Alma Mater Studiorum - Università di Bologna

Dottorato di Ricerca in

Computer Science and Engineering

XXXII Ciclo

β-Conversion, Efficiently

Presentata da . . . . . Andrea Condoluci

Supervisore . . . . . Prof. Claudio Sacerdoti Coen

Coordinatore Dottorato . . . . . Prof. Davide Sangiorgi

Settore Concorsuale 01/B1-Informatica

Settore Scientifico Disciplinare INF-01-Informatica

Esame finale anno 2020

Preface

Type-checking in dependent type theories relies on conversion, *i.e.* testing given λ-terms for equality up to β-evaluation and α-renaming.

Computer tools based on the λ-calculus currently implement conversion by means of algorithms whose complexity has not been identified, and in some cases even subject to an exponential time overhead with respect to the natural cost models (number of evaluation steps and size of input λ-terms).

This dissertation shows that in the pure λ-calculus it is possible to obtain conversion algorithms with bilinear time complexity when evaluation is carried following evaluation strategies that generalize Call-by-Value to the stronger case required by conversion.

Acknowledgements

There are several people that I would like to thank for their support. I had the luck of having two supervisors, one official and the other unofficial, but equally important: I wish this experience to any student. Both my supervisors are not only great scientists, but also possess their unique human quality that enriched me a lot.

First of all, many thanks to my supervisor, Claudio Sacerdoti Coen. He not only taught how to program and how to be a researcher, but also how to run after an idea, by always pushing me to improve, and by always investing all his energy and passion.

Many thanks also to my other supervisor, Beniamino Accattoli. He thaught me how to widen my sight and see the bigger picture, and initiated me to the utopian but worthwhile struggle to understand and be understood by others.

I am really grateful to Herman Geuvers and Xavier Leroy for reviewing this dissertation, for their time, and for the useful advice and suggestions.

Thanks to Giulio Guerrieri, for teaching me that geographical distance is a concept that simply does not exist in the life of a researcher, by always popping up randomly in Bologna unexpected but welcome.

Thanks to my office mates at University of Bologna: Gabriele for setting an example with his strong work ethics, Melissa for her taste in fashion, Paolo for the shortest tutorial on proof nets, and Ivan for the shared lunches.

Thanks to the Parsifal team and to Dale Miller for having me at INRIA Saclay, and for all the fruitful discussions: thank you Gabriel, Kaustuv, Maico, Marianela, Roberto, Ulysse, the

two Matteos, in particular Manighetti for his long friendship.

Last but not least, my family: I am foremost grateful to my parents Pina and Francesco and my brother Giulio, whom I wish a magnificent scientific career. But also to the unofficial family: Cristian and Maria for always being there for me, Pietro for financing my vices, and Tommaso for always being thoughtful and caring. A final thank you to the unfortunate flatmates that had to bear me during the writing of this thesis: Lele, Marco, Paolo, Gazza, Corra, Fra, Leo.

Andrea Condoluci, February 2020

Contents

Chapter 1

Introduction

1.1 Perspective

About 5000 years ago *homo sapiens* invented mathematics, a set of notations and instructions to gain mental control over concepts like number and form. Fast-forward to today: every scientific discipline uses mathematical tools to justify its conclusions. But how are mathematical theories themselves justified? One answer lies in logicism (19th century) which advocates to ground mathematics in logic, the discipline that studies the abstract laws of a form of reasoning called deductive. Logicism is quite appealing: in fact, the alliance between deductive reasoning and mathematics dates back to the ancient Greece, and has been very successful ever since. Take the *Elements* of Euclid, considered the most influential textbook of all times. The *Elements* cover elementary geometry and number theory using a format that is still used in mathematics today: the organization of the contents in definitions, axioms, theorems and proofs.

Consistency first! More than two thousands years after the *Elements*, Alfred North Whitehead and Bertrand Russell attempted to continue the program started by Euclid, by showing that all mathematics of the time could also be reduced to some collection of axioms. Their book *Principia Mathematica* was prompted by the discovery that set theory—the theory of sets of real numbers motivated by the field of analysis—gave rise to contradictions due to the naïve and informal way it was formulated back then. The most famous such contradiction is known as Russel's paradox, and intuitively rules out the existence of a "set of all sets": naïve set theory was thought to allow the definition of such a universal set, making it inconsistent, *i.e.* logically unusable. The contradiction arises directly from the principles used to form sets (called comprehension axioms), which are too permissive. In order to make set theory consistent again, one needs to include some limitation on the instances of comprehension that can be used.

What's in a type? In response to Russel's paradox, philosophers proposed new candidate theories to serve as foundation for mathematics. Some were based on Russel's own *type*

theory, a class of formal systems that derive typing judgements about terms: such judgements have the form $t : T$, which reads "the term t has type T". These typing judgements are derived according to axioms and inference rules in such a way that undesired, paradoxical objects cannot be assigned any type. Types fundamentally provide a grammatical restriction over terms: the term for the paradoxical Russel set is ill-typed in such type systems, and hence it does not denote any usable object.

The most successful among type theories is the *simply typed* λ-calculus, and was introduced by Church in 1940 as a general theory of functions. The λ-calculus is as expressive as it is basic: one can only apply functions to arguments, and computation — *i.e.* function invocation — is modeled by a single rule called β. It was only a few decades later, in 1969, that William Alvin Howard formalized the analogy between the simply typed λ-calculus and a proof system for intuitionistic logic known as natural deduction. This analogy is known as the *Curry-Howard correspondence*, and directly relates type theory and proof theory, *i.e.* computer programs and mathematical proofs.

Alexa, check my proof. With the personal computer revolution, the idea that mathematical proofs could be verified by a computer began to spread: among the first so-called proof assistants was *Automath*, initiated by Nicolaas Govert de Bruijn in 1967. Automath was based on type theory and was in a way a precursor of the Curry-Howard correspondence: in this approach, one maps each mathematical statement φ to a type T_φ in a suitable type theory, such that φ holds if and only if there exists a term t of type T_φ, *i.e.* such that $t : T_\varphi$. The role of a proof assistant is to facilitate a human user in writing down the term t, and then to ensure that t has type T_φ. The operation of checking that $t : T_\varphi$ is called type-checking, and when observed through the Curry-Howard mirror it amounts to proof-checking, i.e. verifying that a given term is a valid proof for a given statement.

The ambition of Automath was to handle all of mathematics, but to do so the theory of simple types mentioned above was clearly inadequate: in fact the type theory of Automath is much stronger and consists of a typed λ-calculus with dependent types. Dependent types are a feature of type systems that blurs the distiction between terms and types: terms may occur in type expressions, *i.e.* types depend on terms. For instance, we may have the type $Prime(3)$ — where the term 3 occurs — corresponding to the mathematical predicate that states that the natural number 3 is a prime number. By Curry-Howard, in order to prove that 3 is prime it suffices to provide a proof term t such that $t : Prime(3)$.

Like Automath, modern proof assistants such as *Agda*, *Coq*, *NuPRL* and *Matita* reduce the problem of proof-checking to the problem of type-checking in a programming language with dependent types. We are a long way from a logic utopia where all mathematical proofs are computer-checked, even though a lot of effort has been made towards computer-verified proofs up to a point where human cognition today cannot do without computer aids anymore.

OK Compute. The point of no return was crossed in 2005 with the formalization of the *four-color theorem* in *Coq* by Gonthier, 2007. The four-color theorem informally states that "four colors are enough to color a map", and as harmless as it may seem, it took many failed at-

tempts and more than one hundred years to be proved on paper. But even after its proof the solution did not fully convince all mathematicians, because it combined textual arguments with computer code, that could not be checked easily by human inspection. The issue was then settled with a more elegant revision of the original proof, and by Gonthier's formalization in a proof assistant. A further goal of that formalization was to explore the limits of formal proof systems and investigate new techniques: a particularly useful one was so-called "computational reflection", which relies heavily on a feature of dependent type theories called conversion. In fact, dependent type systems usually include an inference rule like the following:

Conversion rule

$$\frac{t\colon T' \qquad T' =_\beta T}{t\colon T}\ (\text{conv})$$

This is called conversion rule, and reads as follows: in order to prove that a term t has type T, it suffices to show that t has type T', if T and T' are *β-convertible*. β-convertible means that after performing some β computation, T and T' become the same type. Basically, the conversion rule lets computation happen transparently, without leaving any trace in the proof term t.

Every formalized proof requires computation in some way. For instance, suppose we obtained $t\colon Prime(1+2)$: in order to conclude that 3 is a prime number, the term "$1+2$" needs first to compute to "3", so that $Prime(1+2) =_\beta Prime(3)$. In this example we used the sum $+$ of natural numbers because it is a really simple recursive function, but in dependent type systems one can usually encode recursive functions of arbitrary complexity. Let us get back to the formalization of the four-color theorem: by relying heavily on conversion, one can reduce proofs with millions of cases to proof terms that are smaller but require huge computations. Note that these computations, omitted from the final proof term, need to be executed again whenever proof-checking.

As a consequence, proof-checking depends critically on conversion, and in order to implement proof-checking in an efficient way, one also needs to perform β-conversion efficiently. Investigating how to implement β-conversion efficiently is the motivation behind this dissertation, which we will detail in the next section.

1.2 Motivation

As mentioned above, the guiding light of this dissertation is how to make β-conversion in proof assistants more efficient. Conceptually, conversion is made up of two components:

$$\text{Conversion} = \text{Computation} + \text{Comparison}$$

In order to check whether two terms (or types) t and s are convertible, one has to perform both some computation, which in the λ-calculus amounts to β-steps, and also comparison,

which amounts to checking whether corresponding subparts of given terms are the same. (We remain vague on purpose, as we will go into more details later.)

In order to be efficient, both Computation and Comparison must be efficient: this dissertation explores the two parts separately, with computational complexity in mind.

Computation

In the λ-calculus, Computation boils down to evaluation (as in evaluating an expression or a program to its value). Like many proof assistants, also functional programming languages (FPL) are based on the λ-calculus. While a lot of effort has been put into devising efficient interpreters and compilers for FPL, not all of these findings can be transferred verbatim to proof assistants: the reason for this is that the kind of evaluation needed by proof assistants is of a more general kind than the one required by FPL. Comparison, in fact, requires to walk recursively over the structure of λ-terms, forcing one to evaluate and compare also the bodies of unapplied functions. We call this kind of evaluation "strong", *versus* the "weak" kind of evaluation required by FPL (that evaluate the instantiated body of a function only when enough arguments are provided).

To study and improve the efficiency of implementation, one got to get their hands dirty and get away from the tidiness of the formal definition of the λ-calculus:

- *Sharing:* during evaluation, if previously evaluated terms are not recorded, one is forced to perform repeated, unnecessary work that leads to huge inefficiency. The solution is using a mechanism to share subterms, in such a way that multiple instances of previously encountered subterms count as a single, re-usable entity. A concrete, but still idealized way to study shared evaluation — also by the point of view of complexity — is by means of *abstract machines*, the devices that we employ in this dissertation.

- *Open evaluation:* abstract machines for weak evaluation have existed since the 1960's, but strong machines are rare in the literature, because strong evaluation is much more difficult than the weak one. Recently, the research on abstract machines has been focusing on an intermediate setting between weak and strong evaluation, called *open* or *symbolic* evaluation, which is also the setting that we consider in this dissertation. One can perform strong evaluation "by levels", first weakly evaluating the term at top-level, and then iterating weak evaluation in the bodies of functions. The difference between weak and open evaluation is the following: when evaluating the body of a function, the variable that is the formal parameter of that enclosing function behaves as "free" (not being bound to any actual value) hence preventing subexpressions that contain that variable to evaluate to usual values. Weak evaluation in presence of free variables is called "open evaluation", and as we will see it presents challenges of its own.

Comparison

If we visualize λ-terms as *abstract syntax trees*, comparing two λ-terms means traversing their syntax trees and at the same time checking that nodes in corresponding positions represent

the same syntactical construct. Based on this description, the problem of Comparison looks trivial, but there are at least two additional aspects to consider:

- *Sharing.* As discussed above, we need sharing to perform Computation in an efficient way: therefore, after evaluation, λ-terms are not plain λ-terms but λ-terms with sharing. From the point of view of syntax trees, adding sharing to λ-term roughly corresponds to turning trees into directed acyclic graphs (DAGs), or λ-graphs. Note that traversing recursively a DAG may take exponential time in the size of the DAG, and similarly unfolding the sharing to obtain back a λ-tree is an exponential-time operation. In order to show that λ-graphs are just *succint* representations of λ-terms that do not introduce exponential-time blowups, we need to traverse the terms to be compared in a *smart* way.

