigarrow_{\lambda\beta} & (t_1, e \star (x \mapsto v)) & \pi \end{array}$$

Another way to see the rules is inspired from process calculi, which we present in Figure 9.4. We do not fully encode the rules into the π-calculus, but instead simply take inspiration from it to write the rules shown: we write $\bar{r}\langle c, \pi\rangle$ for a thread r with code c and stack π, while νr creates a new thread name r.

9.4.2. Convertibility threads

Since convertibility threads have a tree structure, we can specify a complete convertibility problem with an inductive definition. Thus, we now have the following grammar for convertibility threads:

$$c ::= (v_1, v_2, \xi) \mid \top \mid \bot \mid c_1 \wedge c_2 \mid c_1 \vee c_2$$

Here, (v_1, v_2, ξ) means that value v_1 is convertible with value v_2, up to variables that need to be renamed with ξ, $\top$ means the terms this convertibility thread started with are convertible, $\bot$ that they are not, $c_1 \wedge c_2$ that the result is the conjunction of the two convertibility subproblems, and $c_1 \vee c_2$ that it is the disjunction of them.

CReduce1

$$\frac{\textbf{finished}(r_1, v_1)}{(r_1, v_2, \xi) \rightsquigarrow (v_1, v_2, \xi)}$$

CReduce2

$$\frac{\textbf{finished}(r_2, v_2)}{(v_1, r_2, \xi) \rightsquigarrow (v_1, v_2, \xi)}$$

CAnd1

$$\frac{c_1 \rightsquigarrow c_1'}{c_1 \land c_2 \rightsquigarrow c_1' \land c_2}$$

CAnd2

$$\frac{c_2 \rightsquigarrow c_2'}{c_1 \land c_2 \rightsquigarrow c_1 \land c_2'}$$

COr1

$$\frac{c_1 \rightsquigarrow c_1'}{c_1 \lor c_2 \rightsquigarrow c_1' \lor c_2}$$

COr2

$$\frac{c_2 \rightsquigarrow c_2'}{c_1 \lor c_2 \rightsquigarrow c_1 \lor c_2'}$$

CAndTrue

$$\frac{}{\top \land \top \rightsquigarrow \top}$$

CAndFalse1

$$\frac{}{\bot \land c \rightsquigarrow \bot}$$

CAndFalse2

$$\frac{}{c \land \bot \rightsquigarrow \bot}$$

COrFalse

$$\frac{}{\bot \lor \bot \rightsquigarrow \bot}$$

COrTrue1

$$\frac{}{\top \lor c \rightsquigarrow \top}$$

COrTrue2

$$\frac{}{c \lor \top \rightsquigarrow \top}$$

Figure 9.5.: Structural rules for convertibility threads

We first start with the structural rules for the convertibility threads shown in Figure 9.5. The rules CReduce1 and CReduce2 express the fact that we are allowed to substitute a finished thread with its result; the rules CAnd1, CAnd2, COr1 and COr2 allow reduction of convertibility threads under conjunctions and disjunctions; the remaining rules express the logic properties of short-circuiting conjunction and disjunction.

The rules show in Figure 9.6 are for the convertibility tests of two non-thread values. In the case where neither side has a constant that could be unfolded, the test is easy to perform. If there are constants, a possibility is always to unfold one of the constants, and if both sides have the same constant, we can also try to prove convertibility of the arguments. To express the convertibility of two stacks $\pi = \overline{v}$ and $\pi' = \overline{v'}$, we write $\textbf{cstk}(\pi, \pi', \xi)$ to denote $\bot$ if $|\pi| \neq |\pi'|$, and $(v_1, v_1', \xi) \land \cdots \land (v_n, v_n', \xi)$ otherwise.

In Figure 9.6, the rule CAbs deals with the convertibility of two λ-abstractions, recording that their variables correspond in ξ, and testing the convertibility of their bodies. CVarEq and CVarNEq deal with convertibility of free variables, expressing that they must be the same variable, with stacks that are convertible with one another, while CAbsVar and CVarAbs express that variables and λ-abstractions are never convertible.[4] Finally, CUnfold1 and CUnfold2 are rules for unfolding constants on the left or on the right, while CConst expresses that if we have the same constant on each side, then

[4] If we wanted to support η-expansion as well, these would be the two rules we would need to modify.

$$\textsc{CAbs}\quad \frac{}{((\lambda x_1.t_1, e_1, y_1, v_1), (\lambda x_2.t_2, e_2, y_2, v_2), \xi) \rightsquigarrow (v_1, v_2, (y_1, y_2) :: \xi)}$$

$$\textsc{CVarEq}\quad \frac{(x_1, x_2) \in \xi}{((x_1, \pi_1, \emptyset), (x_2, \pi_2, \emptyset), \xi) \rightsquigarrow \mathbf{cstk}(\pi_1, \pi_2, \xi)} \qquad \textsc{CVarNEq}\quad \frac{(x_1, x_2) \notin \xi}{((x_1, \pi_1, \emptyset), (x_2, \pi_2, \emptyset), \xi) \rightsquigarrow \bot}$$

$$\textsc{CAbsVar}\quad \frac{}{((\lambda x_1.t_1, e_1, y_1, v_1), (x_2, \pi_2, \emptyset), \xi) \rightsquigarrow \bot} \qquad \textsc{CVarAbs}\quad \frac{}{((x_1, \pi_1, \emptyset), (\lambda x_2.t_2, e_2, y_2, v_2), \xi) \rightsquigarrow \bot}$$

$$\textsc{CUnfold1}\quad \frac{}{((c_1, \pi_1, v_1), v_2, \xi) \rightsquigarrow (v_1, v_2, \xi)} \qquad \textsc{CUnfold2}\quad \frac{}{(v_1, (c_2, \pi_2, v_2), \xi) \rightsquigarrow (v_1, v_2, \xi)}$$

$$\textsc{CConst}\quad \frac{}{((c, \pi_1, v_1), (c, \pi_2, v_2), \xi) \rightsquigarrow \mathbf{cstk}(\pi_1, \pi_2, \xi) \vee (v_1, v_2, \xi)}$$

Figure 9.6.: Convertibility rules for non-thread values

convertibility of the arguments is enough to prove convertibility of the complete term, but we also need to unfold the constant in case the arguments are not convertible.

9.5. Putting threads to sleep

Due to the presence of short-circuiting evaluation of $\wedge$ and $\vee$, sometimes a convertibility thread is running but its result will never be used. Although the convertibility threads themselves can simply be garbage-collected, they can have references to reduction threads for which this is not as simple. Indeed, we do not want to perform reductions inside a thread whose value will not matter anyway, but although a subthread in this form may no longer be needed for now, it might be used later. To avoid this, we add a reference count to each thread, indicating whether a thread is *active*: an active thread is one for which the reference count is non-zero. We enforce the invariant that the reference count of a thread is equal to the number of times the thread appears in an *active position*, which are the values of a basic conversion thread, or directly as the code of another active thread.[5] Then, only active threads are allowed to be reduced by the rules for reduction threads, and inactive threads are not said to be running. For instance, in Figure 9.2, the reduction threads r, y and z are running, while all other reduction threads are inactive.

[5] Although this definition may appear to be cyclic, the fact that threads holding references to one another form an acyclic graph ensure that it is well-defined.

Once this is done, the changes to the reduction rules are minimal: besides only allowing active threads to be reduced and initializing new threads created with $\textbf{lazy}(t, e)$ or $\textbf{apply}(v, \pi)$ with a reference count of 0 (since they are not in an active position), the only change to the reduction rules is that in $\leadsto_l$, where we replace the code (x, e) with $e(x)$, we need to increment the reference count of $e(x)$ if it is a thread. This will then have the additional effect of starting the thread if it was a lazy value, since the reference count will rise from 0 to 1 and the thread will become active. Technically, we would also need to modify the rule $\leadsto_s$ to decrease the reference count of r, but since finished threads can never be reduced again, their reference count does not matter.

The changes to the rules for convertibility threads are slightly more extensive. First, in all rules that add values in active positions need to increase the reference count of those values if they are threads. This applies to v_1 and v_2 in CAbs and CConst, v_1 in CUnfold1 and v_2 in CUnfold2, as well as all the π_{1i} and π_{2i} when we create a $\textbf{cstk}(\pi_1, \pi_2, \xi)$ thread, if $|\pi_1| = |\pi_2|$. Moreover, and this is why we needed to add these reference counts, in all short-circuit rules (CAndFalse1, CAndFalse2, COrTrue1 and COrTrue2), the reference counts of all reduction threads that appear in active positions in the deleted thread c need to be decremented according to their multiplicities.

Finally, since the notion of active position depends on which threads are active, changing the reference count of a thread r_1 whose code is another thread r_2 needs to increase the reference count of r_2 if r_1 becomes active, and decrease it if r_1 becomes inactive.

In a way, this is a form of garbage collection with reference counting: we have an acyclic graph of threads with edges meaning that the result of a thread is needed by another thread, and we want to know which threads are accessible from a given set of roots (active positions in the global convertibility thread). However, unlike garbage collection, inactive threads will not necessarily stay inactive forever, and may become active again if another thread requests their value. For instance, starting from a configuration like in Figure 9.2, if the result of C is no longer needed, r will become inactive; but it could become active again if another dependency appears, for instance if z needs the result of r after some more computation. Besides, we want to continuously know which threads are active, since they are the ones which are allowed to run, so adapting other strategies of garbage collection would probably not work as well.

On the other hand, garbage collection remains an interesting question for our abstract machine. Indeed, we want threads (especially lazy values) and other constructions that can no longer change anything about computation to be completely deleted. To that end, we rely on the garbage collection of the implementation.Still, we need to ensure that a thread that is no longer relevant has no dangling pointer to it, so we cannot keep a list of all threads.

Since we need to be able to reach the threads that have to run, we still keep a list of active threads, which we update by adding or removing threads when they change status. This alone would be enough to ensure garbage collection, but we can combine it with another optimisation: instead of keeping a list of active threads, we keep a list of *running* threads. Those threads are precisely those which can perform a reduction, and this is exactly what we need to write an efficient scheduler that picks a running thread, performs a reduction step, and then continues with the next thread.

To maintain this list of running threads, we need to see in which conditions a thread can start or stop running. The first possibility, which we have already considered, is a thread changing its active status. Since we have the reference count, we can simply add or remove the thread from the list of running threads if the active status change causes a running status change. The other possibilities are related to thread termination and waiting: a thread can stop running if it becomes finished, or if it blocks waiting on another thread, and it can restart running if the thread it was waiting on finishes. Then, a way to maintain this list of running threads is to add yet another field to threads, which are a list of the threads currently waiting on it. Then, when a thread starts waiting on another thread, it deletes itself from the list of running threads and adds itself to the list of threads waiting on the child thread. Conversely, when a thread finishes, it adds back all threads waiting for it in the list of running threads. Finally, we need to make sure no inactive thread gets added to the list of running threads, and to do so, when a thread waiting on another becomes inactive (respectively, becomes active), it removes itself (resp. adds itself) from the list of threads waiting on its child. With these changes, we always maintain the list of all running threads, and for a given thread, the length of the list of threads waiting for it is equal to its reference count![6] Besides, there are no more references to non-running threads except in positions which could potentially cause the thread to be resumed later, thus making garbage collection possible. Finally, the cost of bookkeeping is not excessive: if we use doubly-linked lists for all the lists mentioned here, the additions and deletions can be done in $O(1)$ time.[7]

The full rules for the reduction part of this machine are shown in Figure 9.7. There, we assume that W, W_2 are non-empty lists of threads (while W_ε is a possibly empty list of threads), while y is a fresh variable and r_f is a fresh thread reference. There are three additional rules compared to the presentation in Figure 9.3: $\rightsquigarrow_d$, $\rightsquigarrow_{f_1}$ and $\rightsquigarrow_{f_2}$. This is no surprise, as these are the only three rules concerned with thread scheduling: all the other rules are simply isomorphic to the single-thread rules, and put the thread that just ran at the end of the list of active threads; see the presentation of these rules in the single-threaded case for details. On the contrary, rule $\rightsquigarrow_d$ is concerned with a thread that just finished running, and wakes all other threads depending on it, while rules $\rightsquigarrow_{f_1}$ and $\rightsquigarrow_{f_2}$ deal with a thread wanting the result of a thread that has not yet finished running; in that case, it adds itself to the list of threads waiting on the second thread, and adds it to the list of active threads if needed.

[6] Here, we consider that convertibility subthreads with values being a given thread to be waiting on this thread as well.

[7] We can unfortunately still get a linear behaviour: If a thread r_1 is waiting for a single thread r_2, itself waiting for a single thread r_3 and so on until r_n, putting r_1 to sleep will iterate over all these threads until r_n which will be put to sleep. Then, restarting r_1 will iterate again over all the threads until r_n which will be restarted, resulting in $O(n)$ behaviour. This can probably be improved by using a better data structure for thread dependencies, but we did not investigate this further.