- *Variable bindings.* Like in usual programming languages, λ-terms that only differ by the renaming of bound variables must be considered equal: this notion of equality is called α-equality. As a consequence, λ-graphs are not simply *term graphs*, representing first-order terms, but contain a notion of scoping that must be taken into account. For instance, when comparing two functions, one proceeds to traverse their bodies but only after recording in some way that their corresponding formal arguments are identified, even if they have different names. Variable scopes also interact with sharing, making efficient Comparison even more complex.

Conversion

As already mentioned, in this dissertation we will keep the problems of Computation and Comparison unweaved, studying them separately. However, this separation does not actually occur in the implementation of proof assistants: to compare two λ-terms, it is not necessary to fully evaluate them first, and later compare their values. In real implementations Computation and Comparison are interleaved, so that Conversion can fail earlier and be more performant in practice. In fact, failing early is essential for the performance of proof assistants: for instance, it is common for *solvers* to attack a proof goal by repeatedly applying various *tactics*, each requiring the proof assistant to perform convertibility checks and to backtrack in case of failure.

Still, I believe a formal study of these separate issues is fundamental: proof assistants are complex pieces of software, and their users must be able to trust their correctness. Since Conversion is together with proof-checking the central operation performed by the *kernel* of a proof assistant, it is important that all the underlying aspects are well-understood, and I believe this dissertation works towards this goal. I thus leave interleaving Computation and Comparison for future research, even though I will sketch my recommendations in the final chapter.

1.3 Methodology

Asymptotic complexity. Our focus is on the *asymptotic complexity* of the problems and algorithms that we present, to which we provide upper bounds via the *big O notation*. We strive for *linear-time* complexity: in the case of Computation, linear both in the size of the λ-term to be evaluated and in the number of β-steps according to a fixed evaluation strategy (Call-by-Value); in the case of Comparison, linear in the size of the shared terms to be compared. Note that the current state-of-the-art implementations have either unknown complexity, or do not possess the desired complexity.

We do not explore the average-case complexity of our solutions: as already mentioned, in order to improve the average running time of Conversion it is fundamental to interleave Computation and Comparison, so to fail earlier. We leave this to future research (see section 15.2). We also do not to measure the running time of our algorithms on concrete instances by means of *benchmarks*, mainly because there do not exist standard libraries or test suites for Conversion so to compare different approaches.

Full proofs. We provide full and detailed proofs for the contributions presented in the main technical parts. We decided to not include in the body of this dissertations the results for which we do not have fully spelled out proofs: notably, our evaluation machine for Strong Call-by-Value, which we outline in the part about future works, in section 15.1.

1.4 Outline

We now explain briefly the contents of each part of this dissertation.

Part I: Preliminaries

In the first part of this dissertation we provide the background for the topics covered in the following parts: this part contains mainly already existing concepts and results, not new contributions of this dissertation — even though I sometimes took the liberty of deviating a bit from how things are usually presented 😇.

As self-contained as I wish it were, this part is still quite synthetic and in no way exhaustive. However, I tried to lay it out in such a way as to provide all the intuitions and notions that are necessary to understand the following results. To do so, in some parts I provide more than the strictly necessary information, in order to help understand the bigger picture or show related concepts.

- In chapter 2 we provide a cut-to-the-bone introduction to the λ-calculus, two syntaxes (named and nameless), its evaluation semantics, and the shortest glossary of rewriting theory.
- To evaluate λ-terms one must choose an *evaluation strategy*: in chapter 3 we shortly introduce the most common strategies for the λ-calculus, then settle on Call-by-Value

(CbV). We introduce historic Plotkin's CbV λ-calculus, which is however insufficient for our goals because it is adequate only for the λ-terms coming from programming languages, *i.e.* closed terms. For proof assistants, we need also to consider open λ-terms, *i.e.* terms with free variables: we introduce so-called *fireball* λ-calculus, the simplest presentation of Open CbV.

- In chapter 4 we discuss how to implement evaluation, abstractly. Naïve implementations suffer from an exponential-time slowdown called *size explosion*, due to the textual representation of λ-terms: this issue is only solved by *sharing*, a mechanism to reuse subterms during evaluation. We add *explicit sharing* (ES) to the syntax of the λ-calculus. Through sharing, we decompose the meta-level operations at work during evaluation and progressively internalize them in the calculus: we walk through different calculi with ES, finally reaching abstract machines. We discuss in what way abstract machines simulate or implement a calculus, and the techniques to prove simulation. Finally, we show how to extimate the asymptotic overhead of a machine by bounding the number of machine transitions.

- In chapter 5 we introduce strong evaluation and explain why it is relevant. We show one of the few existing abstract machines for strong evaluation, and explain how the literature has optimized it. We then focus on the intermediate case between traditional abstract machines and strong machines, *i.e.* machines for open evaluation.

- In chapter 6 we look at sharing in λ-terms from a different perspective: when λ-terms are represented as syntax trees, sharing amounts to allowing the nodes of the tree to have multiple parents, turning the λ-tree into a λ-graph. We will use λ-graphs in part III to solve Comparison.

Part II: Compute

In this part we focus on Computation. The goal of this part is to improve and simplify the design of already existing abstract machines for Open CbV evaluation, in order to make them more suitable to be generalized to the strong setting. To do so, we present an alternative representation of λ-terms enabled by explicit sharing, which we call *crumbled*. Crumbled terms simplify both the enforcement of the CbV strategy, and the design of the abstract machines implementing it.

- In chapter 7 we introduce crumbled forms, and show how to translate back and forth from λ-terms to crumbled terms. We also discuss some issues of related representations, such as *administrative normal forms* and *continuation-passing style* translations.

- In chapter 8 we introduce the Crumble GLAM, our abstract machine working on crumbled terms. We show that it implements the CbV strategy with only a linear-time overhead.

- In chapter 9 we generalize the Crumble GLAM from the previous chapter to the case of open terms, obtaining the Open Crumble GLAM. A benefit of our approach is that all the

results about the Crumble GLAM scale smoothly to the Open Crumble GLAM: we show that the Open Crumble GLAM implements Open CbV with only a linear-time overhead.

- Plot twist: the machines of chapter 8 and chapter 9 are not actual machines. In chapter 10 we refine them, obtaining the real abstract machines, that we name Pointed Crumble GLAM and Open Pointed Crumble GLAM.
- In chapter 11 we provide an OCaml implementation of the Pointed Crumble GLAMs of chapter 10, together with a discussion of all the necessary data structures and low-level implementation details.

Part III: Compare

In this part we focus on Comparison. On λ-terms with sharing, Comparison amounts to what we call *sharing equality*, *i.e.* the problem of deciding whether two shared λ-term are equal up to the unfolding of the sharing. As already mentioned, unfolding the sharing is an exponential-time operation; instead, we require sharing equality to be checkable in time proportional to the size of the shared terms, in order to match the linear-time overhead of evaluation.

- In chapter 12 we develop a theory of sharing equality which allows to decompose sharing equality in two parts: a part that only manipulates the first-order structure of λ-terms, and a part that takes into account the higher-order nature of λ-terms (binders and variables).
- We turn to the algorithmic side of sharing equality. In chapter 13, we showcase problems related to sharing equality, showing also their computational complexity.
- In chapter 14 we provide our linear-time algorithm for sharing equality, together with full proofs of correctness, completeness, termination and linearity.

Part IV: Conclusions

To conclude, in chapter 15 we provide directions on how to extend the work of this dissertation.

1.5 Contributions

We conclude this introductory chapter by anticipating the contributions of this dissertation:

Crumbling abstract machines. We introduce the crumbled representation of λ-terms: by decomposing nested application, we study how our crumbled representation of λ-terms impacts on the design and the efficiency of abstract machines for CbV evaluation. About the design, it removes the need for data structures encoding the evaluation context, such as the applicative stack and the dump, that get encoded in the environment. About efficiency, we

show that there is no slowdown, clarifying in particular a point raised by Kennedy, about the potential inefficiency of such a representation. Moreover, we prove that everything smoothly scales up to the delicate case of open terms, needed to implement proof assistants. Along the way, we also point out that continuation-passing style transformations—that may be alternatives to our representation—do not scale up to the open case.

A theory of sharing equality. We develop a re-usable, self-contained, and clean theory of sharing equality, independent of the algorithm that computes it. Some of its concepts are implicitly used by other authors, but never emerged from the collective unconscious before (*propagated queries* in particular)—others instead are new. A key point is that we bypass the use of α-equivalence by relating sharing equalities on DAGs with λ-terms represented in a locally nameless way. In such an approach, bound names are represented using de Bruijn indices, while free variables are represented using names—thus α-equivalence collapses on equality. The theory culminates with the sharing equality theorem, which connects equality of λ-terms and sharing equivalences on shared λ-terms, under suitable conditions.

A linear-time sharing equality algorithm. We provide a linear-time algorithm for sharing equality by adapting a linear algorithm for first-order unification by Paterson and Wegman (PW) to λ-terms with sharing. Our algorithm is actually composed by a two-levels, modular approach:

- *Blind check*: a reformulation of PW from which we removed the management of meta-variables for unification. It is used as a first-order test on λ-terms with sharing, to check that the unfolded terms have the same skeleton, ignoring variables.
- *Variables check*: a straightforward algorithm executed after the previous one, testing α-equivalence by checking that bisimilar bound variables have bisimilar binders and that two different free variables are never equated.

The decomposition plus the correctness and the completeness of these checks crucially rely on the sharing equality theory developed in the previous point.

Publications

The majority of the results have been obtained in collaboration with Claudio Sacerdoti Coen[1], Beniamino Accattoli[2], and Giulio Guerrieri[3].

The contributions of this dissertation are based on the following publications:

1. Andrea Condoluci, Beniamino Accattoli, and Claudio Sacerdoti Coen (2019). "Sharing Equality is Linear". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019.*

[1] https://www.cs.unibo.it/~sacerdot/
[2] https://sites.google.com/site/beniaminoaccattoli/
[3] https://www.irif.fr/~giuliog/

Ed. by Ekaterina Komendantskaya. ACM, 9:1–9:14. ISBN: 978-1-4503-7249-7. DOI: 10.1145/3354166.3354174. Long version available on ArXiV.[4]

2. Beniamino Accattoli et al. (2019a). "Crumbling Abstract Machines". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 4:1–4:15. ISBN: 978-1-4503-7249-7. DOI: 10.1145/3354166.3354169. Long version available on ArXiV.[5]

[4] Andrea Condoluci, Beniamino Accattoli, and Claudio Sacerdoti Coen (2019). "Sharing Equality is Linear". In: *arXiv e-prints*. arXiv: 1907.06101.

[5] Beniamino Accattoli et al. (2019b). "Crumbling Abstract Machines". In: *CoRR* abs/1907.06057. arXiv: 1907.06057.

Note on skimming

Throughout the dissertation, we will use the following page-wide dashed line:

to separate the initial, central part of a section from the secondary part, containing technical details that can be skipped at a first read.

Part I

Preliminaries

Chapter 2

The λ-Calculus

The λ-calculus is a quite minimal and elegant formal system (Church, 1941; Barendregt, 1984) widely recognised as the foundation of functional programming languages (Landin, 1965). The λ-calculus is a very well-known theory, which usually does not need more than a couple of paragraphs to be recalled. However, being an essential prerequisite for the rest of this thesis, I have decided to still provide a less concise introduction, so to make this text as self-contained as possible. Nevertheless, all readers with a basic knowledge of the λ-calculus are welcome and encouraged to skip right to the next chapter!

In the following, we briefly introduce its syntax (section 2.1) and reduction semantics (section 2.2). In section 2.3 we provide a brief glossary of rewriting theory with notions that are common to all the calculi in this dissertation. Finally, we present in section 2.4 an alternative but standard syntax for λ-terms where variables are represented by natural numbers instead of usual named variables.

2.1 Syntax

The terms of the λ-calculus are called λ-terms, and their syntax is defined by the following grammar[1]:

Named terms

$$t, s, u, r \quad ::= \quad x \mid \lambda x.t \mid t\ s$$

We denote variables with x, y, z, w and assume an enumerable set of variable names called $\mathcal{V}$. As one can see, a λ-term is either a variable, an abstraction $\lambda x.t$ of a function body t over a variable x, or the application $t\ s$ of two terms t and s. Examples of λ-terms are:

- $\lambda x.x$ denotes a function which takes in input an argument x and then just returns it. We call it I for "identity".

[1] To define the syntax of various objects we use the standard notation technique called *Backus–Naur form*.

- $\lambda x.\lambda y.x$ denotes a function which takes in input two arguments x and y, and then returns the first one. It is usually called "first projection", whereas $\lambda x.\lambda y.y$ is the "second projection".

- $y\ (\lambda x.x)$ is the the term obtained by applying the identity to the variable y. Here the variable y can be regarded as a symbol, with no given value nor any property other than its identifier name.

Abstraction is a binding construct, hence it introduces a notion of scoping. The status of a variable changes according to the scopes in which it occurs, and it can be either *free* or *bound*. Intuitively, a variable occurrence is bound if it is inside the scope of an abstraction that binds it, for instance, the x in $\lambda x.\lambda y.x$; a variable is free in a term if not all occurrences of that variable are bound, for instance the x in $x\ (\lambda x.x)$. We can define formally the free variables of a term as follows:

Definition 2.1 \ Free variables

Let t be a λ-term, and x be a variable. We say that x is free in t if $x \in \mathsf{fv}(t)$, where $\mathsf{fv}(t)$ is the set of free variables of t defined by structural induction on t:

$$\begin{aligned} \mathsf{fv}(x) &:= \{x\} \\ \mathsf{fv}(t\ s) &:= \mathsf{fv}(t) \cup \mathsf{fv}(s) \\ \mathsf{fv}(\lambda x.t) &:= \mathsf{fv}(t) \setminus \{x\} \end{aligned}$$

Likewise, we can define by induction over t the set $\mathsf{bv}(t)$ of bound variables in t, and the set $\mathrm{vars}(t)$ of all the variables occurring in t. We call a λ-term t closed if $\mathsf{fv}(t) = \emptyset$, open otherwise.

Closed *vs* open Closed λ-terms are also called "programs"[2], because the source code of programming languages cannot refer to variables that are not initialized, *i.e.* free variables. We call "closed λ-calculus" the case where λ-terms are required to be closed: it is the ideal setting to model functional programming languages, and it enjoys a simpler semantics. However, other tools based on the λ-calculus—for example proof assistants—need to manipulate also open λ-terms. In chapter 5 we will discuss the additional challenges of the open setting.

Variable names The representation of λ-terms that we have just provided is called named because variables bear names, in the sense that they are identifiers named with $x, y, \ldots$. As we will see in section 2.4, providing names to variables is not strictly necessary: one could simply represent variables with natural numbers. Different representations have different pros and cons, but the biggest point in favour of named λ-terms is being quite human-friendly—in fact every programming language uses named variables.

[2] Or "combinators", for historical reasons.

The problem with named λ-terms is that variables with different names are not exactly different variables. Consider for instance the following two ways of writing down the identity: $\lambda x.x$ and $\lambda y.y$. These two terms clearly represent the same λ-term, just with renamed bound variables: thus the variable x in the first term is somewhat the same as the variable y in the second one. If we instead consider the λ-terms x and y by themselves (where x and y are two different names), then the two are clearly different λ-terms.

As a consequence, equality on named terms is not just syntactic equality, but what we call α-equality[3]: we must identify terms up to the renaming of bound variables. Even though α-equality is conceptually a simple operation, providing a formal definition is not straightforward, which already suggests how complex is working with names explicitly.

In order to define α-equality, we first specify what it means to rename a variable in a term:

Definition 2.2 \ Renaming

Let t be a named term, and x, y be variables such that $y \notin \mathrm{vars}(t)$. We define the renaming of x for y in t, in symbols $t\{x \Leftarrow y\}$, by structural induction on t:

- *Same variable:* $x\{x \Leftarrow y\} := y$;
- *Different variable:* $z\{x \Leftarrow y\} := z$ if $x \neq z$;
- *Application:* $(t\ s)\{x \Leftarrow y\} := (t\{x \Leftarrow y\})\ (s\{x \Leftarrow y\})$;
- *Abstraction (i):* $(\lambda x.t)\{x \leftarrow y\} := \lambda x.t$;
- *Abstraction (ii):* $(\lambda z.t)\{x \Leftarrow y\} := \lambda z.(t\{x \Leftarrow y\})$ if $z \neq x$.

The following definition of α-equality is a variant of the one in Selinger, 2008:

Definition 2.3 \ α-equality $=_\alpha$

We define α-equality as the smallest equivalence relation over named terms such that:

1. *Application*: $t\ s =_\alpha u\ r$ if $t =_\alpha u$ and $s =_\alpha r$;
2. *Abstraction*: $\lambda x.t =_\alpha \lambda x.s$ if $t =_\alpha s$;
3. *Renaming*: $\lambda x.t =_\alpha \lambda y.(t\{x \Leftarrow y\})$ if $y \notin \mathrm{vars}(t)$.

Points 1 and 2 state that α-equality is compatible with the constructors of the λ-calculus, and Point 3 performs the actual renaming of a bound variable: the renaming of multiple bound variables is obtained by transitivity, since $=_\alpha$ is required to be an equivalence relation (*i.e.* reflexive, symmetric, and transitive).

It is standard to identify λ-terms up to α-equality: formally, one defines λ-terms as the quotient of named terms over the equivalence relation $=_\alpha$. The alternative, nameless syntax for λ-terms that we will introduce in section 2.4 actually avoids the problems of reasoning up to α-equivalence classes.

[3] Usually called α-conversion.

Substitutions. Before defining how λ-terms "compute", we need to define the fundamental operation of substitution. We extend the notation introduced for renamings in definition 2.2 to $t\{x \leftarrow s\}$, which is the term obtained from t by replacing all free occurrences of x with the term s. In presence of binders, substitution is not as simple as textual replacement (also known as "grafting"): for instance, the result of $(\lambda x.y)\{y \leftarrow x\}$ should not be $\lambda x.x$ but $\lambda z.x$ for any $z \neq x$. This kind of substitution is called "capture-avoiding", because it substitutes and at the same time renames the bound variables encountered in order to prevent capturing the free variables of the term to be substituted.

Definition 2.4 \ Substitution

Let t, s be λ-terms, and x be a variable. We define the substitution of x with s in t, in symbols $t\{x \leftarrow s\}$, by induction on the size of t:

- *Same variable:* $x\{x \leftarrow s\} \coloneqq s$;
- *Different variable:* $y\{x \leftarrow s\} \coloneqq y$ if $x \neq y$;
- *Application:* $(t\ u)\{x \leftarrow s\} \coloneqq (t\{x \leftarrow s\})(u\{x \leftarrow s\})$;
- *Abstraction (i)* $(\lambda x.t)\{x \leftarrow s\} \coloneqq \lambda x.t$;
- *Abstraction (ii):* $(\lambda y.t)\{x \leftarrow s\} \coloneqq \lambda y'.(t\{y \Leftarrow y'\}\{x \leftarrow s\})$ if $x \neq y$ for some y' such that $y' \notin \{x\} \cup \mathrm{vars}(t) \cup \mathsf{fv}(s)$.

We also define substitutions as objects *per se*, which we denote by σ: they have the form $\sigma = \{x_1 \leftarrow t_1, \ldots, x_k \leftarrow t_k\}$ where $x_1, \ldots, x_k$ are distinct variables. In particular, if $k = 0$ then $\sigma = \{\}$, called the *identity* or *empty* substitution. We define the domain of σ as $\mathsf{dom}(\sigma) \coloneqq \{x_1, \ldots, x_k\}$, and we say that $\sigma(x_i) \coloneqq t_i$ for $i = 1 \ldots k$. We denote by $t\sigma$ the *simultaneous* capture-avoiding substitution of σ in t, which is standard and can be defined in a similar way as in definition 2.4.

Definition 2.5 \ Operations on substitutions

Let σ and σ' be substitutions such that $\mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma') = \emptyset$. We define the *composition* $\sigma \circ \sigma'$ as follows:

$$\sigma \circ \sigma' \coloneqq \{x_1 \leftarrow t_1\sigma', \ldots, x_k \leftarrow t_k\sigma'\} \cup \sigma'$$

if $\sigma = \{x_1 \leftarrow t_1, \ldots, x_k \leftarrow t_k\}$.

We also define the *union* $\sigma \cup \sigma'$ as follows:

$$\sigma \cup \sigma' \coloneqq \{x_1 \leftarrow t_1, \ldots, x_k \leftarrow t_k, y_1 \leftarrow s_1, \ldots, y_h \leftarrow s_h\}$$

if $\sigma' = \{y_1 \leftarrow s_1, \ldots, y_h \leftarrow s_h\}$.

Size. We also define the size $|t|$ of a λ-term t as follows:

Definition 2.6 \ Size $|t|$ of a λ-term

We define the size $|t|$ of a λ-term t, by structural induction on t:

$$\begin{array}{rcl} |x| & \coloneqq & 0 \\ |t\,s| & \coloneqq & 1 + |t| + |s| \\ |\lambda x.t| & \coloneqq & 1 + |t|. \end{array}$$

2.2 Evaluation

The behaviour of λ-terms is usually described in terms of individual steps of computation, approach called *operational* or *small-step semantics*. A computation step is traditionally called *reduction* or *evaluation*, and it is defined in terms of rules: the classical λ-calculus encompasses only one reduction rule, called β and defined as follows.

β-rule (top-level)

$$(\lambda x.t)\ s \ \mapsto_\beta \ t\{x \leftarrow s\}$$

Intuitively β performs a function call, executed as an atomic operation: $(\lambda x.t)\ s$ is called a *redex*, and reduces in one step to $t\{x \leftarrow s\}$ which is the function body t where every free occurrence of the formal parameter x is replaced by the argument s.

For instance, consider the term $I\ t$ where $I = \lambda x.x$ and t is any λ-term: then $I\ t \mapsto_\beta x\{x \leftarrow t\} = t$ as expected. Another example:

Example 2.7 \ Looping

Let $\delta \coloneqq \lambda x.xx$ and $\Delta \coloneqq \delta\delta$. We call δ the *duplicator* because it duplicates its argument by applying it to itself, and we call Δ the *looping combinator* becase it reduces to itself:

$$\Delta = \delta\delta = (\lambda x.xx)\ \delta \mapsto_\beta xx\{x \leftarrow \delta\} = \delta\delta = \Delta.$$

Note however that $\mapsto_\beta$ can only act at top-level: if the redex is deep in the λ-term, as for instance in $(I\ t)\ s$, then $\mapsto_\beta$ is not applicable. One way to push the effect of β inside complex terms is by means of so-called reduction or evaluation contexts, technique known as *reduction semantics*.

Contexts

$$C ::= \langle\cdot\rangle \mid C\,t \mid t\,C \mid \lambda x.C$$

Contextual closure

$$C\langle t\rangle \quad \to_\beta \quad C\langle s\rangle \quad \text{if } t \mapsto_\beta s$$

β-reduction / **Figure 2.1**

Evaluation contexts serve as a way to locate a precise subterm in a complex term, exposing the desired redex. Such contexts are basically λ-terms which contain exactly one occurrence of a special constant $\langle\cdot\rangle$ called the hole, which is a placeholder for a removed subterm. We call the contexts described by the entry C in fig. 2.1 "general" contexts because they can reach any redex, no matter where it is located in a term. The basic operation on (whatever notion of) contexts is the plugging $C\langle t\rangle$ of a λ-term t for the hole $\langle\cdot\rangle$ in C: simply the hole is removed and replaced by t, possibly capturing variables.

Definition 2.8 \ Plugging

Let C be a context and t be a λ-term. We define the plugging of t in C, in symbols $C\langle t\rangle$, by structural induction on C:

$$\begin{aligned}
\langle\cdot\rangle\langle t\rangle &:= t\\
(C\ s)\langle t\rangle &:= C\langle t\rangle\ s\\
(s\ C)\langle t\rangle &:= s\ C\langle t\rangle\\
(\lambda x.C)\langle t\rangle &:= \lambda x.C\langle t\rangle.
\end{aligned}$$

Evaluation contexts can also be composed, by simply plugging one into the other:

Definition 2.9 \ Composition of contexts

Let C, C' be contexts. We define their composition $C\langle C'\rangle$ by induction on the structure of C:

$$\begin{aligned}
\langle\cdot\rangle\langle C'\rangle &:= C'\\
(C\ s)\langle C'\rangle &:= C\langle C'\rangle\ s\\
(s\ C)\langle C'\rangle &:= s\ C\langle C'\rangle\\
(\lambda x.C)\langle C'\rangle &:= \lambda x.(C\langle C'\rangle).
\end{aligned}$$

We can now define one-step β-reduction $\to_\beta$ as the closure of $\mapsto_\beta$ under evaluation contexts, as in fig. 2.1. Let us go back to the term $(I\ t)\ s$ mentioned above: it now holds that $(I\ t)\ s \to_\beta t\ s$ because $I\ t \mapsto_\beta t$ and by using the evaluation context $C := \langle\cdot\rangle\ s$.

We will also call the relation $\to_\beta$ full or unrestricted β-reduction. As we will see in chapter 3, the grammar of evaluation contexts can be restricted in order to disallow the reduction of redexes in certain positions of a term: in this way one can force an order of reduction to redexes, *i.e.* an evaluation strategy.