	Threads	Active
	$(r \mapsto (\mathtt{C}\ (t_1\ t_2, e), \pi, W)) \star T$	$r :: A$
$\rightsquigarrow_a$	$(r \mapsto (\mathtt{C}\ (t_1, e), W, r_f :: \pi)) \star (r_f \mapsto (\mathtt{C}\ (t_2, e), [], [])) \star T$	$A :: r$
	$(r \mapsto (\mathtt{C}\ (x, e), \pi, W)) \star T$	$r :: A$
$\rightsquigarrow_l$	$(r \mapsto (\mathtt{V}\ e(x), \pi, W)) \star T$	$A :: r$
	$(r \mapsto (\mathtt{C}\ (\lambda x.t, e), \pi, W)) \star T$	$r :: A$
$\rightsquigarrow_\lambda$	$(r \mapsto (\mathtt{V}\ (\lambda x.t, e, y, r_f), \pi, W)) \star (r_f \mapsto (\mathtt{C}\ (t, e \star (x \mapsto (y, [], \emptyset))), [], [])) \star T$	$A :: r$
	$(r \mapsto (\mathtt{V}\ v, [], W)) \star T$	$r :: A$
$\rightsquigarrow_d$	$(r \mapsto (\mathtt{V}\ v, [], [])) \star T$	$A \mathbin{+\!\!+} W$
	$(r \mapsto (\mathtt{V}\ r_2, \pi, W)) \star (r_2 \mapsto (\mathtt{V}\ v, [], W_\varepsilon)) \star T$	$r :: A$
$\rightsquigarrow_s$	$(r \mapsto (\mathtt{V}\ v, \pi, W)) \star (r_2 \mapsto (\mathtt{V}\ v, [], W_\varepsilon)) \star T$	$A :: r$
	$(r \mapsto (\mathtt{V}\ r_2, \pi, W)) \star (r_2 \mapsto (c, \pi_2, [])) \star T$	$r :: A$
$\rightsquigarrow_{f_1}$	$(r \mapsto (\mathtt{V}\ r_2, \pi, W)) \star (r_2 \mapsto (c, \pi_2, r :: [])) \star T$	$A :: r_2$
	$(r \mapsto (\mathtt{V}\ r_2, \pi, W)) \star (r_2 \mapsto (c, \pi_2, W_2)) \star T$	$r :: A$
$\rightsquigarrow_{f_2}$	$(r \mapsto (\mathtt{V}\ r_2, \pi, W)) \star (r_2 \mapsto (c, \pi_2, r :: W_2)) \star T$	A
	$(r \mapsto (\mathtt{V}\ (\lambda x.t_1, e, y, v_1), v_2 :: \pi, W)) \star T$	$r :: A$
$\rightsquigarrow_\beta$	$(r \mapsto (\mathtt{C}\ (t_1, e \star (x \mapsto v_2)), \pi, W)) \star T$	$A :: r$
	$(r \mapsto (\mathtt{V}\ (x, \pi_1, \emptyset), \pi_2, W)) \star T$	$r :: A$
$\rightsquigarrow_{n_1}$	$(r \mapsto (\mathtt{V}\ (x, \pi_1 \mathbin{+\!\!+} \pi_2, \emptyset), [], W)) \star T$	$A :: r$
	$(r \mapsto (\mathtt{V}\ (c, \pi_1, v), \pi_2, W)) \star T$	$r :: A$
$\rightsquigarrow_{n_2}$	$(r \mapsto (\mathtt{V}\ (c, \pi_1 \mathbin{+\!\!+} \pi_2, r_f), [], W)) \star (r_f \mapsto (\mathtt{V}\ v, \pi_2, [])) \star T$	$A :: r$

Figure 9.7.: Rules for the abstract machine.

9.6. Improved convertibility rules

While the formulation of the convertibility rules we showed are correct and what has been both formalised and implemented, we actually need slightly different rules if we want to have a complexity guarantee, as studied in chapter 12. Indeed, when we try to prove or disprove convertibility between $c\ t_1$ and $c\ t_2$, while non-convertibility between t_1 and t_2 gives no information about convertibility of $c\ t_1$ and $c\ t_2$, non-convertibility of the unfolded versions is enough to prove non-convertibility of $c\ t_1$ and $c\ t_2$.

To get complexity guarantees, we introduce two additional connectors: $\parallel$ and $\vee\!\!\parallel$, which run two convertibility threads concurrently. For $c_1 \parallel c_2$, we assume both c_1 and c_2 will yield the same result, so evaluation stops as soon as one of c_1 and c_2 has computed a value. For $c_1 \vee\!\!\parallel c_2$, the result is c_2 but assuming $c_1 \Rightarrow c_2$, so that evaluation can stop as soon as c_1 is true, or as soon as c_2 terminates.

$$\textsc{CPar1}\quad \frac{b \in \{\top, \bot\}}{b \parallel c \rightsquigarrow b} \qquad \textsc{CPar2}\quad \frac{b \in \{\top, \bot\}}{c \parallel b \rightsquigarrow b} \qquad \textsc{CParOrTrue1}\quad \frac{}{\top \vee\!\!\parallel c \rightsquigarrow \top} \qquad \textsc{CParOr2}\quad \frac{b \in \{\top, \bot\}}{c \vee\!\!\parallel b \rightsquigarrow b}$$

We also need to restrict CUnfold1 and CUnfold2 to the case where the other value is not a constant, as well as modify CConst as following and add another rule CUnfold12:

$$\textsc{CUnfold12}\quad \frac{c_1 \neq c_2}{((c_1, \pi_1, v_1), (c_2, \pi_2, v_2), \xi) \rightsquigarrow ((c_1, \pi_1, v_1), v_2, \xi) \parallel (v_1, (c_2, \pi_2, v_2), \xi)}$$

$$\textsc{CConst}\quad \frac{}{((c, \pi_1, v_1), (c, \pi_2, v_2), \xi) \rightsquigarrow \mathbf{cstk}(\pi_1, \pi_2, \xi) \vee\!\!\parallel (((c, \pi_1, v_1), v_2, \xi) \parallel (v_1, (c, \pi_2, v_2), \xi))}$$

This would not change much in the formalisation, but this is only useful if we add sharing of convertibility threads, as mentioned in section 9.7: we would otherwise create an extremely large number of duplicate convertibility threads when repeatedly applying those rules.

We need to modify CConst because in some cases where the arguments are not convertible, we should unfold only one side. Consider for instance the function f defined as: $\lambda b.\lambda n.\mathbf{if}\ b\ \mathbf{then}\ n\ \mathbf{else}\ \mathbf{exp2}\ n$, where **exp2** is not a defined constant but has been inlined inside the definition of f. On the convertibility problem $f\ \mathbf{true}\ (f\ \mathbf{false}\ 100) \overset{?}{\equiv} f\ \mathbf{false}\ 100$, we need to unfold f on the left only: the arguments are clearly not convertible, but unfolding f on both sides would need to the expensive computation of **exp2** 100, which is not needed if we unfold f on the left side only.

9.7. Sharing (dis)equality proofs

In the machine we presented, there is sharing of reductions, thanks to the threads, and sharing of subterms, such as in the reduction of $(\lambda x.\lambda y.y\ x\ x)\ t$. However, consider a problem such as $(\lambda x.\lambda y.y\ x\ x)\ t \overset{?}{\equiv} (\lambda x.\lambda y.y\ x\ x)\ u$, which becomes $\lambda y.y\ t\ t \overset{?}{\equiv} \lambda y.y\ u\ u$ with both instances of t and both instances of u shared. In this case, we will obtain two identical conversion subthreads trying to solve the problem $t \overset{?}{\equiv} u$. While this is only rarely a problem since most of the work happens in reduction threads, it can lead to an exponential blowup in the number of conversion threads in some cases.

Fortunately for us, we already have most of the machinery available to solve this problem. However, we must first notice that we cannot do naïve memoization (or a naïve union-find structure) to solve it. Indeed, since we can have threads performing computations in parallel, one thread may still be computing such a convertibility test while another wants to perform the same test. The solution is of course the same as with reduction: when we perform a convertibility test, we associate a new thread to the pair of values we compare, and the result of the comparison will be the result of this thread. If another thread wants to perform the same convertibility test, it needs to wait for the first thread to finish computing the result and can then just report it.

The question remains about how best to exploit known equalities. Unfortunately, there seems to be no simple answer about this. Indeed, even if we know that $t_1 \equiv t_2$, then the tests $t_1 \overset{?}{\equiv} t_3$ and $t_2 \overset{?}{\equiv} t_3$ can have widely different costs. For instance, consider the case where t_1 is **exp2** 30, t_2 is 2^{30}, and t_3 is **exp2** (**add** 29 1). Testing the convertibility of t_1 and t_2 takes quite a bit of time, but then the test $t_1 \overset{?}{\equiv} t_3$ is almost instantaneous, while the test $t_2 \overset{?}{\equiv} t_3$ will cost as much as the first test. Besides, any rule for choosing a canonical member of the equivalence class will have to take into account the other term!

Even if we had a way to choose which of these tests to run, it still wouldn't be enough. Indeed, we could have three concurrent tests running (since we have parallelism), which would be checking if $t_1 \overset{?}{\equiv} t_2$, $t_1 \overset{?}{\equiv} t_3$ and $t_2 \overset{?}{\equiv} t_3$. If one of these tests finishes with a proof of convertibility, what should be done about the other tests? It seems difficult to decide which of the others needs to be stopped and which is allowed to continue running, since one can finish far faster than the other.

In our case, we want to get worst-case time bounds, so instead of using heuristics in these cases, we can choose not to exploit known equalities. Thus, if we have $t_1 \equiv t_2$, then we will test $t_1 \overset{?}{\equiv} t_3$ and $t_2 \overset{?}{\equiv} t_3$ independently, as if they could give different results (but the reduction of t_3 to head-normal form is still shared). With this version, we have at most one convertibility test for each pair of values, and the number of values is linearly bounded by the number of reduction steps.[8]

[8] More precisely, we have at most one convertibility test for each pair of values, where the first one is a reduct from the first initial term, and the second one a reduct from the second initial term, which bounds the total number of convertibility tests in a bilinear way. This is a lot better if one of the initial terms is small and creates only a few values.

The question of whether we can better exploit known equalities or inequalities remains future work, and deserves to be explored further.

10. A virtual machine

The abstract machine we saw in the previous chapter satisfies the subterm property on each thread. For improved execution performance, we would like to translate it to a virtual machine, like we can do with other abstract machines. Since the subterm property holds on each thread, we will again need to have several threads, each with its own code. We will thus first write a simple translation of our abstract machine to a virtual one, then borrow ideas from the literature to optimise the performance of the virtual machine.

10.1. A first simple translation

To start with a simple translation, we use the fact that the machine satisfies the subterm property, so we can associate sequences of opcodes to each subterm, as we did in section 2.7 with other machines. However, since we sometimes have a value in code position instead of a subterm and an environment, this will correspond to intermediate states where part of the sequence of opcodes associated to a term are executed but not all of them. We will extend the state of threads to have an *accumulator* besides the stack to hold the value in such cases, as well as a *return stack* used to thread various pieces of code together.

We can translate terms to sequences of opcodes as follows:

$$\begin{aligned}
[\![t_1\ t_2]\!]_\rho &= \texttt{LAZY}([\![t_2]\!]_\rho; \texttt{RET}); [\![t_1]\!]_\rho; \texttt{APP} \\
[\![x]\!]_\rho &= \texttt{VAR}(\mathbf{index}_\rho(x)); \texttt{FORCE} \\
[\![\lambda x.t]\!]_\rho &= \texttt{ABS}([\![t]\!]_{x::\rho}; \texttt{RET})
\end{aligned}$$

Here, we see six different opcodes, for which we give intuitive semantics below.

- `LAZY` creates a new thread with the code given to it as argument and the current environment, and pushes a reference to this thread on the stack,
- `APP` applies the accumulator to the top of the stack, pushing the rest of the current code to the return stack,
- `VAR` fetches the numbered variable from the environment and puts it in the accumulator,
- `FORCE` ensures the value of the accumulator is not a thread reference, by waiting for its completion if necessary,

- ABS puts a closure with the corresponding code in the accumulator,
- and RET either pops the return stack and resumes computation after the APP call, or exits the current thread with the value of the accumulator if the return stack is empty.

The rules, presented in Figure 10.1 are extremely close to the rules of the abstract machine of the previous chapter. There are a couple differences, however: the new FORCE opcode needs to do something when the accumulator is not a thread, while the previous machine could directly continue the computation. This is captured by rule $\rightsquigarrow_v$: if the accumulator is not a thread value, it is unchanged. Another change is the addition of the RET opcode, which is used to thread together the call stack; this is simply rule $\rightsquigarrow_r$, which would again correspond to an identity operation on the previous machine. Finally, the application of inert terms will need to happen over many APP opcodes, instead of a single application rules like we had previously. Fortunately, the change is relatively straightforward, leading to rules $\rightsquigarrow_{n_1}$ and $\rightsquigarrow_{n_2}$ which take a single value from the stack instead of the complete stack at once.

10.2. Striving for efficiency

While correct, the virtual machine above can be improved by drawing inspiration from ideas in the literature, in particular from the OpenZAM machine [GL02]. We can also see that π and s can share a single stack, since the pushes and pops are always well-balanced.

There are two main ideas in the OpenZAM. The first is to use ZAM's mechanism for multiple application to avoid building intermediate closures when applying an n-ary function to n arguments. The second is to optimise for the normal computation case, by avoiding special cases in the application of functions, and instead having free variables being functions with an opcode ACCU which captures the arguments, as the case for application of a free variable would normally do.