2.3 Glossary

Throughout this dissertations we will consider various variants of the λ-calculus enriched with different reduction rules. In this section, we introduce some terminology from rewriting theory that is common to these variants: to define it uniformily, we parametrize each notion with a "C" (for calculus) and "R" (for reduction). For instance, in order to specialize the following definitions to the classical λ-calculus, one can use the symbol "λ" in place of C and "β" in place of R.

We denote by C-calculus a generic dialect of the λ-calculus, whose terms we call C-terms. Let $\to_{\mathsf{R}}$ be a reduction relation over the terms of the C-calculus, and let t and s be C-terms.

Reduction

- "Contracting" or "firing" a redex are both synonyms of "reducing" a redex.
- We say that "$t \to_{\mathsf{R}}^{n} s$" if t reduces to s in exactly n $\to_{\mathsf{R}}$ steps.
- We say that "t R-reduces to s", in symbols "$t \to_{\mathsf{R}}^{*} s$", if t reduces to s by zero or more $\to_{\mathsf{R}}$ steps.
- We say that "t and s are R-convertible" if $t =_{\mathsf{R}} s$, where $=_{\mathsf{R}}$ is the equivalence relation over C-terms generated by $\to_{\mathsf{R}}$.[4]
- We say that "t is R-strongly normalizing" if there exists no infinite sequence $(t, t', t'', \dots)$ of C-terms starting with t such that $t \to_{\mathsf{R}} t' \to_{\mathsf{R}} t'' \to_{\mathsf{R}} \cdots$.
- $\to_{\mathsf{R}}$ is strongly normalizing if every C-term t is R-strongly normalizing.
- $\to_{\mathsf{R}}$ is deterministic if for every C-term t, $t \to_{\mathsf{R}} s$ and $t \to_{\mathsf{R}} s'$ implies $s = s'$.
- $\to_{\mathsf{R}}$ is confluent if for every C-terms t, s, s' such that $t \to_{\mathsf{R}}^{*} s$ and $t \to_{\mathsf{R}}^{*} s'$, there exists a C-term u such that $s \to_{\mathsf{R}}^{*} u$ and $s' \to_{\mathsf{R}}^{*} u$.

Normal forms

- We say that "t is in R-normal form" if it does not R-reduce further, *i.e.* there is no C-term t' such that $t \to_{\mathsf{R}} t'$.
- We say that "t has R-normal form s" if t R-reduces to s, and s is in R-normal form. If a term t has a unique normal form, we denote it by $\mathrm{nf}_{\mathsf{R}}(t)$.
- $\to_{\mathsf{R}}$ has the unique normal form property if every C-term t has a unique normal form.

We will omit the parameters C and R when clear from the context.

Properties. We quickly mention a couple of fundamental properties of the λ-calculus. The first property is the uniqueness of normal forms:

[4] The convertibility relation $=_{\mathsf{R}}$ must also respect the relevant notion of α-equality for the C-calculus.

Theorem 2.10 \ Uniqueness of β-normal forms (Selinger, 2008)

Full β-reduction in the λ-calculus is confluent, and hence it has the unique normal form property.

We now have enough terminology to state "naïve convertibility", *i.e.* the fact that Computation and Comparison can be unweaved as stated in the introduction:

Theorem 2.11 \ Naïve convertibility

Let t and s be λ-terms. If t and s are strongly normalizing, then $t =_\beta s$ if and only if $\mathrm{nf}(t) =_\alpha \mathrm{nf}(s)$.

Proof. Since t and s are strongly normalizing, $t \to_\beta^* \mathrm{nf}(t)$ and $s \to_\beta^* \mathrm{nf}(s)$. By the definition of $=_\beta$, $t =_\beta s$ if and only if $\mathrm{nf}(t) =_\beta \mathrm{nf}(s)$. By the uniqueness of normal forms (theorem 2.10), $\mathrm{nf}(t) =_\beta \mathrm{nf}(s)$ if and only if $\mathrm{nf}(t) =_\alpha \mathrm{nf}(s)$.

A consequence of theorem 2.11 is that, in order to decide whether $t =_\beta s$, one can:

1. (Computation) Reduce t and s to their β-normal forms $\mathrm{nf}(t)$, $\mathrm{nf}(s)$.
2. (Comparison) Compare $\mathrm{nf}(t)$ and $\mathrm{nf}(s)$.

2.4 Nameless

Named terms are the standard way of representing λ-terms, but reasoning in presence of names is cumbersome, especially names for bound variables as they force the introduction of α-equivalence classes.

Let us for a moment consider only closed λ-terms, *i.e.* terms with no free variables. In this setting the nameless approach is quite popular: by using so-called de Bruijn indices, variables simply consist of an *index*, a natural number indicating the number of abstractions that occur between the abstraction to which the variable is bound and that variable occurrence. For example, the named term $\lambda x.\lambda y.x$ and the nameless term $\lambda\lambda\#1$ denote the same λ-term. In the closed setting, substitution coincides with grafting (since capturing free variables is not possible) and reduction simply becomes a first-order rewriting system.

The nameless representation can be extended to open λ-terms by assigning de Bruijn indices also to free variables, in such a way that free variables get indices that exceed the number of enclosing abstractions: for instance, the named term $y\ (\lambda x.x\ y\ z)$ corresponds to the nameless term $\#0\ (\lambda\#0\ \#1\ \#2)$. But open terms cause various complications, for example different occurrences of the same free variable unnaturally have different indices (like the free variable y in the term above that is assigned both indices $\#0$ and $\#1$). Also, substitution is not grafting anymore, and one needs to introduce a notion of lifting to avoid capture during substitution.

Since in this dissertation we are forced to work with open terms (because proof assistants require them) the most comfortable representation seems the locally nameless representation (see Charguéraud, 2012 for a thorough discussion). This representation combines named and nameless, by using de Bruijn indices for bound variables (thus avoiding the need for α-equivalence), and names for free variables. The syntax of locally nameless terms is:

Locally nameless terms

$$t, s \quad ::= \quad \#i \mid x \mid t\, s \mid \lambda t \quad (i \in \mathbb{N},\, x \in \mathcal{V})$$

where $\#i$ denotes a bound variable of de Bruijn index i ($\mathbb{N}$ is the set of natural numbers), and x denotes a free named variable.

In this dissertation, we will use both the named and the nameless representations of λ-terms. Whenever discussing about λ-calculus or evaluation, we prefer named terms because they are much more human-friendly. However in section 6.1 we will represent λ-terms as graphs, or better as λ-graphs: in order to relate λ-graphs and λ-terms we will use the locally nameless representation, since in λ-graphs variables are represented simply as nodes, and nodes naturally bear no name. As we shall see, to compare two bound variable nodes one compares their binders, but to compare two free variable nodes one uses their identifier, which in implementations is usually their memory address or a user-supplied string.

The benefits of the locally nameless approach will be unequivocal in part III, when we discuss sharing equality, *i.e.* the counterpart in λ-graphs of α-equality in λ-terms. By switching to locally nameless terms and avoiding to reason explicitly on bound names and α-equality, we will obtain much more elegant and shorter proofs.

Chapter 3

Call-by-Value

As already mentioned, β-reduction is not deterministic, meaning that a λ-term can potentially reduce to multiple terms, one for each redex it contains. For instance, the term $(\lambda x.y)\,(I\,z)$ reduces in one step either to y (by using the evaluation context $\langle\cdot\rangle$) or to $(\lambda x.y)\,z$ (by using the context $(\lambda x.y)\,\langle\cdot\rangle$).

To consider the λ-calculus as a programming language, one usually fixes an evaluation strategy, *i.e.* a restriction of β-reduction that is deterministic. In fact, all programming languages use evaluation strategies to decide when to evaluate the arguments of a function, so that the control flow is roughly predictable. This is because evaluating an expression with different orders may in principle change the outcome, but in the full λ-calculus this is not the case since all β-normal forms reached by evaluation are equal (theorem 2.10). Different strategies may require a different number of β-steps, and some may even diverge: however in this dissertation we study evaluation with proof assistants in mind, and in that settings all terms are typed and thus strongly-normalizing.

Evaluation strategies for programming languages are called weak, meaning that they do not reduce redexes inside the bodies of abstractions: only when arguments are supplied to a function, the function is invoked and the body evaluated. Weak evaluation can be specified in the λ-calculus by a syntactic restriction of evaluation contexts in the following way:

Weak evaluation contexts

$$W ::= \langle\cdot\rangle \mid W\,t \mid t\,W$$

The only difference with respect to general contexts C is that in the grammar for W the production $\lambda x.W$ is omitted. We call *weak λ-calculus* the calculus where reduction is the contextual closure of $\mapsto_\beta$ under weak evaluation contexts.

The most common (families of) weak evaluation strategies discussed in the setting of λ-calculus are the following:

Call-by-Value (CbV) is probably the most common evaluation strategy, used by many programming languages like C, OCaml, or Scala. In CbV, each argument of a function must

be completely evaluated before the function is called; it is also known as "applicative order" or strict evaluation, because it forces the evaluation of all function arguments. By doing so it avoids some duplication of work: if an argument is used multiple times in the body of a function, it will not be necessary to evaluate it again. The downside is that CbV may perform useless computation in case the evaluated argument will never be used.

As defined up to now, CbV is not yet a proper evaluation strategy, because we have not fixed yet the order in which the arguments of a function should be evaluated: many languages (such as Common Lisp, Eiffel and Java) evaluate arguments *left-to-right*, some *right-to-left*, and others (such as Scheme, OCaml and C) leave the order unspecified.

Call-by-Name (CbN) is the strategy which does not evaluate function arguments before invoking it: the leftmost applied function is called at each step, and each unevaluated argument is used in place of every occurrence of the formal parameter. CbN is somewhat dual to CbV in that it evaluates expressions only when actually used, but it may instead perform redundant computation in case an argument is used multiple times.

CbN is obtained in the λ-calculus by the following restriction on evaluation contexts, called H for "head":

Weak head contexts

$$H ::= \langle\cdot\rangle \mid H\ t$$

where the production $t\ H$ is omitted because CbN does not evaluate the arguments of applications.

CbN can be considered the weak version of the well-known strategy called "normal order" or Leftmost-Outermost (LO), always reducing the redex that is the leftmost outermost (= not contained in another redex) in the term. LO is a "strong" strategy (we will discuss strong evaluation in chapter 5), and it has a special status in the λ-calculus: it is a *normalizing* strategy, *i.e.* whenever a λ-term has a β-normal form, then the normal order strategy will reduce that term to normal form. LO evaluation can easily be defined through a restriction on evaluation contexts, which can be found in definition 5.2 on page 68.

Call-by-Need (CbNeed) can be considered a cached version of CbN: functions are invoked without first evaluating the arguments, but if an argument is later used and evaluated, its value is stored for subsequent uses. It is also called lazy, because it does not evaluate arguments unless necessary. CbNeed is supposed to represent the best of both CbV and CbN: at a first approximation, CbNeed proceeds like CbN, evaluting a function body without evaluating the argument; only when the moment arrives that the value of a previous argument is needed, then CbNeed evaluates it like CbV would have. This means that each argument is evaluated at most once.

CbNeed cannot be defined in the classical λ-calculus as a straightforward restriction on

contexts, but requires a shared[1] representation of λ-terms (see *e.g.* Maraist, Odersky, and Wadler, 1998). CbNeed is the strategy used by the Haskell language[2].

Each evaluation strategy has its advantages and disadvantages. CbN is the easiest to define and implement, but also the most inefficient, and in fact no real programming language uses it. The debate CbV *vs* CbNeed (*i.e.* strict *vs* lazy) among functional programmers is, historically, quite heated. In theory CbNeed should be more efficient, because it evaluates any λ-term using the same number or fewer β-steps than CbV: CbNeed actually requires the least number of β-steps than any weak strategy (property called "weak optimality", see Balabonski, 2013).[3] There are however different aspects to consider:

- From a theoretical point of view, CbV has a neat definition in the λ-calculus and an harmonic theory (see section 3.1); expressing CbNeed in the λ-calculus instead requires either to add sharing or to extend the calculus with different reduction rules (Ariola and Felleisen, 1997; Chang and Felleisen, 2012).

- From an implementation point of view, CbNeed evaluation may suffer an additional overhead due to its mechanism of "thunks": at each function invocation, CbNeed implementations suspend the argument and allocate a new *closure*, *i.e.* a mapping associating each variable used by the code of the argument with the correct reference at the time the closure was created.[4] In case the data type of an argument is primitive, for instance a boolean or a number, CbV would reduce it right away to its value, which requires only one word in memory. Clearly allocating and managing a closure requires more effort than simply handling an atom, hence CbNeed implementations may in the end suffer higher computational constants. Another issue with CbNeed concerns memory management, and more precisely *garbage collection*, *i.e.* reclaiming the memory occupied by objects that are no longer used: unevaluated thunks may artifically retain objects in memory, marking them as alive even though they may not be actually needed by evaluation.