With these ideas in mind, we can remove the return stack from the previous definition of the machine, and replace it by a number-of-arguments counter as in the ZAM. The compilation function does not change much, but is now specified in continuation-style, where $[\![t]\!]_\rho\ c$ loosely corresponds to $[\![t]\!]_\rho; c$.

$$\begin{aligned}
[\![t\ \overline{u}^n]\!]_\rho\ c &= \texttt{PUSHRETADDR}(c); \mathbf{rev}(\overline{\texttt{LAZY}([\![u]\!]_\rho\ \texttt{RET})}); [\![t]\!]_\rho\ \texttt{APPLY}(n)\\
[\![x]\!]_\rho\ c &= \texttt{VAR}(\mathbf{index}_\rho(x)); \texttt{FORCE}; c\\
[\![\lambda\overline{x}^n.t]\!]_\rho\ c &= \texttt{ABS}(\texttt{GRAB}^n; [\![t]\!]_{\mathbf{rev}(\overline{x}^n)+\rho}\ \texttt{RET}); c
\end{aligned}$$

There are several new opcodes and some changes to previous opcodes:

	Threads	Active
$\rightsquigarrow_a$	$(r \mapsto \mathtt{R}\ (\mathtt{LAZY}(c_2); c_1, e, a, \pi, s, W)) \star T$	$r :: A$
	$(r \mapsto \mathtt{R}\ (c_1, e, a, r_f :: \pi, s, W)) \star (r_f \mapsto \mathtt{R}\ (c_2, e, \emptyset, [], [], [])) \star T$	$A :: r$
$\rightsquigarrow_l$	$(r \mapsto \mathtt{R}\ (\mathtt{VAR}(i); c, e, a, \pi, s, W)) \star T$	$r :: A$
	$(r \mapsto \mathtt{R}\ (c, e, e(i), \pi, s, W)) \star T$	$A :: r$
$\rightsquigarrow_\lambda$	$(r \mapsto \mathtt{R}\ (\mathtt{ABS}(c_1); c_2, e, a, \pi, s, W)) \star T$	$r :: A$
	$(r \mapsto \mathtt{R}\ (c_2, e, (c_1, e, y, r_f), \pi, s, W)) \star (r_f \mapsto \mathtt{R}\ (c_1, (y, [], \emptyset) :: e, \emptyset, [], [], [])) \star T$	$A :: r$
$\rightsquigarrow_r$	$(r \mapsto \mathtt{R}\ (\mathtt{RET}, e, v, \pi, c :: s, W)) \star T$	$r :: A$
	$(r \mapsto \mathtt{R}\ (c, e, v, \pi, s, W)) \star T$	$A :: r$
$\rightsquigarrow_d$	$(r \mapsto \mathtt{R}\ (\mathtt{RET}, e, v, \pi, [], W)) \star T$	$r :: A$
	$(r \mapsto \mathtt{D}\ v) \star T$	$A +\!\!+ W$
$\rightsquigarrow_v$	$(r \mapsto \mathtt{R}\ (\mathtt{FORCE}; c, e, v_{\neg r}, \pi, s, W)) \star T$	$r :: A$
	$(r \mapsto \mathtt{R}\ (c, e, v_{\neg r}, \pi, s, W)) \star T$	$A :: r$
$\rightsquigarrow_s$	$(r \mapsto \mathtt{R}\ (\mathtt{FORCE}; c, e, r_2, \pi, s, W)) \star (r_2 \mapsto \mathtt{D}\ v) \star T$	$r :: A$
	$(r \mapsto \mathtt{R}\ (c, e, v, \pi, s, W)) \star (r_2 \mapsto \mathtt{D}\ v) \star T$	$A :: r$
$\rightsquigarrow_{f_1}$	$(r \mapsto \mathtt{R}\ (\mathtt{FORCE}; c, e, r_2, \pi, s, W)) \star (r_2 \mapsto \mathtt{R}\ (c, e_2, a_2, \pi_2, s_2, [])) \star T$	$r :: A$
	$(r \mapsto \mathtt{R}\ (\mathtt{FORCE}; c, e, r_2, \pi, s, W)) \star (r_2 \mapsto \mathtt{R}\ (c, e_2, a_2, \pi_2, s_2, r :: [])) \star T$	$A :: r_2$
$\rightsquigarrow_{f_2}$	$(r \mapsto \mathtt{R}\ (\mathtt{FORCE}; c, e, r_2, \pi, s, W)) \star (r_2 \mapsto \mathtt{R}\ (c, e_2, a_2, \pi_2, s_2, W_2)) \star T$	$r :: A$
	$(r \mapsto \mathtt{R}\ (\mathtt{FORCE}; c, e, r_2, \pi, s, W)) \star (r_2 \mapsto \mathtt{R}\ (c, e_2, a_2, \pi_2, s_2, r :: W_2)) \star T$	A
$\rightsquigarrow_\beta$	$(r \mapsto \mathtt{R}\ (\mathtt{APP}; c_1, e, (c_2, e_2, y, v_1), v_2 :: \pi, s, W)) \star T$	$r :: A$
	$(r \mapsto \mathtt{R}\ (c_2, v_2 :: e_2, \emptyset, \pi, c_1 :: s, W)) \star T$	$A :: r$
$\rightsquigarrow_{n_1}$	$(r \mapsto \mathtt{R}\ (\mathtt{APP}; c_1, e, (x, \pi_1, \emptyset), v :: \pi_2, s, W)) \star T$	$r :: A$
	$(r \mapsto \mathtt{R}\ (c_1, e, (x, \pi_1 :: v, \emptyset), \pi_2, s, W)) \star T$	$A :: r$
$\rightsquigarrow_{n_2}$	$(r \mapsto \mathtt{R}\ (\mathtt{APP}; c_1, e, (c, \pi_1, v_1), v_2 :: \pi_2, s, W)) \star T$	$r :: A$
	$(r \mapsto \mathtt{R}\ (c_1, e, (c, \pi_1 :: v_2, r_f), \pi_2, s, W)) \star (r_f \mapsto \mathtt{R}\ (\mathtt{APP}; \mathtt{RET}, e, v_1, v_2 :: [], [], [])) \star T$	$A :: r$

Figure 10.1.: Rules for the basic virtual machine

- `APPLY`(n) applies the accumulator to the top n values of the stack. Unlike the old `APP` opcode, it does not push a return address on the stack as this is done by `PUSHRETADDR`, nor does it move the arguments from the stack to the environment,
- `GRAB` moves the top value from the stack to the environment, possibly building a closure in the case of an under-applied function,
- `RET` now performs application in case of an over-applied function.

Besides, we have two additional opcodes `ACCU` and `ACCUCONST`, respectively for the application of free variables and of constants. The value of an inert term with a constant term becomes $(\texttt{ACCUCONST}, \mathbf{rev}(\pi), c, v)$ from (c, π, v) for compatibility with the form of values for abstractions. Closures still have a normal form which is computed one argument at a time, which is a compromise between simplicity of the machine and efficiency of computation of normal forms, which are a lot less frequent than application of n-ary functions to n arguments.

Once we adapt those opcodes from Grégoire and Leroy's work, the rest of the translation is relatively straightforward, and the full rules are presented in Figure 10.2.

	Threads	Active
	$(r \mapsto \mathtt{R}\ (\mathtt{PUSHRETADDR}(c_2); c_1, e, a, \pi, n, W)) \star T$	$r :: A$
$\rightsquigarrow_p$	$(r \mapsto \mathtt{R}\ (c_1, e, a, \langle c_2, n \rangle :: \pi, n, W)) \star T$	$A :: r$
	$(r \mapsto \mathtt{R}\ (\mathtt{LAZY}(c_2); c_1, e, a, \pi, n, W)) \star T$	$r :: A$
$\rightsquigarrow_a$	$(r \mapsto \mathtt{R}\ (c_1, e, a, r_f :: \pi, n, W)) \star (r_f \mapsto \mathtt{R}\ (c_2, e, \emptyset, [], 0, [])) \star T$	$A :: r$
	$(r \mapsto \mathtt{R}\ (\mathtt{VAR}(i); c, e, a, \pi, n, W)) \star T$	$r :: A$
$\rightsquigarrow_l$	$(r \mapsto \mathtt{R}\ (c, e, e(i), \pi, n, W)) \star T$	$A :: r$
	$(r \mapsto \mathtt{R}\ (\mathtt{ABS}(c_1); c_2, e, a, \pi, n, W)) \star T$	$r :: A$
$\rightsquigarrow_\lambda$	$(r \mapsto \mathtt{R}\ (c_2, e, (c_1, e, y, r_f), \pi, n, W)) \star$ $(r_f \mapsto \mathtt{R}\ (c_1, e, \emptyset, (\mathtt{ACCU}, [], y, \emptyset) :: [], 1, [])) \star T$	$A :: r$
	$(r \mapsto \mathtt{R}\ (\mathtt{RET}, e, v, \langle c, n \rangle :: \pi, 0, W)) \star T$	$r :: A$
$\rightsquigarrow_{r_1}$	$(r \mapsto \mathtt{R}\ (c, e, v, \pi, n, W)) \star T$	$A :: r$
	$(r \mapsto \mathtt{R}\ (\mathtt{RET}, e, (c_2, e_2, y, v_1), \pi, n_{>0}, W)) \star T$	$r :: A$
$\rightsquigarrow_{r_2}$	$(r \mapsto \mathtt{R}\ (c_2, e_2, (c_2, e_2, y, v_1), \pi, n_{>0}, W)) \star T$	$A :: r$
	$(r \mapsto \mathtt{R}\ (\mathtt{RET}, e, v, [], n, W)) \star T$	$r :: A$
$\rightsquigarrow_d$	$(r \mapsto \mathtt{D}\ v) \star T$	$A +\!\!+ W$
	$(r \mapsto \mathtt{R}\ (\mathtt{FORCE}; c, e, v_{\neg r}, \pi, n, W)) \star T$	$r :: A$
$\rightsquigarrow_v$	$(r \mapsto \mathtt{R}\ (c, e, v_{\neg r}, \pi, n, W)) \star T$	$A :: r$
	$(r \mapsto \mathtt{R}\ (\mathtt{FORCE}; c, e, r_2, \pi, n, W)) \star (r_2 \mapsto \mathtt{D}\ v) \star T$	$r :: A$
$\rightsquigarrow_s$	$(r \mapsto \mathtt{R}\ (c, e, v, \pi, n, W)) \star (r_2 \mapsto \mathtt{D}\ v) \star T$	$A :: r$

Figure 10.2.: Rules of the full virtual machine

	Threads	Active
	$(r \mapsto \texttt{R}\ (\texttt{FORCE}; c, e, r_2, \pi, n, W)) \star (r_2 \mapsto \texttt{R}\ (c, e_2, a_2, \pi_2, n_2, [])) \star T$	$r :: A$
$\rightsquigarrow_{f_1}$	$(r \mapsto \texttt{R}\ (\texttt{FORCE}; c, e, r_2, \pi, n, W)) \star (r_2 \mapsto \texttt{R}\ (c, e_2, a_2, \pi_2, n_2, r :: [])) \star T$	$A :: r_2$
	$(r \mapsto \texttt{R}\ (\texttt{FORCE}; c, e, r_2, \pi, n, W)) \star (r_2 \mapsto \texttt{R}\ (c, e_2, a_2, \pi_2, n_2, W_2)) \star T$	$r :: A$
$\rightsquigarrow_{f_2}$	$(r \mapsto \texttt{R}\ (\texttt{FORCE}; c, e, r_2, \pi, n, W)) \star (r_2 \mapsto \texttt{R}\ (c, e_2, a_2, \pi_2, n_2, r :: W_2)) \star T$	A
	$(r \mapsto \texttt{R}\ (\texttt{GRAB}; c, e, a, v :: \pi, n + 1, W)) \star T$	$r :: A$
$\rightsquigarrow_{g_1}$	$(r \mapsto \texttt{R}\ (c, v :: e, a, \pi, n, W)) \star T$	$A :: r$
	$(r \mapsto \texttt{R}\ (\texttt{GRAB}; c, e, a, \langle c_2, n \rangle :: \pi, 0, W)) \star T$	$r :: A$
$\rightsquigarrow_{g_2}$	$(r \mapsto \texttt{R}\ (c_2, e, (\texttt{GRAB}; c, e, y, r_f), \pi, n, W)) \star (r_f \mapsto \texttt{R}\ (c, e, \emptyset, [], 0, [])) \star T$	$A :: r$
	$(r \mapsto \texttt{R}\ (\texttt{GRAB}; c, e, a, [], 0, W)) \star T$	$r :: A$
$\rightsquigarrow_{g_3}$	$(r \mapsto \texttt{D}\ (c, e, y, r_f)) \star (r_f \mapsto \texttt{R}\ (c, (\texttt{ACCU}, [], y, \emptyset) :: e, \emptyset, [], 0, [])) \star T$	$A \mathbin{+\!\!+} W$
	$(r \mapsto \texttt{R}\ (\texttt{APPLY}(n_2); c_1, e, (c_2, e_2, y, v_1), \pi, n, W)) \star T$	$r :: A$
$\rightsquigarrow_{\beta}$	$(r \mapsto \texttt{R}\ (c_2, e_2, (c_2, e_2, y, v_1), \pi, n_2, W)) \star T$	$A :: r$
	$(r \mapsto \texttt{R}\ (\texttt{ACCU}, e, (c_2, e_2, x, \emptyset), \overline{v}^n \mathbin{+\!\!+} \langle c_1, n_2 \rangle :: \pi, n, W)) \star T$	$r :: A$
$\rightsquigarrow_{n_1}$	$(r \mapsto \texttt{R}\ (c_1, e, (\texttt{ACCU}, \overline{v}^n \mathbin{+\!\!+} e, x, \emptyset), \pi, n_2, W)) \star T$	$A :: r$
	$(r \mapsto \texttt{R}\ (\texttt{ACCUCONST}, e, (c_2, e_2, c, v_1), \overline{v}^n \mathbin{+\!\!+} \langle c_1, n_2 \rangle :: \pi, n, W)) \star T$	$r :: A$
$\rightsquigarrow_{n_2}$	$(r \mapsto \texttt{R}\ (c_1, e, (\texttt{ACCUCONST}, \overline{v}^n \mathbin{+\!\!+} e, c, r_f), \pi, n_2, W)) \star$ $(r_f \mapsto \texttt{R}\ (\texttt{APPLY}(n); \texttt{RET}, e, v_1, \overline{v}^n, 0, [])) \star T$	$A :: r$

Figure 10.2.: Rules of the full virtual machine (cont.)

11. Coq proof

11.1. Overview

We wrote a Coq proof that the parallel convertibility test we showed in the previous chapters is correct. The formalisation includes a set of threads performing reduction, as well as a tree structure describing the convertibility threads. As in the previous part (chapter 6), we include the extension with constructors and pattern matching, and we also include defined constants in our formalisation since they are necessary to have an interesting thread structure. However, we do not prove the part of the previous machines concerning the management of active threads, supposing that any thread might make a step at any moment. The definition of terms is the same as in chapter 6, as is the definition of the reduction relation.