- From programmers point of view, CbV has one additional advantage: it simplifies the reasoning about asymptotic complexity of algorithms (Okasaki, 1999). In strict languages, in fact, reasoning about the running time of a given program is relatively straightforward, since one can predict when subexpressions will be evaluated just by looking at the source code. On the contrary, in lazy languages it is very diffuclt to predict when a given subexpression will be evaluated, if ever. This has implications that concern users, for instance in the design and complexity analysis of data structures: strict languages are well-suited for worst-case data structures, while lazy languages for amortized ones (Okasaki, 1999).

[1] See section 4.2 and chapter 6.

[2] `https://www.haskell.org/`

[3] However a low number of β steps does not necessarily make a strategy efficient: it depends on the underlying cost model. For instance, *optimal reduction* aims at minimizing the number of β steps, but only at the cost of huge hidden computations required to achieve optimality, see also page 85.

[4] A formal definition of closures is given on 48.

Why CbV? As we will discuss in chapter 5, proof assistants require a different kind of evaluation than programming languages, called *strong* evaluation. In the strong setting of proof assistants, different strategies do not actually affect the result of evaluation; in this dissertation however we focus on the CbV strategy, mainly for the following reasons:

- Currently, the Coq proof assistant can perform evaluation either through an interpreter or through compilation: the interpreter supports all the main strategies outlined above; compilation (to bytecode or to native code) instead implements CbV and is much faster, having performance somewhat comparable to that of the OCaml language (in the weak setting). Some Coq libraries, like the aforementioned formalization of the four-color theorem, crucially rely on the speed of the Strong CbV compiler. Performing efficient CbNeed is very desirable, but as of today there simply do not exist efficient strong machines for CbNeed.
- As already mentioned CbNeed evaluation requires a mechanism of *thunk*s (to suspend the evaluation of arguments until needed) and of *update* (to store the value of an argument after evaluation). Since a thunk may be shared in memory, *i.e.* multiple objects may refer to it at the same time, updating it in place results in all objects referring instead to its computed value: this can be seen as a way to share not only syntax, but also *computation*. Traditional evaluation machines for CbV, instead, do not need to update: arguments can be evaluated right away since computation is not shared. By avoiding the need to update suspensions, one obtains simpler machines, and can thus rely on simpler properties that are invariant under execution.

  Note however that in the strong case, in order to be efficient it will be necessary to share computation also when performing CbV (for more information, see the discussion on page 219). Being conceptually similar to the update mechanism of CbNeed, we hope to generalize our approach also to CbNeed in the future.

3.1 Plotkin's CbV

In this section we restrict the λ-calculus to the Call-by-Value strategy obtaining λ_{Plot}, which was first introduced and studied by Plotkin, 1975. As already mentioned, in CbV the arguments to functions must be fully evaluated before the function is invoked: as we will see below, in the setting of λ-calculus this is obtained by restricting β to fire only when the argument of the redex is a value.

The syntax and reduction semantics of λ_{Plot} are in fig. 3.1.

Syntax

Terms	t, s	$::= v \mid t\ s$
Values	v	$::= x \mid \lambda x.t$
Right v-contexts	R	$::= \langle\cdot\rangle \mid t\ R \mid R\ v$

Reduction rule at top-level

$$(\lambda x.t)\ v \mapsto_{\beta/v} t\{x \leftarrow v\}$$

Contextual closure

$$R\langle t\rangle \to_{\beta/v} R\langle s\rangle \quad \text{if} \quad t \mapsto_{\beta/v} s$$

Plotkin's calculus λ_{Plot} / **Figure 3.1**

Syntax. The grammar for terms in fig. 3.1 produces the same terms as the terms of the classical λ-calculus, but here variables and abstractions have a special status: we call them values, and denote them by "v". We implicitly reuse for λ_{Plot} the basic definitions from chapter 2, like capture-avoiding substitutions, the set of free variables $\mathsf{fv}(\bullet)$, and the notion of open and closed terms.

Evaluation. As already mentioned, we restrict the β rule in such a way that it can be fired only when the argument of the redex is a value: we thus obtain the top-level β/v rule.

The reduction relation is then obtained as usual by contextual closure: for λ_{Plot} we use *right evaluation v-contexts*, denoted by the entry R in fig. 3.1. These contexts force an order of evaluation called "right-to-left": due to the production $R\ v$, one can reduce the left part of an application only if the right part has already been reduced to a value. We could as well work with left-to-right evaluation (but we won't) by using the following contexts:

Left v-contexts

$$L \quad ::= \langle\cdot\rangle \mid v\ L \mid L\ t$$

Open λ-terms. CbV evaluation is defined in λ_{Plot} for all λ-terms, even for those that do not correspond to programs in a functional programming language: open λ-terms (terms with free variables) do not represent any program because in computers all variables are bound at runtime to a value in memory. Actually, as we discuss in the next section, the operational semantics that we have provided here is *adequate* only for closed λ-terms. When restricted to closed λ-terms, λ_{Plot} is called Closed CbV: in this setting values cannot be variables, and evaluation can fire a β-redex $(\lambda x.t)\ s$ only if the argument s is a closed value, *i.e.* a closed abstraction; and in the production $R\ v$ of right v-contexts, v is always a closed abstraction. In this dissertation we consider λ_{Plot} only for its Closed CbV setting; nonetheless, in accordance to the original definition given by Plotkin (1975), we still say that a value is either a variable or an abstraction. This choice is mainly for technical reasons, because for example in section 7.3 we will treat variables and abstractions in the same manner. Whether variables should or

should not be considered as values may seem a pointless question, but its impact propagates from the design of a calculus up to the complexity of evaluation: for a thorough discussion of values *vs* variables, see Accattoli and Coen, 2014.

Properties. The key property of Plotkin's Closed CbV is called *harmony*, and reads as follows:

Proposition 3.1 \ Harmony of λ_{Plot}

A closed λ-term is β/v-normal if and only if it is a (closed) value *i.e.* a (closed) λ-abstraction.

Proof. Let t be a closed λ-term. We prove separately the two directions of the double implication:

($\Rightarrow$) Let us assume that t is β/v-normal. We proceed by induction on the structure of t:

- *Variable:* t cannot be a variable, because t is closed by hypothesis.
- *Abstraction:* if t is an abstraction, then we conclude trivially.
- *Application:* t cannot be an application. To prove it, let us assume that $t = su$, and derive a contradiction. Since t is by hypothesis closed and normal, then also s and u are closed and normal, and thus by *i.h.* we obtain that they both are closed abstractions. But then t cannot be normal, because β/v clearly is applicable with evaluation context $\langle\cdot\rangle$.

($\Leftarrow$) If t is an abstraction, then clearly it is β/v-normal by the definition of evaluation contexts for λ_{Plot}.

Harmony expresses a form of completeness of values with respect to β-reduction: every closed λ-term will either diverge, or the term will evaluate to an abstraction, meaning that every (weak) β-redex will eventually become a β/v-redex and be fired.

As we will see in the next section, harmony is broken when allowing open λ-terms: for instance, the term zz is open and clearly in β-normal form, but it is not a value. This causes some issues which will discuss in upcoming section 3.2.

Before turning to the next section, we prove a bunch of properties of λ_{Plot} that will come handy in later sections — the reader can skip them at a first read.

The first property is that v-contexts are closed under composition:

Proposition 3.2 \ Composition of right v-contexts

Let R and R' be right v-contexts. Then their composition $R\langle R'\rangle$ is a right v-context.

Proof. By induction on the structure of the right v-context R. Cases:

- *Hole, i.e.* $R := \langle\cdot\rangle$: then, $R\langle R'\rangle = R'$ is a right v-context by hypothesis.
- *Right, i.e.* $R := tR''$: then, $R\langle R'\rangle = tR''\langle R'\rangle$ is a right v-context because $R''\langle R'\rangle$ is a right v-context by *i.h.*
- *Left, i.e.* $R := R''v$: then, $R\langle R'\rangle = R''\langle R'\rangle v$ is a right v-context because $R''\langle R'\rangle$ is a right v-context by *i.h.*

The following proposition shows that values are closed under substitition: this is a neat property of λ_{Plot} that unfortunately is not preserved in the open case (see page 36).

Proposition 3.3 \ Closure under substitution

Let v and v' be values. Then $v\{x\leftarrow v'\}$ is a value. If moreover v is a λ-abstraction, then $v\{x\leftarrow v'\}$ is so.

Proof. Since v is a value, there are only two cases:

- *Variable, i.e.* either $v := x$ and then $v\{x\leftarrow v'\} = v'$ which is a value by hypothesis; or $v := y \neq x$ and then $v\{x\leftarrow v'\} = y$ which is a value.
- *Abstraction, i.e.* $v := \lambda y.s$ and we can suppose without loss of generality that $y \notin \mathsf{fv}(v) \cup \{x\}$; therefore, $v\{x\leftarrow v'\} = \lambda y.(s\{x\leftarrow v'\})$ which is a λ-abstraction and hence a value.

Another neat property that will not be preserved in the open case is that the substitution of values commutes with β/v reduction:

Proposition 3.4 \ Value substitutions and evaluation commute

Let t and s be λ-terms, and v be a value. If $t \to_{\beta/v} s$ then $t\{x\leftarrow v\} \to_{\beta/v} s\{x\leftarrow v\}$.

Proof. By induction on the definition of $t \to_{\beta/v} s$. Cases:

- *Root-step, i.e.* $t := (\lambda y.u)v' \mapsto_{\beta/v} u\{y\leftarrow v'\} =: s$ and we can suppose without loss of generality that $y \notin \mathsf{fv}(v) \cup \{x\}$. According to proposition 3.3, $v'\{x\leftarrow v\}$ is a value. As a consequence, $t\{x\leftarrow v\} = (\lambda y.u\{x\leftarrow v\})(v'\{x\leftarrow v\}) \to_{\beta/v} u\{y\leftarrow v'\{x\leftarrow v\}\} = u\{y\leftarrow v'\}\{x\leftarrow v\} = s\{x\leftarrow v\}$.
- *Application right, i.e.* $t := ur \to_{\beta/v} up =: s$ with $r \to_{\beta/v} p$; by *i.h.* $r\{x\leftarrow v\} \to_{\beta/v} p\{x\leftarrow v\}$, and therefore $t\{x\leftarrow v\} = u\{x\leftarrow v\}(r\{x\leftarrow v\}) \to_{\beta/v} u\{x\leftarrow v\}(p\{x\leftarrow v\}) = s\{x\leftarrow v\}$.
- *Application left, i.e.* $t := uv' \to_{\beta/v} rv' =: s$ with $u \to_{\beta/v} r$; by *i.h.*, $u\{x\leftarrow v\} \to_{\beta/v} r\{x\leftarrow v\}$; according to proposition 3.3, $v'\{x\leftarrow v\}$ is a value

and hence $t\{x\leftarrow v\} = u\{x\leftarrow v\}(v'\{x\leftarrow v\}) \to_{\beta/v} r\{x\leftarrow v\}(v'\{x\leftarrow v\}) = s\{x\leftarrow v\}$.

3.2 Open CbV

As mentioned in the previous section, the harmony of CbV is lost on open λ-terms: open normal forms are not necessarily values, for instance the term zz. As a consequence, there may be stuck β-redexes like $(\lambda y.t)\ (zz)$ that will never be fired by $\to_{\beta/v}$ because their argument is normal but it is not a value. The weakness of β/v-reduction is widely recognized by the literature, and it manifests in various ways: we will present them in the following paragraphs, and only later introduce the well-behaved Open CbV calculus that we use in this dissertation.

The first issue of λ_{Plot} with open λ-terms is of a syntactical nature: as already noticed by Plotkin himself, the transformation of open CbV λ-terms into continuation-passing style (CPS) is incomplete. We will discuss further this point in section 7.4.

Secondly, stuck redexes can affect termination, impacting on the notion of *observational equivalence* (a notion of equivalence that identifies terms exhibiting the same converge behaviour). Consider for example the following term:

Example 3.5 \ Stuck β/v-redex

$$(\lambda y.\delta)\ (zz)\ \delta \quad \text{where} \quad \delta := \lambda x.xx.$$

Under CbN, the term diverges:

$$\underline{(\lambda y.\delta)\ (zz)}\ \delta \to_\beta \delta\delta \to_\beta \delta\delta \to_\beta \cdots$$

but in CbV it is in normal form, and as we explain below, one would expect it to behave like the divergent term $\Omega := \delta\delta$ also in CbV.

Solvability. Another issue with open terms is evident in the notion of *solvability*. Historically, solvability was introduced in connection with the definability in the λ-calculus of partial recursive functions, but actually it is a pervasive tool in the semantic analysis of the λ-calculus: it underlies the notions of approximants, Böhm trees, separability, and sensible λ-teories.

Definition 3.6 \ Solvable term

A λ-term t is said *solvable* if there exists a head context H such that

$$H\langle t\rangle \quad \to_\beta^* \quad I$$

where I is the identity, and head contexts are defined by the following grammar:[5]

$$H ::= \langle\cdot\rangle \mid H\ t \mid \lambda x.H.$$

Terms that are not solvable are called unsolvable (duh!), and they can be used to adequately represent in the λ-calculus the everywhere undefined function (Wadsworth, 1976).

To study CbV-solvability, one can restrict the reduction relation $\to_\beta$ used in definition 3.6 to the one of λ_{Plot}, *i.e.* $\to_{\beta/v}$: if λ_{Plot} were well-behaved, then it should enjoy an *internal* operational characterization of CbV-solvability. But this is not the case: let us consider for instance as t the stuck term from example 3.5. Whatever the context H, the term $H\langle t\rangle$ either diverges or it remains stuck, hence t is by definition unsolvable, even though it is not divergent. This lack of an internal characterization of CbV-solvability shows an inherent weakness in the rewriting of λ_{Plot}.

Adequacy. A further confirmation comes from *denotational semantics*. In that approach, each λ-term is assigned a denotation, which is a syntax-free mathematical interpretations of that term. Well-behaved denotations intuitively guarantee that a calculus is independent from its own syntax, showing that it is not ad-hoc (Accattoli and Guerrieri, 2018). A fundamental property of denotations is *adequacy*, which means that denotations correctly characterize normalizable terms. As first noticed by Ronchi Della Rocca and Paolini, 2004, λ_{Plot} is not adequate with respect to *relational semantics*, a well-known denotational semantics coming from the linear logic interpretation of CbV (Ehrhard, 2012).

In order to fix the issues sketched above, the literature has explored and deeply studied different presentations of open CbV, plus their rewriting, cost models, abstract machines, and denotational semantics (Accattoli and Coen, 2015; Accattoli and Guerrieri, 2016; Accattoli and Guerrieri, 2017; Accattoli and Guerrieri, 2018).

For instance, various calculi have been proposed to solve the issue of premature CbV normal forms discussed above; one solution (Carraro and Guerrieri, 2014) extends the calculus with shuffling rules, which commute stuck redexes in order to create new viable redexes. Let us go back to the stuck term of example 3.5: in the shuffling calculus, the rightmost δ is commuted inside the leftmost abstraction, and then the term does diverge because reduction is allowed inside the body of an applied abstraction:

$$\underline{(\lambda y.\delta)}\ (zz)\ \underline{\delta} \to_{\mathsf{shuffle}} (\lambda y.\underline{\delta\delta})\ (zz) \to_\beta (\lambda y.\underline{\delta\delta})\ (zz) \to_\beta \cdots$$

Other solutions extend β/v-reduction with other rewriting rules or constructors, or even go as far as changing the applicative structure of terms, as in the sequent calculus approach of Curien and Herbelin, 2000. Accattoli and Guerrieri, 2016 compared different incarnations of open CbV, proving that they are all equivalent from a termination point of view and hence justifying the existence of a unique notion of Open CbV. We present Open CbV in the next section.

[5] These head contexts are slightly more general than the ones introduced on page 26, which were weak: here their grammar also includes the production "$\lambda x.H$".

The fireball calculus. In this section we introduce the fireball calculus λ_{fire}, the simplest presentation of Open CbV. The fireball calculus was introduced without a name and studied first by Paolini and Ronchi Della Rocca, 1999; Ronchi Della Rocca and Paolini, 2004. It has then been rediscovered by Grégoire and Leroy, 2002 to improve the implementation of Coq, and later by Accattoli and Coen, 2015 to study cost models, where it was also named. We present it following Accattoli and Coen, 2015, changing only inessential, cosmetic details.

Syntax

$$\begin{array}{rrcl} \text{Fireballs} & f & ::= & v \mid i \\ \text{Inerts} & i & ::= & x\ f_1 \cdots f_n \quad (n > 0) \\ \text{Right f-contexts} & R & ::= & \langle\cdot\rangle \mid t\ R \mid R\ f \end{array}$$

Rules at top-level

$$\begin{array}{lll} (\lambda x.t)\ v & \mapsto_{\beta/v} & t\{x \leftarrow v\} \\ (\lambda x.t)\ i & \mapsto_{\beta/i} & t\{x \leftarrow i\} \end{array}$$

Contextual closure

$$\begin{array}{lll} R\langle t\rangle & \to_{\beta/v} & R\langle s\rangle \quad \text{if } t \mapsto_{\beta/v} s \\ R\langle t\rangle & \to_{\beta/i} & R\langle s\rangle \quad \text{if } t \mapsto_{\beta/i} s \end{array}$$

Reduction

$$\to_{\beta/f} \ := \ \to_{\beta/v} \cup \to_{\beta/i}$$

The fireball calculus λ_{fire} / **Figure 3.2**

The fireball calculus λ_{fire} is defined in fig. 3.2. The idea is to avoid stuck λ-terms by generalizing the values of the CbV λ-calculus—*i.e.* abstractions and variables— to fireballs[6], by extending variables to more general "inert" λ-terms. Fireballs (noted f) and inerts (noted i) are defined by mutual induction (see fig. 3.2). For instance, x and $\lambda x.y$ are fireballs as values, while $y\ (\lambda x.x)$, xy, and $z\ (\lambda x.x)\ (zz)\ (\lambda y.zy)$ are fireballs as inert λ-terms.

Syntax. The main feature of inert λ-terms is that they are open, normal, and that when plugged in a context they cannot create new redexes, hence the name. Essentially, they are the "neutral terms" of Open CbV. In the presentation of Grégoire and Leroy, 2002, inert λ-terms are called "accumulators" and fireballs are simply called values. Variables are, morally, both values and inert λ-terms. In Accattoli and Coen, 2015 they were considered as inert λ-terms, while here, for minor technical reasons we prefer to consider them as values and not as inert λ-terms.

[6] "Fireball" is a pun on "fire-able", *i.e.* an open, weak normal form, that can be fired by CbV.

Evaluation. Evaluation is given by the "call-by-fireball" reduction $\to_{\beta/f}$: the β-rule can fire, "lighting" the argument, only if the argument is a fireball. We actually distinguish two subrules: one that "lights" values, noted $\to_{\beta/v}$, and one that lights inert λ-terms, noted $\to_{\beta/i}$ (see fig. 3.2).

We endow the calculus with the right-to-left evaluation strategy, defined via right evaluation f-contexts R (which we denote, by a slight abuse of notation, with the same letter R as the right v-contexts introduced in the previous section). A more general calculus is defined in Accattoli and Guerrieri, 2016, for which both left-to-right and right-to-left strategies are shown to be complete. We omit details about the rewriting theory of the fireball calculus because our focus here is on evaluation.

Let us have a look at an example of evaluation sequence in λ_{fire}:

Example 3.7 \ λ_{fire} evaluation

Let $t := (\lambda z.z\ (y\ z))\ (\lambda x.x)$. Then:

$$t \to_{\beta/f} (\lambda x.x)\ (y\ (\lambda x.x)) \to_{\beta/f} y\ (\lambda x.x)$$

where the final term $y\ (\lambda x.x)$ is a fireball (and β/f-normal).

Properties. As discussed in section 3.1, Closed CbV enjoys harmony, *i.e.* a closed λ-term t is β/v-normal if and only if it is an abstraction. The fireball calculus λ_{fire} satisfies an analogous property in the (more general) open setting by replacing abstractions with fireballs (proposition 3.8/1). Moreover, the fireball calculus is a conservative extension of Closed CbV: on closed terms it collapses on Closed CbV (proposition 3.8/2). No other presentation of Open CbV mentioned above has these good properties.

Proposition 3.8 \ Properties of λ_{fire}

Let t, s be λ-terms:

1. *Open harmony*: t is β/f-normal if and only if t is a fireball.
2. *Conservative open extension*: if t is closed, $t \to_{\beta/f} s$ if and only if $t \to_{\beta/v} s$.

Proof. In Accattoli and Guerrieri, 2018.

In the rest of this section we provide a few properties of λ_{fire} that the reader can skip at a first read. In the next chapter, we will discuss how to implement evaluation, first in the general case and then according to (Open) CbV.

The following proposition summarizes the key property of inert λ-terms: substitution of

inert λ-terms does not create or erase β/f-redexes, and hence can always be avoided when implementing β/f.

Proposition 3.9 \ Inert substitutions and evaluation commute

Let t, s be λ-terms, and i be an inert λ-term. Then:

$$t \to_{\beta/f} s \quad \text{if and only if} \quad t\{x \leftarrow i\} \to_{\beta/f} s\{x \leftarrow i\}.$$

Proof. In Accattoli and Guerrieri, 2018.

Note that with general λ-terms (or even fireballs) instead of inert ones, evaluation and substitution do not commute, in the sense that both directions of proposition 3.9 do not hold:

- Direction $\Leftarrow$ is false because a substitution can create β/f-redexes, as in
$$(xy)\{x \leftarrow \lambda z.z\} = (\lambda z.z)\, y.$$
Note that, for the same reason, direction $\Leftarrow$ is false also when considering β-reduction $\to_\beta$ instead of $\to_{\beta/f}$ and general λ-terms instead of inert ones.
- Direction $\Rightarrow$ is false because a substitution can erase β/f-redexes, as in
$$((\lambda x.z)\,(xx))\{x \leftarrow \delta\} = (\lambda x.z)\,(\delta\delta)$$
where $\delta := \lambda y.yy$.

These counterexamples rely on the fact that fireballs are not closed under substitution (unlike values, see proposition 3.3): the substitution of a value (which is a fireball) for a variable in an inert λ-term (which is a fireball) may yield a λ-term that is not a fireball. However, the following weaker variant of closure under substitution holds:

Proposition 3.10 \ Semi-closure under substitution

Let v be a λ-value, $x_1, \dots, x_n$ be pairwise distinct variables, and $f_1, \dots, f_n$ be fireballs. Then $v\{x_1 \leftarrow f_1, \dots, x_n \leftarrow f_n\}$ is a fireball. If moreover v is a λ-abstraction, then $v\{x_1 \leftarrow f_1, \dots, x_n \leftarrow f_n\}$ is a λ-abstraction.

Proof. Let $\sigma := \{x_1 \leftarrow f_1, \dots, x_n \leftarrow f_n\}$. Since v is a value, there are only two cases:

- *Variable, i.e.* either $v := x_i$ for some $1 \le i \le n$ and then $v\sigma = f_i$ which is a fireball by hypothesis; or $v := y \neq x_i$ for all $1 \le i \le n$ and then $v\sigma = y$ which is a fireball.
- *Abstraction, i.e.* $v := \lambda y.s$ and we can suppose without loss of generality that $y \notin \bigcup_{i=1}^{n} \mathsf{fv}(f_i) \cup \{x_1, \dots, x_n\}$; therefore, $v\sigma = \lambda y.(s\sigma)$ which is a λ-abstraction and hence a fireball.

As a consequence of semi-closure, we prove that β/v reductions are mapped by fireball substitutions to β/f reductions. This is a property that our evaluation machines in part II will exploit in a crucial way.

Proposition 3.11 \ Substitution

Let t and s be λ-terms, $x_1, \dots, x_n$ pairwise distinct variables, and $f_1, \dots, f_n$ be fireballs. If $t \to_{\beta/v} s$ then $t\{x_1 \leftarrow f_1, \dots, x_n \leftarrow f_n\} \to_{\beta/f} s\{x_1 \leftarrow f_1, \dots, x_n \leftarrow f_n\}$.

Proof. Let $\sigma := \{x_1 \leftarrow f_1, \dots, x_n \leftarrow f_n\}$. By induction on the definition of $t \to_{\beta/v} s$. Cases:

- *Root-step, i.e.* $t := (\lambda y.u)v' \mapsto_{\beta/v} u\{y \leftarrow v'\} =: s$ and we can suppose without loss of generality that $y \notin \bigcup_{i=1} \mathsf{fv}(f_i) \cup \{x_1, \dots, x_n\}$. According to proposition 3.10, $v'\sigma$ is a fireball. As a consequence, $t\sigma = (\lambda y.u\sigma)(v'\sigma) \to_{\beta/f} u\sigma\{y \leftarrow v'\sigma\} = u\{y \leftarrow v'\}\sigma = s\sigma$.
- *Application right, i.e.* $t := ur \to_{\beta/v} up =: s$ with $r \to_{\beta/v} p$; by *i.h.* $r\sigma \to_{\beta/f} p\sigma$, and therefore $t\sigma = u\sigma(r\sigma) \to_{\beta/f} u\sigma(p\sigma) = s\sigma$.
- *Application left, i.e.* $t := uv' \to_{\beta/v} rv' =: s$ with $u \to_{\beta/v} r$; by *i.h.*, $u\sigma \to_{\beta/f} r\sigma$; according to proposition 3.10, $v'\sigma$ is a fireball and hence $t\sigma = u\sigma(v'\sigma) \to_{\beta/f} r\sigma(v'\sigma) = s\sigma$.