11.2. States of the abstract machine

Reduction threads are specified by a map of identifiers (`rthreadptr`) to their description (`rthread`). The inductive types defining a thread is given below:

```
Inductive cont : Type :=
| Kid : cont
| Kapp : value -> cont -> cont
| Kswitch :
  list (nat * term) -> list value (* env *) ->
  list (list nat * value) -> cont -> cont

with value : Type :=
| Thread : rthreadptr -> value
| Neutral : (nat * cont * option value) (* neutral *) -> value
| Clos : term -> list value (* env *) -> nat -> value -> value
| Block : nat -> list value -> value.

Definition env := list value.
Definition neutral := (nat * cont * option value)%type.

Inductive code :=
| Term : term -> env -> code
| Val : value -> code.

Record rthread := mkrthread {
  rt_code : code ;
  rt_cont : cont ;
}.
```

The type `cont` corresponds to the stacks π in chapter 9: the reason why the type is more complex is that we handle pattern-matching as well, leading to the inclusion of the `Kswitch` constructor. We can indeed see that without this constructor, `cont` is simply isomorphic to a list of values, which was the type of the stack π. The type `value` is a direct translation of the type of values in the machine, with the added case `Block`. As we can see, in the definitions of `cont` and `value`, we had to manually expand the definitions of `neutral` and `env`, since those could not be defined at the same time.

The definition of the state of the reduction threads then simply needs the mapping of thread pointers to the corresponding threads, as well as a count for generation of fresh variable names:

```
Record state := mkstate {
  st_rthreads : list rthread ;
  st_freename : nat ;
}.
```

We can also define the convertibility threads. As we do not try to formalise which threads are active or not, we do not need a mapping of identifiers to threads, and we can simply formalise them by an inductive structure over which we will be doing rewrites:

```
Inductive cthread :=
| cthread_done : bool -> cthread
| cthread_reduce : value -> value -> list nat -> list nat -> cthread
| cthread_and : cthread -> cthread -> cthread
| cthread_or : cthread -> cthread -> cthread.
```

The different cases are:

- `cthread_done b`, which is a convertibility thread which has run to completion with `b` as the result of the convertibility test,
- `cthread_and` and `cthread_or`, which express the result of the convertibility test is given either by the conjunction or disjunction of the results of two other threads,
- `cthread_reduce v1 v2 varmap1 varmap2`, which corresponds to a still-running convertibility thread between the values `v1` and `v2`, where `varmap1` and `varmap2` specify how to interpret the free variables of both values, which correspond to a bound variable with a de Bruijn index corresponding to the position of the variable in the list.

The global state of the machine is then simply defined as `state * cthread`.

11.3. Definition of the reduction relation

With this definition, we can add the rules for evaluation. The rules for reduction threads are completely independent of the convertibility threads, and can be specified as a function of type `state -> rthreadptr -> state`. Thus, the rules for a given reduction thread are deterministic. When a thread cannot be reduced at that point, either because the thread is waiting for another or because it is stuck due to a type error, we simply return the unchanged state, a value which is enough to prove correctness (if the machine produces a result, then it is the correct result), but neither the absence of type errors (which might exist in the original program anyway, as we do not make any assumptions about it) nor the absence of deadlocks (which cannot happen due to the acyclic graph structure of reduction threads, although this is not proven).

For instance, the reduction rules for application, variables, as well as reading the output of a finished thread are given below:

```
Definition step_r (st : state) (rid : rthreadptr) : state :=
  match nth_error st.(st_rthreads) rid with
  | None => st
  | Some rthread =>
    match rthread.(rt_code) with
    | Term (app u v) e =>
      let st2v := makelazy st v e in
      update_rthread (fst st2v) rid {|
        rt_code := Term u e ; rt_cont := Kapp (snd st2v) rthread.(rt_cont)
      |}
    | Term (var n) e =>
      match nth_error e n with
      | None => st (* variable not found *)
      | Some v =>
        update_rthread st rid {|
          rt_code := Val v ; rt_cont := rthread.(rt_cont)
        |}
      end
    | Val (Thread rid2) =>
      match is_finished st rid2 with
      | None => st (* Thread is not finished yet, wait *)
      | Some v =>
        update_rthread st rid {|
          rt_code := Val v ; rt_cont := rthread.(rt_cont)
        |}
      end
    [...]
    end
  end
```

In this code, we can see that if `rid` does not reference an existing thread, then the state is returned unchanged as a dummy value (as this function will only ever be called on references to existing threads). We can also see the application case, where a new thread is created using the function `makelazy`, which returns a new state and a value corresponding to the newly created thread. While in the current version `makelazy` always creates a new thread, even in the variable case, as this simplifies the proofs a little, there is no technical difficulty in changing it to perform the variable optimisation as with the strong call-by-need machine. Recall, however, that while this optimisation gives better results in practice, it gives the same asymptotic complexity since this is call-by-need and not call-by-name.

In the variable case, the variable is given by its de Bruijn index, as usual. We extract its binding from the environment, and update the thread with the value read from the variable. If the variable was not found in the environment, we return `st` unchanged, as this corresponds to an incorrect variable reference, more precisely a non-closed input term.[1] Note that this time, `st` is not just a dummy value: while this case should never happen during the operation of the machine, the invariants necessary to enforce this were not proven. Thus, we need to return a state that will not affect the correctness proofs, and for this, returning `st` unchanged is the best way, as this corresponds to a no-op.

In the case of a thread value, a reduction step would correspond to reading the result of the thread once it has finished, thus introducing a dependency on this thread. The `is_finished` function returns `None` is the given thread has not finished, and `Some v` if it has finished with value `v` (meaning that its code is `Val v` and its continuation is `Kid`). Thus, we update the value if the thread it depends on has finished. However, as we want a total function here (and we are not trying to prove the absence of deadlocks), we simply return `st`, as this thread cannot perform a reduction step for the moment. Again, this is not just a dummy value: this can happen if we try to reduce a thread which is currently waiting on another.

If we wanted to specify (and prove) that deadlocks and errors cannot happen, we could modify the definition of `step_r` slightly, by returning a type with three different constructors: `Step st`, when we perform a reduction step with new state `st`, `Error`, when the reduction step would result in an error (such as accessing an undefined variable), and `Waiting rid`, when we are waiting on another thread with identified `rid`. Proving the absence of errors and deadlocks could then be specified as saying that in all states accessible from the initial state, we have no thread such that the result of `step_r` is `Error`, and that from each thread, we can follow the chains of threads waiting on another until we get to a thread which can step (this would be specified using an accessibility predicate).

However, proving the absence of errors would require proving a certain number of additional invariants, for instance the fact that all terms are closed in their environment,

[1] We require closed input terms in the development, but this is not a limitation: to test convertibility of two open terms, it suffices to add leading λ-abstractions to cover all bound variables of both terms. The resulting terms are convertible if, and only if, the original open terms were convertible.

or even the definition of forbidden patterns, and this work has thus not been done for now. Proving the absence of deadlocks is probably easier, likely not requiring additional invariants, but has not yet been done either.

With the definition above, we can give reduction threads their semantics, but we still need to give the semantics of convertibility threads. Luckily, they do not modify the state, so they can simply be given as a `state -> cthread -> cthread -> Prop` inductive relation. We show several cases of this relation below to illustrate how it works.

```
Inductive cthread_red (st : state) : cthread -> cthread -> Prop :=
| cthread_reduce_1 :
    forall rid v1 v2 varmap1 varmap2,
      is_finished st rid = Some v1 ->
      cthread_red st
        (cthread_reduce (Thread rid) v2 varmap1 varmap2)
        (cthread_reduce v1 v2 varmap1 varmap2)
| cthread_reduce_clos_clos :
    forall t1 t2 e1 e2 x1 x2 v1 v2 varmap1 varmap2,
      cthread_red st
        (cthread_reduce (Clos t1 e1 x1 v1) (Clos t2 e2 x2 v2) varmap1 varmap2)
        (cthread_reduce v1 v2 (x1 :: varmap1) (x2 :: varmap2))
| cthread_reduce_same_var_unfold :
    forall x c1 c2 uf1 uf2 varmap1 varmap2,
      cthread_red st
        (cthread_reduce
          (Neutral (x, c1, Some uf1))
          (Neutral (x, c2, Some uf2))
          varmap1 varmap2)
        (cthread_or
          (cthread_reduce uf1 uf2 varmap1 varmap2)
          (cmp_cont_cthread c1 c2 varmap1 varmap2))
| cthread_or_false :
    cthread_red st
      (cthread_or (cthread_done false) (cthread_done false))
      (cthread_done false)
| cthread_or_true_1 :
    forall ct, cthread_red st
      (cthread_or (cthread_done true) ct)
      (cthread_done true)
| cthread_or_true_2 :
    forall ct, cthread_red st
      (cthread_or ct (cthread_done true))
      (cthread_done true)
| cthread_or_1 :
    forall ct1 ct2 ct3, cthread_red st ct1 ct2 ->
```

```
      cthread_red st
        (cthread_or ct1 ct3)
        (cthread_or ct2 ct3)
[...]
.
```

Here, we can first see that if we have a convertibility thread referencing a reduction thread that has succeeded with value `v1`, then we can replace the thread reference to the computed value. Moreover, we can see an example of a convertibility thread between two values that have been evaluated to be closures: in this case, we record the name of the free variables inside the variable maps, and we say that the original terms were convertible if and only if the bodies of the functions are convertible. We can also see the case of two identical head constants, in which case we generate a `cthread_or` of further reductions and of the arguments.

The other kinds of reduction rules for convertibility threads are structural. First, if we have a conjunction or a disjunction of convertibility threads, we can simplify it if we have enough information to know the result, using short-circuiting evaluation when possible. Second, we allow reduction under the branches of convertibility threads so that they will eventually produce a result.

Finally, with both the rules for reduction and convertibility threads defined, we can define the rules for a single step of reduction, which is either a reduction or a convertibility step:

```
Inductive step : (cthread * state) -> (cthread * state) -> Prop :=
| step_rthread :
  forall ct st rid,
    rid < length st.(st_rthreads) ->
      step (ct, st) (ct, (step_r st rid))
| step_cthread :
  forall ct1 ct2 st,
    cthread_red st ct1 ct2 ->
    step (ct1, st) (ct2, st).
```

11.4. Proving the correctness of reduction

With the reduction rules defined, we need to formalise the readback function to prove the correctness. It is quite complex, with three mutually recursive inductive predicates specifying the rules for threads, values and continuations. Here we give the definition for the core λ-calculus.

```
Inductive read_thread st defs : list nat -> rthreadptr -> term -> Prop :=
| read_thread_val : forall varmap rid v c t h,
    nth_error st.(st_rthreads) rid =
      Some {| rt_code := Val v ; rt_cont := c |} ->
    read_val st defs varmap v t ->
    read_cont st defs varmap c h ->
    read_thread st defs varmap rid (fill_hctx h t)
| read_thread_term : forall varmap rid e el c t h,
    nth_error st.(st_rthreads) rid =
      Some {| rt_code := Term t e ; rt_cont := c |} ->
    closed_at t (length e) ->
    Forall2 (read_val st defs varmap) e el ->
    read_cont st defs varmap c h ->
    no_dvar t ->
    read_thread st defs varmap rid (fill_hctx h (subst (read_env el) t))

with read_val st defs : list nat -> value -> term -> Prop :=
| read_val_thread :
    forall varmap rid t,
      read_thread st defs varmap rid t ->
      read_val st defs varmap (Thread rid) t
| read_val_clos :
    forall varmap t e el x vdeep tdeep,
      Forall2 (read_val st defs varmap) e el ->
      read_val st defs (x :: varmap) vdeep tdeep ->
      convertible beta (subst (lift_subst (read_env el)) t) tdeep ->
      no_dvar t -> length defs <= x < st.(st_freename) ->
      x \notin varmap -> closed_at t (S (length e)) ->
      read_val st defs varmap (Clos t e x vdeep)
        (subst (read_env el) (abs t))
| read_val_neutral :
    forall varmap x c h uf tuf,
      read_cont st defs varmap c h ->
      if_Some3 (fun v2 t2 def =>
                  read_val st defs varmap v2 t2 /\
                  convertible beta (fill_hctx h def) t2 /\
                  closed_at def 0)
               uf tuf (nth_error defs x) ->
      x < st.(st_freename) ->
      (nth_error defs x = None -> In x varmap) ->
      read_val st defs varmap (Neutral (x, c, uf))
        (fill_hctx h (
          match index Nat.eq_dec varmap x with
          | None => dvar x
          | Some n => var n
          end))
[...]
```

```
with read_cont st defs : list nat -> cont -> hctx -> Prop :=
| read_cont_kid : forall varmap, read_cont st defs varmap Kid h_hole
| read_cont_kapp :
    forall varmap v c t h,
      read_val st defs varmap v t ->
      read_cont st defs varmap c h ->
      read_cont st defs varmap (Kapp v c)
        (compose_hctx h (h_app h_hole t))
[...]
.
```

The rules for the readback of a thread are quite straightforward: when it is a value, we simply read the value and the continuation, and put the value inside the context obtained as the readback of the continuation. When it is a term, we read its environment, perform the substitution, read the continuation, and the result is obtained by putting the substituted term inside the context of the environment. In that case, there are some additional hypotheses to enforce invariants on the terms, namely that they may no longer contain references to defined variables (which are expanded to normal variables with a definition at the very beginning), and that the term is closed in its environment.

The rules for reading continuations are very simple as well: `Kid` is directly read to the basic context `h_hole`, while for `Kapp`, we read the rest of the context and the value, and the result is obtained by extending the context appropriately.

The complex part is really about reading values. Here, we have three cases in the core λ-calculus: threads, closures, and neutral terms. For threads, we simply read the thread and get its associated value.

For closures, we need its environment to be read to a given substitution. Moreover, we check that the variable `x` defined by the closure is not part of the variable map associating de Bruijn indices to bound variables, and that this variable `x` corresponds to a name that will never be used again, and which was not a defined name at the beginning. We further need, and that is the crucial part, that the value corresponding to the reduced body of the λ-abstraction is β-convertible with the unreduced abstraction.[2]

For neutral values, most of the constraints are administrative, with the exception of the same condition of convertibility with the value corresponding to an unfolded definition.

As can be seen, these mutually-recursive predicates are very complex, and become even more so when adding the rules for constructors and pattern matching. Since we need to perform a large number of proofs by induction on these predicates, we developed a small library in Ltac2 for generating induction principles which are more general that Coq's,

[2] Note that we only prove convertibility and not single-direction reduction here: this is a lot simpler to prove, as very complex invariants are needed if we want to prove single-direction reduction to be stable with respect to reductions happening inside the environment `e`.

able to handle mutual induction as well as recursing under Forall, Forall2 and other such constructions. Obtaining a general induction principle becomes as easy as:

```
Definition read_ind st defs :=
  Induction For [
    read_thread st defs ;
    read_val st defs ;
    read_cont st defs
  ].
```

Refining those invariants was a lot of work, but given those invariants, the proof of preservation requires little imagination.