Moreover, it holds the following property of composition of f-contexts that corresponds to proposition 3.2 in λ_{Plot}:

Proposition 3.12 \ Composition of right f-contexts

Let R and R' be right f-contexts. Then their composition $R\langle R'\rangle$ is a right f-context.

Proof. By induction on the right f-context R. Cases:

- *Hole, i.e.* $R := \langle\cdot\rangle$: then, $R\langle R'\rangle = R'$ is a right f-context by hypothesis.
- *Right, i.e.* $R := t\ R''$: then, $R\langle R'\rangle = t\ R''\langle R'\rangle$ is a right f-context because $R''\langle R'\rangle$ is a right f-context by *i.h.*
- *Left, i.e.* $R := R''\ f$: then, $R\langle R'\rangle = R''\langle R'\rangle\ f$ is a right f-context because $R''\langle R'\rangle$ is a right f-context by *i.h.*

Chapter 4

Implementing Evaluation

Suppose we would like to implement β-reduction literally, exactly as defined through reduction semantics. The naïve way is to implement β-reduction as the iteration of single β-reduction steps, where each reduction step $t \to_\beta s$ actually consists of three distinct actions at meta-level (cf. the definition of reduction in fig. 2.1):

1. Unplugging: search the given term t for a redex by decomposing it into an evaluation context C and a redex t' such that $t = C\langle t' \rangle$.
2. Reduction: perform the β-step $t' \mapsto_\beta s'$.
3. Plugging: plug the reduced term s' back into the reduction context C, obtaining the term $C\langle s' \rangle = s$.

Let us have a look at the computational complexity of implementing each part:

1. Given a term t, searching for a redex requires to transverse t, which introduces an overhead proportional to the size of t.
2. Due to substitution, performing $t' \mapsto_\beta s'$ requires time proportional to the square of the size of t'.
3. Plugging back the reduced term requires time linear in the size of the context C.

As a consequence, implementing a sequence of n reduction steps $t \to_\beta^n s$ in the naïve way requires time proportional to $|t|^{2^n}$, *i.e.* exponential in the number of β-steps. This clearly collides with the success of functional programming languages, for which efficient interpreters and compilers have existed for a long time.

In this chapter, we will discuss how to reduce the implementation overhead caused by the three operations outlined above. Note that the naïve approach is inefficient for mainly two reasons:

- The first reason is the need to (un)plug: at each β-step, an intermediate term is constructed by plugging, and afterwards it is immediately decomposed by the subsequent unplugging. A solution to this problem is not to discard the evaluation context at

each step, instead transforming it into the next valid context according to the chosen strategy—this process can be understood in terms of the refocusing technique of Danvy and Nielsen, 2004. In section 4.3 we will introduce *abstract machines*, state-transitition machines used to evaluate λ-terms. A state of these machines includes the λ-term being evaluated plus auxiliary data structures, some of which encode the current evaluation context. The location of the current redex and which parts of the term have already been evaluated are useful information that facilitates the search for the next redex.

- The second reason, and the cause of the most dramatic inefficiency, is the implementation of β itself, or more precisely of substition. In section 4.1 we will show a phenomenon called "size explosion", caused by the literal implementation of substitution and explaining the exponential overhead. The solution to this issue is to implement substitution not as defined, but by recording the substitutions due to β-steps without actually carrying them, and then only performing them partially and on-demand. As we will see, evaluation machines usually accomplish this by keeping a data structure called "environment", where to store such delayed substitutions. The decomposition of β and the delay of substitution are the topic of section 4.2.

Implementation in three steps. We are now going to set off on a journey from calculus to implementation. This trip will walk us through increasing refinements of the reduction semantics, for which we will employ the following terminology:

- Calculus for a "small-step" semantics where both substitution and search for the redex are meta-level operations;
- Calculus with ES (section 4.2) for a "micro-step" semantics where substitution is decomposed through explicit substitutions, but the search for the redex is still meta-level and expressed via evaluation contexts;
- Abstract machine (section 4.3) for a micro-step semantics where both substitution and search for the redex are decomposed—search for redexes is handled via one or more data structures called *stack*, *dump*, *frame*, and so on. Also, the management of names is explicit, *i.e.* terms are not quotiented under α-equivalence, and renamings are performed manually.

Computational complexity. Finding what is the actual implementation cost of evaluation is essential to analyse the computational complexity of programs written in the λ-calculus. Note that the λ-calculus is Turing complete—*i.e.* it can encode any computable function—and it is quite convenient as a programming language thanks to its higher-order nature, which allows to specify algorithms in a quite abstract and succint way. For complexity analyses of algorithms, one needs to fix a cost model for the underlying programming language, *i.e.* an assignment of each language primitive to the actual work needed to effectively implement it.

The most natural cost model for functional programming languages is the "unitary" cost model, where the number of computation steps is taken as complexity measure. In the λ-calculus this translates to the number of β-steps, or better the number of R-steps for a fixed evaluation strategy $\to_{\mathsf{R}}$. Under this model, implementing a reduction sequence like

$$t \quad \to_{\mathsf{R}}^{n} \quad s$$

is supposed to have a cost that is proportional to the number n of R-steps. Unfortunately this is not possible in the classical λ-calculus: every legitimate strategy suffers from a problem called "size explosion", due to the representation of λ-terms itself. We discuss size explosion in the next section, and provide a solution in section 4.2.

4.1 Size Explosion

Let us set aside the problem of plugging and unplugging for a moment, and consider β-reduction alone. In this section we will show that the ordinary way of representing λ-terms—strings where substitution is basically textual replacement—suffers from a problem called "size explosion", for which the size of λ-terms may grow exponentially with the number of β-steps. This issue does not depend on any particular strategy of evaluation, hence it concerns the representation of λ-terms itself.

In order to do so, we define a family $\{t_n\}_{n\in\mathbb{N}}$ of λ-terms, indexed on natural numbers, such that each term t_n has size proportional to n, but its normal form has size exponential in n:

Example 4.1 \ Exploding family (Accattoli and Lago, 2014)

We define the family $\{t_n\}_{n\in\mathbb{N}}$ by induction on n:

$$\begin{array}{lcl} t_0 & \coloneqq & s \\ t_{n+1} & \coloneqq & (\lambda x.t_n)\ s \end{array}$$

where $s \coloneqq y\ x\ x$.

We also define the additional family $\{s_n\}_{n\in\mathbb{N}}$ of λ-terms, in such a way that s_n is the normal form of t_n for each $n \in \mathbb{N}$, as we will prove in proposition 4.2.

$$\begin{array}{rcl} s_0 & \coloneqq & s \\ s_{n+1} & \coloneqq & y\ s_n\ s_n \end{array}$$

Let us have a look on how the terms reduce:

$$\begin{array}{l} t_0 = s = s_0 \\ t_1 = (\lambda x.y\ x\ x)\ s \mapsto_\beta y\ s\ s = s_1 \\ t_2 = (\lambda x.t_1)\ s \mapsto_\beta ((\lambda x.t_0)\ (y\ x\ x))\{x \leftarrow s\} = (\lambda x.t_0)\ (y\ s\ s) \\ \qquad \mapsto_\beta t_0\{x \leftarrow y\ s\ s\} = s_2 \\ \vdots \end{array}$$

Apparently each term t_n reduces in n β-steps to the term s_n, which we prove formally in the following proposition together with other properties on the size of the terms:

Proposition 4.2 \ Properties of $\{t_n\}_n$ and $\{s_n\}_n$

Let $\{t_n\}_n$ and $\{s_n\}_n$ be the families defined above. Then for every $n \in \mathbb{N}$:

1. $t_n \mapsto_\beta^n s_n$
2. s_n is in β-normal form
3. $|t_n| = 4n + 2$
4. $|s_n| = 2^{n+2} - 2$

Proof. Points 2, 3, and 4 can be proved by easy induction on n; we turn to prove Point 1. The case $n = 0$ is trivial because $t_0 = s_0$. In order to prove the case $n > 0$, we actually prove the following more general statement: $(\lambda x.t_i)\ s_j \mapsto_\beta^{i+1} s_{i+j+1}$ for every $i, j \in \mathbb{N}$; then, since $t_{n+1} = (\lambda x.t_n)\ s_0$, it follows that $t_{n+1} \mapsto_\beta^{n+1} s_{n+0+1} = s_{n+1}$. We proceed by induction on i. Base case: $(\lambda x.t_0)\ s_j = (\lambda x.yxx)\ s_j \mapsto_\beta ys_js_j = s_{j+1}$. Inductive case: $(\lambda x.t_{i+1})\ s_j = (\lambda x.(\lambda x.t_i)\ s)\ s_j \mapsto_\beta ((\lambda x.t_i)\ s)\{x \leftarrow s_j\} = (\lambda x.t_i)\ (ys_js_j) = (\lambda x.t_i)\ s_{j+1}$, and since by *i.h.* $(\lambda x.t_i)\ s_{j+1} \mapsto_\beta^{i+1} s_{i+j+2}$ we conclude. ∎

Note that in proposition 4.2 we used full (top-level) β-reduction to reduce the terms of the exploding family; however, one can actually show that all usual evaluation strategies (like CbN or CbV) reduce t_n to the same normal form in n steps, therefore size explosion is independent of the chosen strategy. It follows that no efficient implementation of the λ-calculus can use the literal definition of λ-terms and substitution.

Let us now understand the cause of this phenomenon. Basically, explosion is caused by unnecessary duplication: the i-th β-step causes the substitution $\{x \leftarrow s_i\}$ which duplicates the term s_i but it is not really needed. In fact, the terms s_i are what we called inert terms, meaning that they cannot create new redexes when substituted in any term (cf. proposition 3.9). Instead of carrying each substitution, we could delay it obtaining:

$$t_n \quad \mapsto_\beta^n \quad s\underbrace{\{x \leftarrow s\} \cdots \{x \leftarrow s\}}_{n \text{ times}}$$

The idea to avoid size explosion is to reach a normal form without carrying out the substitutions that are not needed. In fact the expression $s\{x \leftarrow s\} \cdots \{x \leftarrow s\}$ can be seen as a compact way of representing the normal form s_n at meta-level, which only takes space proportional to n. In order to keep the representation of the output compact, common subterms like s must be shared *explicitly* along the evaluation process. Sharing is the topic of the next section.

4.2 Explicit Sharing

As discussed in the previous section, all tools based on the λ-calculus must use sharing to circumvent the issue of size explosion.

The simplest way to extend the λ-calculus with sharing is by adding to its syntax the well-known "`let`" construct. A `let`-expression has the form $\texttt{let}\ x = s\ \texttt{in}\ t$, reading "$t$ where x stands for s". `let`-expressions were used for the first time by Dana Scott in *LCF*, his deductive system for computable functions introduced in 1969, and have been used ever since in many functional programming languages as a mechanism for local definitions. A definitional mechanism is basically a way to shorten a term by recording once a subexpression that is used multiple times, which simply amounts to sharing. As is customary, we write $t[x \leftarrow s]$ instead of $\texttt{let}\ x = s\ \texttt{in}\ t$, and call it ES for explicit sharing, or explicit subsitution.

It is important not to confuse the two terms $t\{x \leftarrow s\}$ and $t[x \leftarrow s]$. The first one is the term obtained by substituting all the occurrences of x in t with s: this kind of substitution is the meta-level operation described in definition 2.4 through operations that are external to the language of the λ-calculus. The second one instead is a substitution internalized in the language by the new constructor $\bullet\,[\,\bullet \leftarrow \bullet\,]$, and handled by specific reduction rules which we will discuss below.

First of all, let us introduce formally the syntax of explicit substitutions:

λ-calculus with ES

$$t, s ::= x \mid \lambda x.\, t \mid t\ s \mid t[x \leftarrow s]$$

Consider for instance the term $(x\,x)[x \leftarrow I]$ where $I := \lambda y.\, y$. Here the variable x acts as a sharing point, and I is shared by means of the ES $[x \leftarrow I]$. The corresponding unshared λ-term is $I\ I$ where I is occurs twice as a subterm.