The outline of the proof is the following: first, we define a points_to relation between two thread identifiers, where a thread points to another if it includes a reference to the other in it. We prove that this relation is well-founded, which shows that our graph of threads is acyclic. Next, we prove that when reducing a thread rid, the readback of all threads indirectly pointed to by rid is unchanged, since their values did not change and they could not have depended on rid.

We then prove the following statement, that states that for the thread rid, the readback corresponds to either zero or one beta-reduction from the value it was read back to, as well as a few additional properties, using a large case analysis and exploiting the full power of our invariant.

```
Lemma step_r_beta_hnf :
  forall st st2 defs varmap rid t,
    defs_ok defs st ->
    Forall (fun x => x < st_freename st) varmap ->
    NoDup varmap ->
    step_r st rid = st2 ->
    read_thread st defs varmap rid t ->
    same_read_plus st st2 defs rid /\
    (forall rid2,
      nth_error (st_rthreads st) rid2 = None ->
      nth_error (st_rthreads st2) rid2 <> None ->
      exists t2 varmap2,
        varmap_ok (st_freename st2) varmap2 /\
        read_thread st2 defs varmap2 rid2 t2) /\
    exists t2,
      read_thread st2 defs varmap rid t2 /\
      reflc beta_hnf t t2.
```

Finally, we show that for threads which were indirectly pointing to rid, we have reductions of subterms for their readback, enabling us to prove the preservation theorem, whose statement is given below.

```
Lemma step_r_correct_val :
  forall st defs rid varmap v t,
    defs_ok defs st -> state_wf st defs ->
    read_val st defs varmap v t ->
    varmap_ok (st_freename st) varmap ->
    exists t2,
      read_val (step_r st rid) defs varmap v t2 /\
      star beta t t2.
```

Here, the various hypotheses are simple invariants expressing that there is no conflict between defined constants and neutral variables introduced when reducing under λ-abstractions, and also that every thread can meaningfully be read back to a term (thus bringing all other invariants with them). The conclusion says that if a value can be read back to a term, it can still be read back after reduction to a term that is a β-reduced version of the previous term.

11.5. Proving the correctness of convertibility

The other part of the readback is concerned with convertibility threads. For those, the readback relation is a lot simpler and given below:

```
Inductive read_cthread st defs : cthread -> bool -> Prop :=
| read_cthread_done :
  forall b, read_cthread st defs (cthread_done b) b
| read_cthread_reduce :
  forall v1 v2 varmap1 varmap2 b,
    (forall t1 t2,
       read_val st defs varmap1 v1 t1 ->
       read_val st defs varmap2 v2 t2 ->
       reflect (convertible (betaiota defs) t1 t2) b) ->
  read_cthread st defs (cthread_reduce v1 v2 varmap1 varmap2) b
| read_cthread_or_false :
    forall ct1 ct2,
      read_cthread st defs ct1 false ->
      read_cthread st defs ct2 false ->
      read_cthread st defs (cthread_or ct1 ct2) false
| read_cthread_or_true_1 :
    forall ct1 ct2,
      read_cthread st defs ct1 true ->
      read_cthread st defs (cthread_or ct1 ct2) true
| read_cthread_or_true_2 :
    forall ct1 ct2,
      read_cthread st defs ct2 true ->
      read_cthread st defs (cthread_or ct1 ct2) true
```

```
[...]
.
```

Here, the idea is that the readback of a convertibility thread is a Boolean, which is true whenever the inputs are convertible and false otherwise. In the case of `cthread_done`, the result is already computed and the readback is thus this Boolean, while in the `cthread_reduce` case, we say that the readback of the thread is obtained by reading the terms, and the readback is true if and only if the terms thus read are convertible. For the case of `cthread_or` and `cthread_and`, we cannot simply use Coq's Boolean connectors `bor` and `band`, as with the short-circuiting evaluation, one branch may not have a readback at all before it is removed by a short-circuiting evaluation. Thus, we specify them as three different rules, allowing `cthread_or` to be read back to `true` even if only one of the sides is read back to `true` and the other cannot be read back at all.

The preservation theorem for `cthread` readback is quite simple to state:

```
Lemma cthread_red_correct :
  forall st defs ct1 ct2 b,
    defs_wf defs ->
    cthread_wf st defs ct1 ->
    cthread_red st ct1 ct2 ->
    read_cthread st defs ct2 b ->
    read_cthread st defs ct1 b.
```

Here, `defs_wf` and `cthread_wf` express simple well-formedness definitions as before, and the theorem states that if after reduction, the `cthread` can be read back to a Boolean `b`, then it could be read back before as well. Thus, if after a number of reduction steps, the thread evaluates to `cthread_done b`, we obtain that the original thread can be read back to `b` as well, meaning that the input terms are convertible or not according to `b`.

The full statement of the final correctness theorem is thus given below:

```
Lemma all_correct :
  forall defs t1 t2 st b,
    defs_wf defs ->
    closed_at t1 0 -> closed_at t2 0 ->
    dvar_below (length defs) t1 -> dvar_below (length defs) t2 ->
    star step (init_conv defs t1 t2) (cthread_done b, st) ->
    reflect (convertible (betaiota defs) t1 t2) b.
```

It expresses that given two closed terms and well-formed definitions, where the instances of `dvar` inside the terms only reference existing definitions, if after some steps from the initial state created from `defs`, `t1` and `t2` we obtain a `cthread_done b` convertibility thread, then the convertibility of `t1` and `t2` is indeed expressed by `b`.

Note that, as we said earlier, this does not guarantee the absence of errors or deadlocks, and thus that a `cthread_done` state will be reached. However, it guarantees that once it is reached, then the result thus obtained is correct.

11.6. A library for generating induction principles in Coq

We wrote a small library that allows us to generate induction principles, even when Coq's own generated induction principles are not strong enough, for instance for the case of nested recursion. The Ltac tactic language of Coq does not allow access to enough information to generate them, but its successor Ltac2 does, which is why we used it for that library, even if the rest of the proof is written in Ltac.

Writing the library did not require any particular insights: we simply generate the induction hypotheses in a naïve manner from the contents of the different constructors of the inductive type. However, we apply well-chosen theorems when we encounter arguments such as `list t` or `Forall P l`. In these cases, we apply a previously-defined theorem to get a stronger induction hypothesis than Coq would be able to produce. One thing to note is that those theorems need to be closed with `Defined.` instead of `Qed.`, as the guardedness check must be able to look under those definitions to see that the induction hypothesis is indeed used in a positive position.

The core loop for generating the induction hypotheses corresponding to a given argument of a given constructor looks like the following:

```
Ltac2 rec constrind_hyp
    (v : constr) (inds : constr list) (hrecs : constr list) :=
  let t := Std.eval_hnf (Constr.type v) in
  match is_ind_prefix_l inds t hrecs with
  | Some argshrec =>
    let (args, hrec) := argshrec in mk_app (applist hrec args) v
  | None =>
    lazy_match! t with
    | list ?a =>
      mk_Forall_proof_smart (Fresh.in_goal @x) a
        (fun x => constrind_hyp x inds hrecs) v
    | @Forall ?t ?p ?l =>
      mk_Forall_impl_smart t p l (fun h => constrind_hyp h inds hrecs) v
    | ?a /\ ?b =>
      mk_and_proof_smart
        (constrind_hyp (mk_app 'proj1_transparent v) inds hrecs)
        (constrind_hyp (mk_app 'proj2_transparent v) inds hrecs)
    [...]
    | _ => 'I
    end
  end.
```

As we can see here, the syntax of Ltac2 has a lot of differences with that of Ltac, and can be very close to a conventional ML-like programming language. In the above code, we evaluate the type of the argument to its head-normal form, and then continue depending on what we get:

- If the type of the argument was the type of the inductive type we are currently generating an induction principle for (or one of the types defined mutually with it), we use the induction hypothesis.
- If the type is of some select list of known types (such as `list`, the `Forall` type or conjunction), we use smart constructors for the theorems mentioned above to get an induction hypothesis as strong as possible.
- Otherwise, we do not know what we can do with this argument and simply generate the type `True` for the hypothesis.

As we can seen, the above pattern-matching can be easily extended when adding a new type such as `Forall2`, allowing us to generate induction principles very efficiently. Although we generate very few such induction principles in the actual development shown before and the library could be replaced by tedious hand-written induction principles, its help has been invaluable when writing the proof and we needed to modify the definitions, allowing to experiment a lot more with what precise definitions we needed for the proof to work.

12. Complexity bounds for convertibility

In this chapter, we study the question of providing complexity bounds for our machine. However, due to the undecidable nature of untyped convertibility, the length of the shortest proof can grow faster than any computable function. Even in a very restricted typed setting, the simply typed λ-calculus, convertibility is already known to be TOWER-hard [Sta79], enough to erase even an exponential speedup. Therefore, we would like to provide a complexity bound depending on the size of a shortest proof of convertibility. However, defining what a shortest proof means is tricky: do we mean a shortest mathematical proof in some ambient theory such as ZFC? Do we mean, as in [Con20], the number of steps to reduce both terms to normal forms and compare the results?

Neither of these notions feels appropriate. Mathematical proofs are significantly too powerful, and it would be almost impossible to get any relation between the complexity of our convertibility check and the length of a shortest mathematical proof. On the other hand, our machine can prove convertibility *faster* than reducing both terms to normal forms, and it does not feel right to say that our machine can end up computing a proof in time that is sublinear of the length of the proof.

For these reasons, we argue that the pertinent notion here is the length of a shortest proof of a form corresponding to the proofs that can be produced by our machine. In this case, it means reduction with sharing of reduction steps, and convertibility with the shortcutting rules when stacks are compared (in the case of non-convertibility), or when applying the same constant to convertible stacks (in the case of convertibility).

12.1. Reduction structures

To give a proper definition of the length of a proof, we must first specify how reduction steps are to be counted. We will do that through the notion of *reduction structure*. The intuition behind reduction structures is that implementations of λ-calculus reduction use some form of graph to represent sharing in an implementation of λ-calculus reduction, but how the sharing is precisely implemented does not matter for our proof of complexity. Thus, we abstract the details of reduction, and consider a set of possible graphs, which we will call states, and a set of vertices for each of these graphs, which we will call handles.

A reduction structure is a $5-$tuple $(R, \rightarrow, H, \mathbf{read}, \mathbf{head})$, with the following properties:

- R is a set, called the set of *states* of the structure;
- $\rightarrow$ is a relation between elements of R, called the *reduction relation* of the structure;

- H is a function from states to sets of *handles*, such that for each r, r', if $r \to r'$, then $H_r \subseteq H'_r$;
- **read**, called the *readback function* of the structure, is a function taking a state r and a handle $h \in H_r$ and returning a λ-term;
- **head**, the *head-taking function* of the structure, is a function taking a state r and a handle $h \in H_r$, and returning one of:
  - $\perp$;
  - $\lambda x.h'$, in which case $\mathbf{read}(r, h) = \lambda x.\mathbf{read}(r, h')$;
  - $x\ \overline{h'}$, in which case $\mathbf{read}(r, h) = x\ \overline{\mathbf{read}(r, h')}$;
  - $(c\ \overline{h'}, h'')$, in which case $\mathbf{read}(r, h) = \overline{c\ \mathbf{read}(r, h')}$, c is a defined constant expanding to d, and $\mathbf{read}(r, h'') \equiv_\beta d\ \overline{\mathbf{read}(r, h')}$;

We also require the following properties:

- *compatibility of read with β*: for all states r, r' and handles $h \in H_r$, if $r \to r'$, then $\mathbf{read}(r, h) \equiv_\beta \mathbf{read}(r', h)$.[1]
- *validity of head*: for each r and $h \in H_r$, for each h' appearing in $\mathbf{head}(r, h)$, we have $h' \in H_r$;
- $\to$ is strongly confluent;
- *progression of head*: for all r, r' and $h \in H_r$, if $r \to r'$ and $\mathbf{head}(r, h) \neq \perp$, then $\mathbf{head}(r', h) = \mathbf{head}(r, h)$;
- *determinism of head*: for all r, r', r'' and $h \in H_r$ such that $r \to^* r'$ and $r \to^* r''$, if $\mathbf{head}(r', h) \neq \perp$ and $\mathbf{head}(r'', h) \neq \perp$, then $\mathbf{head}(r', h) = \mathbf{head}(r'', h)$.

To give some intuition, handles correspond quite precisely to what were called `values` in the Coq proof of chapter 11, and states correspond to the type `state`. Almost all of the properties shown here are proved in Coq, except determinism, which is also true only up to renaming of thread identifiers and free variables in the Coq version.

We will say that handle h is *computed* in state r if $\mathbf{head}(r, h) \neq \perp$.

With the definition of reduction structure in mind, we can give a formal definition of the length of a proof. For that, we assume a given reduction structure $(R, \to, H, \mathbf{read}, \mathbf{head})$, and we will consider a state r, as well as two handles $h_1, h_2 \in H_r$ for which we want to prove equality or inequality.

A proof of convertibility between $\mathbf{read}(r, h_1)$ and $\mathbf{read}(r, h_2)$ then consists in two kinds of steps: reduction steps, where we replace r with r' such that $r \to r'$, and convertibility steps, where we deduce a (non-)convertibility relation from already known such relations.

[1] We see here that we used $\equiv_\beta$ instead of $\to_\beta^*$. Although we should think of it as $\to_\beta^*$, $\equiv_\beta$ is enough for the properties we want and far easier to prove.

$$\textsc{Red}\quad \frac{r \to r'}{(r,\mathcal{E}) \rightsquigarrow_r (r',\mathcal{E})} \qquad \textsc{Unfold1}\quad \frac{\mathbf{head}(r,h_1) = (c\ \overline{h'}, h'') \qquad (h'' \overset{?}{\equiv} h_2) \in \mathcal{E}}{(r,\mathcal{E}) \rightsquigarrow_c (r, \mathcal{E} \cup \{h_1 \overset{?}{\equiv} h_2\})}$$

$$\textsc{Unfold2}\quad \frac{\mathbf{head}(r,h_2) = (c\ \overline{h'}, h'') \qquad (h_1 \overset{?}{\equiv} h'') \in \mathcal{E}}{(r,\mathcal{E}) \rightsquigarrow_c (r, \mathcal{E} \cup \{h_1 \overset{?}{\equiv} h_2\})} \qquad \textsc{Head}\quad \frac{\mathcal{E} \vdash \mathbf{head}(r,h_1) \overset{?}{\equiv} \mathbf{head}(r,h_2)}{(r,\mathcal{E}) \rightsquigarrow_c (r, \mathcal{E} \cup \{h_1 \overset{?}{\equiv} h_2\})}$$

$$\textsc{Abs}\quad \frac{(h_1 \overset{?}{\equiv} h_2) \in \mathcal{E}}{\mathcal{E} \vdash \lambda x.h_1 \overset{?}{\equiv} \lambda x.h_2} \qquad \textsc{VarEq}\quad \frac{\forall i, (h_i \equiv h'_i) \in \mathcal{E}}{\mathcal{E} \vdash x\ \overline{h}^n \equiv x\ \overline{h'}^n} \qquad \textsc{VarNEq1}\quad \frac{(h_i \not\equiv h'_i) \in \mathcal{E}}{\mathcal{E} \vdash x\ \overline{h}^n \not\equiv x\ \overline{h'}^n} \qquad \textsc{VarNEq2}\quad \frac{n \neq m}{\mathcal{E} \vdash x\ \overline{h}^n \not\equiv x\ \overline{h'}^m}$$

$$\textsc{VarNEq3}\quad \frac{x \neq y}{\mathcal{E} \vdash x\ \overline{h} \not\equiv y\ \overline{h'}} \qquad \textsc{VarAbs}\quad \frac{}{\mathcal{E} \vdash x\ \overline{h} \not\equiv \lambda y.h'} \qquad \textsc{AbsVar}\quad \frac{}{\mathcal{E} \vdash \lambda x.h \not\equiv y\ \overline{h'}} \qquad \textsc{ConstEq}\quad \frac{\forall i, (h_i \equiv h'_i) \in \mathcal{E}}{\mathcal{E} \vdash (c\ \overline{h}^n, h'') \equiv (c\ \overline{h'}^n, h''')}$$

Figure 12.1.: Convertibility rules for non-thread values. In the rules, $\overset{?}{\equiv}$ means either $\equiv$ or $\not\equiv$.

We start with an initial state $(r, \emptyset)$ where we are in state r and we have proved no relations, and we perform the steps shown in Figure 12.1. The proof of convertibility (resp. non-convertibility) is complete when we have $(h_1 \equiv h_2) \in \mathcal{E}$ (resp. $(h_1 \not\equiv h_2) \in \mathcal{E}$).

The first four rules specify how we should compare handles: Red says that we can always perform a reduction step in the structure, while rules Unfold1 and Unfold2 express the fact that we can unfold constants on either side. Finally rules Head states that if both the heads of the handles have been computed, we only need to compare the heads.

The next eight rules specify how heads should be compared: the Var rules specify how terms where both heads are variables should be compared, while ConstEq expresses the fact that if both terms are the same constant applied to convertible terms, they are convertible. The Abs looks simple, but this is because we have conveniently ignored the question of the renaming of the variable bound by the λ-abstraction, to simplify the proof.[2]

[2] We could however formally justify this with an additional *well-bindedness* hypothesis on our reduction structure: formally, each handle h would carry a list L of potentially-bound variables, so that $\mathbf{fv}(\mathbf{read}(r,h)) \subseteq L$, and whenever we have h, h' with associated lists L, L' such that $\mathbf{head}(r,h)$ contains h', we either have $\mathbf{head}(r,h) = \lambda x.h'$ and $L' = x :: L$, or $L = L'$. Then, the Abs rule could be extended to add a mapping between the variables of the lists associated between h_1 and h_2, and the VarEq and VarNEq3 rules would check that mapping.

$$\begin{tikzcd}
r \arrow[r] \arrow[d] & \mathbf{step}(r, h) \arrow[d, dashed, "\leq 1"] \\
r' \arrow[r] & \mathbf{step}(r', h)
\end{tikzcd}$$

Figure 12.2.: Strong diamond property for **step**

$$\begin{tikzcd}
r \arrow[r] \arrow[d] & \cdots \arrow[r] \arrow[d] & r'' \arrow[d] & & \\
\mathbf{step}(r, h) \arrow[r, dashed, "\leq 1"'] & \cdots \arrow[r, dashed, "\leq 1"'] & \mathbf{step}(r'', h) \arrow[r] & \cdots \arrow[r] & r'
\end{tikzcd}$$

Figure 12.3.: The strong diamond property implies shortest paths

12.2. Effective reduction structures

While reduction structures are expressive enough to define the notion of the length of a proof, they lack features for effective computation, since the reduction relation $\to$ is only a relation and is thus not constructive: given r, it is unclear how to compute r' such that $r \to r'$.

In order to extend them enough to allow us to specify our algorithm in terms of conversion structures, we introduce *effective reduction structures*, which are reduction structures extended with another operation, **step**, that given a state r and a handle $h \in H_r$ such that $\mathbf{head}(r, h) = \bot$, returns a state such that $r \to \mathbf{step}(r, h)$, and the step $r \to \mathbf{step}(r, h)$ is necessary to have on any path from r to a state r' where $\mathbf{head}(r', h) \neq \bot$. Formally, we will only ask the fact that if we also have $r \to r'$ for some $r' \neq \mathbf{step}(r, h)$, then $\mathbf{head}(r', h) = \bot$ and $\mathbf{step}(r, h) \to^{\leq 1} \mathbf{step}(r', h)$. This property is a form of strong diamond lemma, as depicted in Figure 12.2.

These two conditions together are very strong, implying that if we have a reduction sequence $r \to^* r'$, such that $\mathbf{head}(r', h) \neq \bot$ but $\mathbf{head}(r, h) = \bot$, then there exists a reduction sequence $r \to \mathbf{step}(r, h) \to^* r'$ that is at most as long. To see that, we simply consider the first step of this sequence of the form $r'' \to \mathbf{step}(r'', h)$, which necessarily exists since $\mathbf{head}(r', h) \neq \bot$. Then, as shown in Figure 12.3, we can use the second condition to see that the path $r \to \mathbf{step}(r, h) \to^* \mathbf{step}(r'', h)$ is at most as long as the path $r \to^* r'' \to \mathbf{step}(r'', h)$, concluding this proof.

Furthermore, the same proof applies to any set of handles for which we want to compute all their heads: if we have a set X of handles, such that their heads are not all already computed, any first step of the form $\mathbf{step}(r, h)$ where $h \in X$ and $\mathbf{head}(r, h) = \bot$ is on a shortest path from r to a state where all the handles of X have been computed. Besides, thanks to strong confluence, beginning with any step not of this form will either be on a shortest path, or not count at all, but it cannot make the length of the shortest path from the new state bigger.

Intuitively, the **step** function defines a critical path of reduction to get to a state where a given handle is computed. In our Coq or OCaml development, **step**, for a given handle, corresponds to making a single step in the thread the handle is currently waiting for, possibly transitively, while for λ-calculus, it would correspond to head reduction – but then we need parallel reduction to be able to enforce the strong diamond property.

We require two additional properties of our reduction structure to be able to properly define our algorithm and guarantee its complexity:

- *acyclicity of handles*: if we define $h' \ll_r h$ whenever h' appears in $\mathbf{head}(r, h)$, then the transitive closure of $\ll_r$ is a strict pre-order;
- *limited growth of handles*: for each r, H_r is finite; furthermore, there exists a constant c such that for all r, r', if $r \to r'$, then $|H'_r| \leq |H_r| + c$.

With this, we can see our machine from chapter 9 fits this framework as follows:

- We have a set of reduction threads, each associated with a non-computed handle, and at most one reduction thread per handle. A reduction thread associated to h can perform a reduction step by replacing r with $\mathbf{step}(r, h)$. Besides, if there exists a convertibility thread between h_1 and h_2, there exists a reduction thread associated to at least one of h_1 and h_2 if they are not both already computed.
- We have a set of convertibility threads, each between a pair of two handles, and no two threads corresponding to the same pair. These threads can perform a convertibility step if both of their handles have been computed, and will possibly create other convertibility threads using the different logic connectors at our disposal so that they cover all possible cases shown in Figure 12.1, as we have shown before.
- We schedule all those threads in a round-robin manner, and stop as soon as we have found a proof of convertibility or non-convertibility of our initial problem.
- The constant c corresponds to the maximum of 1 and the maximal arity of a constructor in the initial terms.

Notice how, in this definition, we did not talk about active threads at all: this is because we consider an inactive thread like it did not exist at all. In fact, the only rules that we have are that there must be threads for parts that are needed for a potential proof; but we have no constraint on having only necessary threads running, as we simply bound the number of threads globally using the number of handles.

Let us now prove a complexity bound on our machine depending on the size of any given proof of (non-)convertibility between h_1 and h_2 in our model. Suppose this proof uses n_r reduction steps, and n_c convertibility steps, for a total of n steps.

If we could choose optimally at each step which thread is executed, our machine could conclude in at most n steps as well. For this, we would just to execute the convertibility steps in the same order as in the proof; and for the reduction steps, we notice that if we need the head of a handle h, we necessarily have a reduction thread allowing us to use a reduction step $r \to \mathbf{step}(r, h)$, which exists on a shortest path of reduction to compute the heads of all the handles we need, as noted before.

Without choosing which thread is executed at each step, we can still ensure that our machine concludes after n rounds of executing the threads has been done, since spurious reduction steps cannot increase the number of reduction steps needed other than themselves, and spurious convertibility steps have no incidence on the rest other than increasing the number of threads.

Thus, we only need to bound the number of threads to get a complexity bound. This is easily done: since performing a single step of reduction can only create up to c handles, if we consider all the reductions happening when we execute all the threads in a single round, if there were n_h handles before, there can be at most $(c+1)n_h$ handles afterwards. As a consequence, when we have made k rounds, there can be at most $O((c+1)^k)$ handles and thus reduction threads, and at most $O((c+1)^{2k})$ convertibility threads. Since executing each reduction or convertibility step takes time $O(1)$, and we will find a proof after at most n rounds, the total time taken by the machine is at most $O((c+1)^{2n})$.

13. Experimental evaluation

13.1. Methodology

For experimental evaluation, we relied on our OCaml implementation of the convertibility checker, which corresponds to the version verified in Coq in chapter 11, extended with fixpoints and the handling of active threads. Variables in the input terms were represented by the type `string`, which comes with additional costs compared to Coq's internal de Bruijn indices. On the Coq side, we instrumented Coq's convertibility checker so that it prints the time taken (this is much more precise than just relying on Coq's `Time` command, which also accounts for other aspects such as typechecking). We used Coq 8.15.2, extended with these changes to the convertibility checker. Moreover, both Coq and our implementation were compiled using OCaml 4.12.1, and the experiments were made on a laptop with an 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz CPU and 2x 16GiB SODIMM DDR4 Synchronous 3200 MHz (0.3 ns) RAM, running Linux 5.15.74 with NixOS 22.05.

We also compared an extended version of our convertibility checker as described in chapter 12, which performs sharing of convertibility tests and tries unfolding both sides independently in a convertibility test, and therefore has a guaranteed complexity bound.

13.2. Testcases

The different testcases we used are described and commented below, while the precise amount of time taken is described in Figure 13.1. On all the testcases, the timings remain quite small: we often hit stack overflows with larger inputs, and we can already spot the exponential behaviours with these inputs.

To begin, we consider the expansion of definitions:

```
Fixpoint exp2 n :=
  match n with
  | O => 1
  | S n => 2 * exp2 n
  end.

Definition zero (n : nat) := 0.
```

Testcase	Coq	Ours	Ours2
`test1`	3e-5	5e-5	6e-3
`test1c`	1e-5	5e-5	6e-5
`test2`	0.14	5e-6	2e-5
`test3`	9e-5	2e-4	5e-5
`test4`	0.018	0.013	9e-5

Testcase	Coq	Ours	Ours2
`test5`	4e-6	6e-6	8e-6
`test6`	0.61	1e-6	8e-6
`test7`	3e-5	7e-5	2e-4
`test8`	0.078	5e-5	2e-4
`test9`	2e-5	6e-5/0.18*	0.15

Figure 13.1.: Timings, in seconds, for the examples of convertibility problems given in the text for both Coq and both our OCaml implementations. See the main text for the explanation on the two results given for `test9` with our tool.

```
(* fast *)
Definition test1 :
  exp2 15 = exp2 (14 + 1)
:= eq_refl.
(* slow *)
Definition test2 :
  zero (exp2 15) = zero (exp2 16)
:= eq_refl.
```

Here, our testcases are `test1` and `test2`, whose definitions generate a convertibility test between the two sides of the equality. When using Coq, the definition of `test1` is fast, while the definition of `test2` is slow. Indeed, in both cases, Coq tries to prove the convertibility of the arguments before unfolding the definition. While this is a good move for `test1` as it allows it to prove convertibility without expanding `exp2`, in the case of `test2`, it tries and fails to prove the convertibility of `exp2 15` and `exp2 16`, costing a lot of work, while expanding `zero` would have proved convertibility immediately. We also studied a version of `test1` using Church integers instead, shown in the results as `test1c`.

Here, our own convertibility checker gets the best of both worlds by doing the work in parallel, and both the convertibility checks for `test1` and `test2` are fast. Our guaranteed-complexity checker does a bit worse on `test1` than on `test1c`, which we experimentally measured at $O(n^{2.8})$ complexity compared to $O(n)$ when using n instead of the constant 15 in the example. This is caused by the large amount of unfolding opportunities in the branch where we unfold `exp2`, causing an explosion in the number of threads. Solving this issue remains future work, discussed in Section 14.2.1.

Next, we consider what happens with terms that have a size exponential in the size of their memory representation, because there is a lot of sharing inside the term itself.

```
Inductive tree :=
| L : tree
| N : tree -> tree -> tree.
```

```
Fixpoint explode_share n t :=
  match n with
  | O => t
  | S n => explode_share n (N t t)
  end.

Fixpoint left_depth t :=
  match t with
  | L => 0
  | N t1 t2 => S (left_depth t1)
  end.

Fixpoint left_depth2 t :=
  match t with
  | L => 0
  | N t1 t2 => left_depth2 t1 + 1
  end.

(* fast *)
Definition test3 :
  left_depth (explode_share 15 L) = left_depth2 (explode_share 15 L)
:= eq_refl.

(* slow *)
Definition test4 :
  explode_share 15 L = explode_share 14 (N L L)
:= eq_refl.
```

Here, `explode_share` takes an argument `n` and a tree `t` and generates a tree with 2^n copies of `t`. However, this only takes time linear in `n` to evaluate, as these instances are shared. The definitions `left_depth` and `left_depth2` both compute the length of the leftmost branch of `t`, in linear time of the result for `left_depth`, and quadratic time for `left_depth2`.

When defining `test3`, the convertibility test is fast both in Coq and with our convertibility checker because terms are shared, so the computation of both sides takes only linear time. However, Coq is slow when defining `test4`, because after expanding `explode_share`, it has to prove the convertibility of the exact same terms multiple times. For the same reason, only the extended version of our convertibility check, which does sharing of convertibility tests, is fast.

Another interesting test is about the order in which the arguments of constructors (or identical defined constants) are compared. For this, we consider two very similar tests, given below.

```
(* fast *)
Fail Definition test5 :
  (exp2 15, false) = (exp2 16, true)
:= eq_refl.

(* slow *)
Fail Definition test6 :
  (false, exp2 15) = (true, exp2 16)
:= eq_refl.
```

Here, with Coq, the first test of non-convertibility is almost instantaneous: Coq starts by comparing `true` and `false`, since Coq evaluates such arguments right-to-left. They are different, so the test stops immediately. However, the second test is a lot slower: indeed, Coq starts by comparing `exp2 15` and `exp2 16`, which fails, but after a long time.

With our convertibility checker, both tests are as fast as one another: we test the convertibility of the arguments in parallel, so we immediately detect that `false` and `true` are not convertible, and return this result.

Another particularity of Coq is that once a constant is unfolded, it stays unfolded for future tests, preventing us from benefiting from the optimisation with folded constant. In the following tests, we can see the problem that this poses:

```
Definition is_zero n :=
  match n with
  | O => true
  | S n => false
  end.

(* fast *)
Definition test7pair n := (is_zero n, n).
Definition test7 :
  test7pair (exp2 15) = (false, exp2 15)
:= eq_refl.

(* slow *)
Definition test8pair n := (n, is_zero n).
Definition test8 :
  test8pair (exp2 15) = (exp2 15, false)
:= eq_refl.
```

Again, both tests are the same except for the order of arguments. When defining `test7`, Coq first starts by comparing `exp2 15` and `exp2 15`, which is almost instantaneous thanks to the folded constant optimisation. Then, it compares `is_zero (exp2 15)` with `false`, which takes only linear time, thanks to Coq's laziness.

However, when defining `test8`, Coq first starts by comparing `is_zero (exp2 15)` with `false`, forcing it to unfold `exp2` to prove the convertibility. Once this is done, it has to compare `exp2 15` with a version of `exp2 15` which is already partially computed and where `exp2` has been unfolded. At this point, it has no way but to expand `exp2` on the other side, and the time taken is exponential.

With our convertibility checker, both tests are fast. Indeed, when we unfold a constant, we also keep the original folded value, allowing us to still benefit from the folded constant optimisation if we encounter it again.

Of course, this comparison wouldn't be honest if we didn't show the shortcomings of our own convertibility checker as well. Consider the following example:

```
Definition f0 (n : nat) := n.
Definition f1 n := f0 (f0 n).
Definition f2 n := f1 (f1 n).
Definition f3 n := f2 (f2 n).
Definition f4 n := f3 (f3 n).
Definition test9 :
  f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (
  f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (
  f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 0
  )))))))))))))))))))))))))))))
= f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (
  f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (
  f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 (f4 0
  )))))))))))))))))))))))))))))
:= eq_refl.
```

On this example, Coq is almost instantaneous by applying the folded constant optimisation repeatedly, but since we explore both what happens when unfolding and when not unfolding, our convertibility checker is a lot slower. However, this depends heavily on the unfolding order of constants: if we choose to always unfold the older constant first when there are two different head constants, we obtain a result quite fast, as when we unfold `f4` on one side, then we will repeatedly unfold successively `f3`, `f2`, `f1` and `f0` on that side until this side has only `f4`, preventing the folded constant optimisation from applying and spawning new threads until that point. This will in turn severely limit the number of total convertibility threads that are created and thus allowing the code to run quite fast, albeit still slower than Coq. On the other hand, if we always unfold the *newer* constant first (which is often the best choice in Coq), when we unfold `f4` on one side, we will match this by unfolding `f4` on the other side next, making `f3` appear as the head constant on both sides, making the folded constant optimisation apply again, and so on with `f2`, `f1` and `f0`, creating in total a very large number of convertibility threads, and thus making the code run very slowly. Likewise, the extended version of our convertibility check, which does not use heuristics but always unfolds both sides in parallel is slow on this example.

However, such examples seem to be quite pathological, and we think they should not happen in practice. Besides, we strongly suspect that, once we implement sharing of equality proofs, we will have a guaranteed complexity of our convertibility test in terms of the shortest existing convertibility proof, which looks like a desirable property that Coq does not have.

14. Future and related work

14.1. Related work

14.1.1. The complexity of convertibility

In [Con20], Condoluci studies the complexity of convertibility in the pure λ-calculus. In it, she proves that if it takes n steps to reduce t_1 and t_2 to normal forms, then the complexity of determining whether t_1 and t_2 are convertible is $O((|t_1| + |t_2|)n)$, by reducing both terms to normal form (with sharing) using a well-chosen abstract machine, followed by a linear test to check equality of the normal forms in this representation.[1] Here, n is the number of steps needed to compute the normal form using strong call-by-value, but strong call-by-need (for instance using the strategy we outlined in the first part of this thesis) would work just as well and yield the same bounds.[2] This is a very good complexity, perhaps even the optimal one if we only allow ourselves to depend on those two quantities. Because it is restricted to the pure λ-calculus, it cannot exploit the folded constant optimisation, which is crucial to get good performance in Coq.

14.1.2. Verification of the Coq kernel

The MetaCoq project [Soz+20] contains a formal verification of the Coq kernel, including its convertibility checker. It is an impressive Coq development, which even includes the formalisation of cofixpoints and universes (with cumulativity) which we have not studied at all. What is formalised is the strategy we showed at the beginning of chapter 9, including termination, but only the soundness has been proved: if the strategy specified by the proof says that t_1 and t_2 are convertible, then they are indeed convertible, but if it answers they are not convertible, there is no proof they are indeed not convertible. Our own development does not include a proof of termination, nor does it handle the more complex features of Coq. However, we proved both soundness and completeness, and the strategy that is verified is more efficient in numerous cases, as we saw in chapter 13.

[1] This second step is in itself not trivial: a naïve test using an union-find data structure would incur an additional $O(\alpha((|t_1| + |t_2|)n))$ cost.

[2] Actually, n would likely be lower, but the constant factor higher, as is frequent with call-by-need compared to call-by-value.

14.2. Future work

14.2.1. Improved scheduling of threads

While our complexity proof shows a bound of $O(r^{n_r+n_c})$ running time (for some constant $r > 1$) to prove the convertibility when the shortest convertibility proof has n_r reduction steps and n_c convertibility steps, we would like to improve this bound to something linear in n_r. With the version of the machine we described, it is unfortunately not the case, since convertibility steps on the non-branching case can exponentially increase the number of threads, diluting the useful work. We expect we could improve this by giving a time budget to each active thread, so that the sum of all these time budgets is 1, and dividing the actual time allocated to each thread to be proportional to the time budget of the given thread. In that case, we believe we could get a complexity bound of $O((n_r + n_{c_1})r^{n_{c_2}})$, where n_{c_1} is the number of non-branching convertibility steps (the steps in the shortest proof where only once convertibility rule can be applied with the given heads), while n_{c_2} is the number of branching convertibility steps. This would make our convertibility check competitive with Coq in the `test1` case, instead of having a cubic slowdown. However, we have not proved or implemented any part of this yet.

14.2.2. Proving the absence of deadlocks and errors as well as termination

Our Coq formalisation does not try to prove the absence of deadlocks and errors in the machine, simply using dummy transitions to the same state, making the term effectively stuck. Proving their absence remains future work, and while it seems like the invariants needed should not be too complicated, there still is a risk as long as the proof is not written that we have missed an invariant. If an invariant is missing, it would in this case most likely be that we need full β-reduction instead of simply β-equivalence between a λ-abstraction and its reduced form.

Proving termination (assuming termination of the input term) is another complex problem, and this one would very probably need to prove β-equivalence between a λ-abstraction and its reduced form. While we are very confident that this holds, the proof would likely be extremely complex: a previous version for strong call-by-need small-step reduction semantics took about the same proof size and effort as the either our big-step semantics for strong call-by-need in the previous part, or the convertibility test of this part, while only handling the base lambda-calculus (without constructors, matching or defined constants), and being very complex to scale: the invariant depended on several readback functions, with the same reference to a lazy value being sometimes expanded to the definition of the lazy value and sometimes not, even in the same term! Thus, proving β-reduction between a λ-abstraction and its reduced form in our convertibility test remains a substantial amount of work, either practical by adapting the existing proof, or theoretical by finding a simpler proof, which we have not begun yet.

14.2.3. Encoding as concurrent programming

We have the intuition that our semantics might be amenable to encoding in a concurrent programming language, whose thread/process abstraction would be used instead of our home-made threads and scheduling. Since we took care of enforcing the subterm property in our machine, it might even be possible to compile our input problem into a program for that language, which we can then compile and run to get the answer to the problem we were considering. We could also write an interpreter to avoid the initial cost of compilation, but it is likely that for small problems, our own implementation would be at an advantage given its simplicity, and for large problems, the initial cost of compilation would be small compared to the cost of running the program. We would then be able to benefit from the capabilities of the host language for running on multiple cores or even computing nodes, as a true parallel proof search.

The host language would however need to satisfy several criteria rarely seen together to be able to encode our machine. Indeed, we need process creation to be an extremely lightweight operation, since a new thread is created for each β-reduction. Call-by-need is already known to be less efficient than call-by-value, but process creation is always significantly more expensive, even if the new process starts as inactive: the smallest size of an Erlang process, themselves extremely small, is already in the order of 300 bytes, far larger than a simple thunk. We would also need other processes to pause and resume a running process, a capability seldom found in concurrent languages.

In it unclear whether it would be worth pursuing this further, however. In practice, convertibility problems are quite small, requiring either only small amounts of computation, or large amounts of reduction but where the actual terms being compared are quite small (the classic example being proofs by reflection where the only amount of convertibility needed is between `true` and a term that evaluates to `true` after a lot of computation). Besides, interactive theorem provers are supposed to be interactive, in a relatively tight feedback loop between the user and the tool, meaning that problems large enough so to be worth the cost of distributed computing are unlikely to arise in practice.

14.2.4. Producing and replaying proof traces

Our parallel machine was an answer to the initial remark that we would need an oracle to know whether we should unfold the definitions or compare the arguments when encountering a problem of the form $f\ t_1 \overset{?}{\equiv} f\ t_2$. However, once we have proved convertibility or non-convertibility with our machine, we know exactly what we need to do to find again the proof we initially found. Thus, we could extend our convertibility checker to record such traces, using them afterwards in a sequential checker that would verify the proof a lot faster in $O(r)$ time, while at the same time, due to its sequential nature, being a lot easier to formally verify and trust, and probably with a smaller constant factor as well.

Since interactive theorem provers such as Coq spend a lot of time replaying proofs (either when running an independent checker on the proofs, or when a user performs an unrelated change in the code and the proof needs to be verified again), it would likely be useful to be able to extract the proof trace and use it as an oracle to consult on the various decisions about unfolding (or about which argument to compare in cases of non-convertibility), allowing us to consistently outperform Coq.

Conclusions

In this thesis, we studied a simple and efficient strong call-by-need semantics, as well as a convertibility checker that uses parallelism to search efficiently for a proof of convertibility or non-convertibility. With both the strong call-by-need semantics and the convertibility checker being verified in Coq, we can put trust in them in spite of their complexity, and perhaps, in the more distant future, integrate them into the Coq code base.

Such a goal would require extending at least the theory (and, preferably for trust, the proofs) with all other features that Coq supports and that we do not: cofixpoints, universes and cumulativity, as well as newer features such as `SProp`.

A perhaps more realistic short-term goal would be to turn our checker into an untrusted proof producer, producing a trace that would then be used by Coq to check convertibility; it is likely this would require very limited changes to the current convertibility checker used in Coq.

Independently from these practical matters, our work yields the first convertibility checker that we know of that has worst-case complexity bounds depending only on the size of the smallest proof, with realistic proof rules faster than just reducing to normal form.

While work remains to produce a formal proof of complexity or termination, we believe that specifying, formalising and proving the machine was the largest and most interesting part of the work. With it done, we have a framework to use for the remaining points, which we think will only require small adaptations of the existing proof.

We hope our work will lead others to think about worst-case complexity of the convertibility checkers in use within their tools instead of relying on heuristics that can go wrong and leaving the user with bad experiences where seemingly simple problems take a very long time.

We also hope it will help shift the vision of convertibility from a simple computation problem to something closer to a proof search, opening up new possibilities for proof rules and search strategies able to use them efficiently, perhaps exploiting injectivity of some functions or analysing which arguments of functions are required.

Bibliography

[ACC21] Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. "Strong Call-by-Value is Reasonable, Implosively". In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021.* IEEE, 2021, pp. 1–14. DOI: 10.1109/LICS52264.2021.9470630. URL: https://doi.org/10.1109/LICS52264.2021.9470630.

[ACM04] Andrea Asperti, Paolo Coppola, and Simone Martini. "(Optimal) duplication is not elementary recursive". In: *Inf. Comput.* 193.1 (2004), pp. 21–56. DOI: 10.1016/J.IC.2004.05.001. URL: https://doi.org/10.1016/j.ic.2004.05.001.

[AGN96] Andrea Asperti, Cecilia Giovanetti, and Andrea Naletto. "The Bologna Optimal Higher-Order Machine". In: *J. Funct. Program.* 6.6 (1996), pp. 763–810. DOI: 10.1017/S0956796800001994. URL: https://doi.org/10.1017/S0956796800001994.

[Ana+17] Abhishek Anand et al. "CertiCoq: A verified compiler for Coq". In: *The Third International Workshop on Coq for Programming Languages.* 2017. URL: https://popl17.sigplan.org/details/main/9/CertiCoq-A-verified-compiler-for-Coq.

[Ari+95] Zena M. Ariola et al. "The Call-by-Need Lambda Calculus". In: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995.* Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 233–246. DOI: 10.1145/199448.199507. URL: https://doi.org/10.1145/199448.199507.

[Asp17] Andrea Asperti. *About the efficient reduction of lambda terms.* 2017. arXiv: 1701.04240 [cs.LO].

[Bal+17] Thibaut Balabonski et al. "Foundations of strong call by need". In: *Proc. ACM Program. Lang.* 1.ICFP (2017), 20:1–20:29. DOI: 10.1145/3110264. URL: https://doi.org/10.1145/3110264.

[Bal12] Thibaut Balabonski. "A unified approach to fully lazy sharing". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012.* Ed. by John Field and Michael Hicks. ACM, 2012, pp. 469–480. DOI: 10.1145/2103656.2103713. URL: https://doi.org/10.1145/2103656.2103713.

[Bar67] J. Barkley Rosser. "Haskell B. Curry and Robert Feys. Combinatory logic. Volume I. With two sections by William Craig. Studies in logic and the foundations of mathematics. North-Holland Publishing Company, Amsterdam1958, xvi + 417 pp." In: *Journal of Symbolic Logic* 32.2 (1967), pp. 267–268. DOI: 10.1017/S0022481200114203.

[BCD22] Malgorzata Biernacka, Witold Charatonik, and Tomasz Drab. "A simple and efficient implementation of strong call by need by an abstract machine". In: *Proc. ACM Program. Lang.* 6.ICFP (2022), pp. 109–136. DOI: 10.1145/3549822. URL: https://doi.org/10.1145/3549822.

[BDG11] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. "Full reduction at full throttle". In: *Certified Programs and Proofs*. Certified Programs and Proofs. Kenting, Taiwan: Springer, Dec. 2011. DOI: 10.1007/978-3-642-25379-9_26. URL: https://inria.hal.science/hal-00650940.

[BLM05] Tomasz Blanc, Jean-Jacques Lévy, and Luc Maranget. "Sharing in the Weak Lambda-Calculus". In: *Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday*. Ed. by Aart Middeldorp et al. Vol. 3838. Lecture Notes in Computer Science. Springer, 2005, pp. 70–87. DOI: 10.1007/11601548_7. URL: https://doi.org/10.1007/11601548%5C_7.

[BLM21] Thibaut Balabonski, Antoine Lanco, and Guillaume Melquiond. "A Strong Call-By-Need Calculus". In: *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*. Ed. by Naoki Kobayashi. Vol. 195. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 9:1–9:22. DOI: 10.4230/LIPIcs.FSCD.2021.9. URL: https://doi.org/10.4230/LIPIcs.FSCD.2021.9.

[BWP17] Christoph Benzmüller, L. Weber, and Bruno Woltzenlogel Paleo. "Computer-Assisted Analysis of the Anderson-Hájek Ontological Controversy". In: *Logica Universalis* 11.1 (2017), pp. 139–151. DOI: 10.1007/S11787-017-0160-9. URL: https://doi.org/10.1007/s11787-017-0160-9.

[CH88] Thierry Coquand and Gérard Huet. "The calculus of constructions". In: *Information and Computation* 76.2 (1988), pp. 95–120. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(88)90005-3. URL: https://www.sciencedirect.com/science/article/pii/0890540188900053.

[Cha12] Arthur Charguéraud. "The Locally Nameless Representation". In: *J. Autom. Reason.* 49.3 (2012), pp. 363–408. DOI: 10.1007/S10817-011-9225-2. URL: https://doi.org/10.1007/s10817-011-9225-2.

[Cha13] Arthur Charguéraud. "Pretty-Big-Step Semantics". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 41–60. DOI: 10.1007/978-3-642-37036-6_3. URL: https://doi.org/10.1007/978-3-642-37036-6%5C_3.

[Chu32] Alonzo Church. "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics* 33.2 (1932), pp. 346–366. ISSN: 0003486X. URL: `http://www.jstor.org/stable/1968337` (visited on 01/26/2024).

[Chu33] Alonzo Church. "A Set of Postulates For the Foundation of Logic". In: *Annals of Mathematics* 34.4 (1933), pp. 839–864. ISSN: 0003486X. URL: `http://www.jstor.org/stable/1968702` (visited on 05/10/2024).

[Chu36] Alonzo Church. "An unsolvable problem of elementary number theory." In: *Journal of Symbolic Logic* 58.2 (1936), pp. 345–363. DOI: `10.2307/2268571`.

[Cio13] Ştefan Ciobâcă. "From Small-Step Semantics to Big-Step Semantics, Automatically". In: *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings.* Ed. by Einar Broch Johnsen and Luigia Petre. Vol. 7940. Lecture Notes in Computer Science. Springer, 2013, pp. 347–361. DOI: `10.1007/978-3-642-38613-8\_24`. URL: `https://doi.org/10.1007/978-3-642-38613-8%5C_24`.

[Con20] Andrea Condoluci. "Beta-Conversion, Efficiently". PhD thesis. alma, Apr. 2020. URL: `http://amsdottorato.unibo.it/9444/`.

[Cré07] Pierre Crégut. "Strongly reducing variants of the Krivine abstract machine". In: *High. Order Symb. Comput.* 20.3 (2007), pp. 209–230. DOI: `10.1007/S10990-007-9015-Z`. URL: `https://doi.org/10.1007/s10990-007-9015-z`.

[Dar09] Zaynah Dargaye. "Vérification formelle d'un compilateur optimisant pour langages fonctionnels. (Formal verification of an optimizing compiler for functional languages)". PhD thesis. Paris Diderot University, France, 2009. URL: `https://tel.archives-ouvertes.fr/tel-00452440`.

[GAL92] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. "The Geometry of Optimal Lambda Reduction". In: *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992.* Ed. by Ravi Sethi. ACM Press, 1992, pp. 15–26. DOI: `10.1145/143165.143172`. URL: `https://doi.org/10.1145/143165.143172`.

[GL02] Benjamin Grégoire and Xavier Leroy. "A compiled implementation of strong reduction". In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.* Ed. by Mitchell Wand and Simon L. Peyton Jones. ACM, 2002, pp. 235–246. DOI: `10.1145/581478.581501`. URL: `https://doi.org/10.1145/581478.581501`.

[Gon+13] Georges Gonthier et al. "A Machine-Checked Proof of the Odd Order Theorem". In: *Interactive Theorem Proving.* Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 163–179. ISBN: 978-3-642-39634-2.

[Gon08] Georges Gonthier. "Formal Proof—The Four- Color Theorem". In: 2008. URL: `https://api.semanticscholar.org/CorpusID:12620754`.

[GS17] Stefano Guerrini and Marco Solieri. "Is the Optimal Implementation Inefficient? Elementarily Not". In: *2nd International Conference on Formal Struc-*

tures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK. Ed. by Dale Miller. Vol. 84. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 17:1–17:16. DOI: `10.4230/LIPICS.FSCD.2017.17`. URL: `https://doi.org/10.4230/LIPIcs.FSCD.2017.17`.

[How80] William Alvin Howard. "The Formulae-as-Types Notion of Construction". In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by Haskell Curry et al. Academic Press, 1980.

[Hur+13] Chung-Kil Hur et al. "The power of parameterization in coinductive proof". In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 193–206. DOI: `10.1145/2429069.2429093`. URL: `https://doi.org/10.1145/2429069.2429093`.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Jon92] Simon L. Peyton Jones. "Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine". In: *J. Funct. Program.* 2.2 (1992), pp. 127–202. DOI: `10.1017/S0956796800000319`. URL: `https://doi.org/10.1017/S0956796800000319`.

[Kan+23] Hrutvik Kanabar et al. "PureCake: A Verified Compiler for a Lazy Functional Language". In: *Proc. ACM Program. Lang.* 7.PLDI (2023), pp. 952–976. DOI: `10.1145/3591259`. URL: `https://doi.org/10.1145/3591259`.

[Kri07] Jean-Louis Krivine. "A call-by-name lambda-calculus machine". In: *High. Order Symb. Comput.* 20.3 (2007), pp. 199–207. DOI: `10.1007/S10990-007-9018-9`. URL: `https://doi.org/10.1007/s10990-007-9018-9`.

[KSF18] Fabian Kunze, Gert Smolka, and Yannick Forster. "Formal Small-Step Verification of a Call-by-Value Lambda Calculus Machine". In: *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*. Ed. by Sukyoung Ryu. Vol. 11275. Lecture Notes in Computer Science. Springer, 2018, pp. 264–283. DOI: `10.1007/978-3-030-02768-1\_15`. URL: `https://doi.org/10.1007/978-3-030-02768-1%5C_15`.

[Kum+14] Ramana Kumar et al. "CakeML: a verified implementation of ML". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 179–192. DOI: `10.1145/2535838.2535841`. URL: `https://doi.org/10.1145/2535838.2535841`.

[Lam90] John Lamping. "An Algorithm for Optimal Lambda Calculus Reduction". In: *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*. Ed. by Frances E. Allen. ACM Press, 1990, pp. 16–30. DOI: `10.1145/96709.96711`. URL: `https://doi.org/10.1145/96709.96711`.

[Lau93] John Launchbury. "A Natural Semantics for Lazy Evaluation". In: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*. Ed. by Mary S. Van Deusen and Bernard Lang. ACM Press, 1993, pp. 144–154. DOI: `10.1145/158511.158618`. URL: `https://doi.org/10.1145/158511.158618`.

[Ler09] Xavier Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (2009), pp. 107–115. URL: `http://xavierleroy.org/publi/compcert-CACM.pdf`.

[LM96] Julia L. Lawall and Harry G. Mairson. "Optimality and Inefficiency: What Isn't a Cost Model of the Lambda Calculus?" In: *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*. Ed. by Robert Harper and Richard L. Wexelblat. ACM, 1996, pp. 92–101. DOI: `10.1145/232627.232639`. URL: `https://doi.org/10.1145/232627.232639`.

[MCT10] Andrew McCreight, Tim Chevalier, and Andrew P. Tolmach. "A certified framework for compiling and executing garbage-collected languages". In: *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. Ed. by Paul Hudak and Stephanie Weirich. ACM, 2010, pp. 273–284. DOI: `10.1145/1863543.1863584`. URL: `https://doi.org/10.1145/1863543.1863584`.

[Poi94] H. Poincaré. "Sur la nature du raisonnement mathématique". In: *Revue de Métaphysique et de Morale* 2.4 (1894), pp. 371–384. ISSN: 00351571, 21025177. URL: `http://www.jstor.org/stable/40891545` (visited on 10/24/2023).

[Soz+20] Matthieu Sozeau et al. "The MetaCoq Project". In: *J. Autom. Reason.* 64.5 (2020), pp. 947–999. DOI: `10.1007/S10817-019-09540-0`. URL: `https://doi.org/10.1007/s10817-019-09540-0`.

[SS18] George Stelle and Darko Stefanovic. "Verifiably Lazy: Verified Compilation of Call-by-Need". In: *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages, IFL 2018, Lowell, MA, USA, September 5-7, 2018*. Ed. by Matteo Cimini and Jay McCarthy. ACM, 2018, pp. 49–58. DOI: `10.1145/3310232.3310236`. URL: `https://doi.org/10.1145/3310232.3310236`.

[Sta79] Richard Statman. "The Typed lambda-Calculus is not Elementary Recursive". In: *Theor. Comput. Sci.* 9 (1979), pp. 73–81. DOI: `10.1016/0304-3975(79)90007-0`. URL: `https://doi.org/10.1016/0304-3975(79)90007-0`.

[SW10] Olin Shivers and Mitchell Wand. "Bottom-up beta-reduction: Uplinks and lambda-DAGs". In: *Fundam. Informaticae* 103.1-4 (2010), pp. 247–287. DOI: `10.3233/FI-2010-328`. URL: `https://doi.org/10.3233/FI-2010-328`.

[Swi12] Wouter Swierstra. "From Mathematics to Abstract Machine: A formal derivation of an executable Krivine machine". In: *Proceedings Fourth Work-*

shop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012. Ed. by James Chapman and Paul Blain Levy. Vol. 76. EPTCS. 2012, pp. 163–177. DOI: `10.4204/EPTCS.76.10`. URL: `https://doi.org/10.4204/EPTCS.76.10`.

[The24] The Coq Development Team. *The Coq Reference Manual – Release 8.19.0*. `https://coq.inria.fr/doc/V8.19.0/refman`. 2024.

[Tho84] Ken Thompson. “Reflections on Trusting Trust”. In: *Commun. ACM* 27.8 (1984), pp. 761–763. DOI: `10.1145/358198.358210`. URL: `https://doi.org/10.1145/358198.358210`.

[Tur37] A. M. Turing. “Computability and λ-definability”. In: *Journal of Symbolic Logic* 2.4 (1937), pp. 153–163. DOI: `10.2307/2268280`.

[Wad71] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda-calculus*. University of Oxford, 1971. URL: `https://books.google.fr/books?id=kl1QIQAACAAJ`.