Thanks to ES, we can decompose β-reduction into more atomic steps, decoupling function call and substitution. As a matter of fact, λ-calculi with ES were first introduced exactly to bridge the gap between the λ-calculus and its implementations, or better between how substitution is defined in the λ-calculus and how evaluation machines actually perform it to avoid size explosion. We will relate ES and machines in upcoming section 4.3; in the rest of this section, we showcase different calculi with ES, obtaining a progressive decomposition of substitution.

The simplest decomposition splits β-reduction in the following way:

$$(\lambda x.t)s \quad \to_{\beta\texttt{ES}} \quad t[x \leftarrow s] \quad \to_{\texttt{sub}} \quad t\{x \leftarrow s\}.$$

The rule $\to_{\beta\texttt{ES}}$ performs the function call but delays the actual substitution by means of an ES. The rule $\to_{\texttt{sub}}$ then performs the corresponding meta-level substitution. The resulting calculus is called $\lambda\texttt{sub}$ and its reduction rules are in fig. 4.1.

Contexts

$$E ::= \langle\cdot\rangle \mid E[x{\leftarrow}t]$$
$$C ::= \langle\cdot\rangle \mid C\,t \mid t\,C \mid \lambda x.C \mid C[x{\leftarrow}t] \mid t[x{\leftarrow}C]$$

Top-level rules

$$E\langle\lambda x.t\rangle\, s \mapsto_{\beta\mathtt{ES}} E\langle t[x{\leftarrow}s]\rangle$$
$$t[x{\leftarrow}s] \mapsto_{\mathtt{sub}} t\{x{\leftarrow}s\}$$

Contextual closure

$$C\langle t\rangle \to_{\beta\mathtt{ES}} C\langle s\rangle \quad \text{if } t \mapsto_{\beta\mathtt{ES}} s$$
$$C\langle t\rangle \to_{\mathtt{sub}} C\langle s\rangle \quad \text{if } t \mapsto_{\mathtt{sub}} s$$

The $\lambda\mathtt{sub}$ calculus / **Figure 4.1**

As usual, we define the reductions of $\lambda\mathtt{sub}$ as the closure of some top-level rules under evaluation contexts, denoted by C in fig. 4.1: these C contexts are the most general contexts for terms with ES, and their grammar includes general contexts for classical λ-terms (see fig. 2.1) plus productions for reducing below and inside explicit subsitutions.

Note that the definition of $\mapsto_{\beta\mathtt{ES}}$ requires an additional kind of contexts, denoted by E for "environment context": these contexts allow to append a sequence of ES to a term, so that $\mapsto_{\beta\mathtt{ES}}$ is applicable even if $\lambda x.t$ and s are inerspersed with explicit substitutions. We say that $\mapsto_{\beta\mathtt{ES}}$ "acts at a distance".

It may seem that ES in $\lambda\mathtt{sub}$ are somewhat redundant, since $\lambda\mathtt{sub}$ introduces new explicit substitutions but then it just executes them in one shot. On the contrary, this calculus already (but partially) solves the size explosion problem outlined in section 4.1. Let us for example consider the term t_2 from the exploding family of example 4.1:

$$t_2 = (\lambda x.t_1)\, s \mapsto_{\beta\mathtt{ES}} t_1[x{\leftarrow}s] = (\lambda x.t_0)\, s\, [x{\leftarrow}s] \mapsto_{\beta\mathtt{ES}} t_0[x{\leftarrow}s][x{\leftarrow}s] = s[x{\leftarrow}s][x{\leftarrow}s].$$

Similarly, t_n reduces in $n \mapsto_{\beta\mathtt{ES}}$-steps to the term $s[x{\leftarrow}s]\cdots[x{\leftarrow}s]$, which is a compact representation of the normal form whose size is linear in n (go to page 163 for a discussion on succint representations). Even though the term $s[x{\leftarrow}s]\cdots[x{\leftarrow}s]$ is not in actual normal form — because we could perform the substitution of every ES $[x{\leftarrow}s]$ — performing these substitutions is *useless* in this case because they only remove sharing without creating any new redex.

$\lambda\mathtt{sub}$ is a valuable tool to study the λ-calculus: in fact, it has a crucial role in the proofs of some results on λ-calculus for which no other proof is known (see Accattoli and Kesner, 2012). However, traditional calculi with ES usually decompose further the rule $\to_{\mathtt{sub}}$, obtaining a whole set of reduction rules that provide a more fine-grained control over substitution (see Kesner, 2007 for a survey). In fact, the decomposition of substitution provided by $\lambda\mathtt{sub}$ is still insufficient to solve size explosion in a completely satisfactory way.

Linear substitutions. Consider for example the term $x\,y\,x\cdots x\,[x{\leftarrow}I]$. The variable x occurs at head position, thus we are forced to perform the substitution of I for x in order to expose the redex Iy that is present in its unfolding $Iy\,I\cdots I$. However, applying the rule $\to_{\mathtt{sub}}$ forces to substitute the identity many times, once for each occurrence of the variable x: this is clearly unnecessary, as it suffices to substitute the identity just for the one occurrence of x in head position.

A first refinement of the $\to_{\mathtt{sub}}$ rule is to make explicit substitutions act on only one occurrence of a variable at a time: the resulting reduction rule is called $\to_{\mathtt{lsub}}$ for "linear substitution". Now we can reduce the term above as follows:

$$x\,y\,x\cdots x\,[x{\leftarrow}I] \quad \to_{\mathtt{lsub}} \quad I\,y\,x\cdots x\,[x{\leftarrow}I] \quad \to_{\beta\mathtt{ES}} \quad z\,[z{\leftarrow}y]\,x\cdots x\,[x{\leftarrow}I].$$

As we can see, performing additional substitution steps starting from the rightmost term $z\,[z{\leftarrow}y]\,x\cdots x\,[x{\leftarrow}I]$ is useless, because its unfolding $y\,I\cdots I$ is already in normal form. The idea to avoid a substitution unless necessary to create a redex is the core of the notion of useful sharing (Accattoli and Lago, 2014), and it also motivates an important optimization for evaluation machines called *substituting abstractions on-demand*, which we will discuss on page 72.

The calculus obtained by replacing $\to_{\mathtt{sub}}$ with $\to_{\mathtt{lsub}}$ is called $\lambda\mathtt{lsub}$ or *linear substitution calculus*, and its reduction rules are in fig. 4.2.

Reduction rules

$$\begin{array}{lll} E\langle\lambda x.t\rangle\,s & \mapsto_{\beta\mathtt{ES}} & E\langle t[x{\leftarrow}s]\rangle \\ C\langle x\rangle[x{\leftarrow}t] & \mapsto_{\mathtt{lsub}} & C\langle t\rangle[x{\leftarrow}t] \end{array}$$

Contextual closure

$$\begin{array}{lll} C\langle t\rangle \to_{\beta\mathtt{ES}} C\langle s\rangle & \text{if} & t \mapsto_{\beta\mathtt{ES}} s \\ C\langle t\rangle \to_{\mathtt{lsub}} C\langle s\rangle & \text{if} & t \mapsto_{\mathtt{lsub}} s \end{array}$$

The $\lambda\mathtt{lsub}$ calculus / **Figure 4.2**

As one can see in fig. 4.2, also the definition of the the top-level rule $\mapsto_{\mathtt{lsub}}$ requires contexts, which are used to locate the exact variable occurrence that one desires to substitute.

Our main reasons for introducing $\lambda\mathtt{lsub}$ is that it can uniformly represent many different evaluation machines for the λ-calculus present in the literature, as we will see in the next section. $\lambda\mathtt{lsub}$ was introduced by Accattoli (2012) as a slight variation over a calculus proposed by Milner (2007), and as a simplification of the structural calculus $\lambda\mathsf{j}$ of Accattoli and Kesner (2010). Apart from being well-behaved as a rewriting system (like the λ-calculus, it has a residual system, Accattoli et al., 2014), its main feature is a tight relationship with graphical formalisms like linear logic proof nets (Accattoli, 2018). In fact, the reductions of $\lambda\mathtt{lsub}$ can be seen as cut-elimination steps in linear logic (Ariola, Bohannon, and Sabry, 2009; Accattoli, Barenbaum, and Mazza, 2014a): $\to_{\beta\mathtt{ES}}$ corresponds to the "multiplicative" case, and $\to_{\mathtt{lsub}}$ to the "exponential" one, see also page section 6.2.

Even though it will not be necessary in this dissertation, one can push the decomposition of substitution to extremes, by removing the action at a distance and adding explicit reduction rules to propagate the substitution inside the term, until variables are reached. The simplest way to proceed is by adding a new reduction rule for each case of definition 2.4: this yields the calculus $\lambda\mathsf{x}$, whose rules are in fig. 4.3.

$$
\begin{array}{rcll}
(\lambda x.t)\, s & \mapsto & t[x \leftarrow s] & \\
x[x \leftarrow s] & \mapsto & s & \\
y[x \leftarrow s] & \mapsto & y & \text{if } x \neq y \\
(t\, u)[x \leftarrow s] & \mapsto & (t[x \leftarrow s])\, (u[x \leftarrow s]) & \\
(\lambda y.t)[x \leftarrow s] & \mapsto & \lambda y.(t[x \leftarrow s]) & \text{if } x \neq y \text{ and } y \notin \mathsf{fv}(s)
\end{array}
$$

Reduction rules for $\lambda\mathsf{x}$ / **Figure 4.3**

$\lambda\mathsf{x}$ was introduced by Bloo and Rose, 1995; Rose, 1992, but the literature actually contains many slightly different calculi of explicit substitutions, some using de Bruijn notation (Abadi et al., 1991; Gonthier, Abadi, and Lévy, 1992; Ferreira, Kesner, and Puel, 1996) or level notation (Lescanne and Rouyer-Degli, 1995), some others using named variables like $\lambda\mathsf{x}$. The pioneer calculus with ES was $\lambda\sigma$, introduced by Abadi et al. (1991). Shortly after, a counterexample by Melliès (1995) showed that $\lambda\sigma$ behaved in an undesirable way, which sparked a quest for the ultimate well-behaved calculus of explicit substitutions (Di Cosmo and Kesner, 1997).

As a final remark, we would like to point out that ES do not only allow to decompose β-reduction, but have additional benefits. From an operational semantics point of view, ES allow elegant formulations of subtle strategies such as CbNeed evaluation—various presentations of CbNeed use ES (Wadsworth, 1971; Launchbury, 1993; Maraist, Odersky, and Wadler, 1998; Ariola and Felleisen, 1997; Sestoft, 1997a; Kutzner and Schmidt-Schauß, 1998) and a particularly simple one is by Accattoli, Barenbaum, and Mazza (2014a). From a rewriting point of view, ES enable proof techniques that are not available within the λ-calculus, see for instance (Accattoli, 2012). Also, from a logical point of view, ES are the proof terms corresponding to the extension of natural deduction with a cut rule (Barendregt and Ghilezan, 2000), and there is almost a one-to-one correspondence between cut-elimination rules and the reduction rules of calculi like $\lambda\mathsf{x}$.

Instead of being scattered all over a term, ES can be grouped together in finite sequences called environments. As we will see in the next section, abstract machines typically rely on such environments.

4.3 Abstract Machines

Abstract machines are first-order state transition systems used to model the implementation of programming languages. Their strength is to be abstract enough to leave out irrelevant

implementation details, but otherwise concrete enough to approximate the core of realistic run-time systems.

Abstract machines are a very convenient way to implement evaluation functions of λ-calculi (section 4.4) and study their implementation overhead (section 4.5). As we will see below, a state of such machines has various components, in which the term to be evaluated is decomposed in order to perform evaluation.

Evaluating a λ-term t through an abstract machine consists of the following process:

$$t \xhookrightarrow[\text{encoding}]{} \underline{t} =: \mathcal{S}_0 \xrightarrow[\text{execution}]{} \mathcal{S} \xrightarrow[\text{readback}]{} \mathcal{S}_{\downarrow}$$

First, one injects or "encodes" the λ-term t obtaining an initial state $\mathcal{S}_0$ of the machine; as we will see, injection usually consists in taking t as "code" and all other machine components as empty. Then, an execution ρ is a sequence of transitions of the machine $\mathcal{S}_0 \to^* \mathcal{S}$ from the initial state $\mathcal{S}_0$—we call $\mathcal{S}$ a *reachable* state. Finally, one can perform the readback $\mathcal{S}_{\downarrow}$ of a (final) state to a term: recall however that reading back to λ-terms is an exponential time operation (because of size explosion, as discussed in section 4.1), thus one is forced to preserve the sharing when reading back in actual implementations.

Definition 4.3 \ Abstract machine

An abstract machine `M` is a labeled transition system whose initial states are obtained from λ-terms by an encoding function $\underline{\bullet}$.

Early reduction machines, like the GMD by Berkling (1974), implemented λ-terms as strings and the operation of substitution in a literal way, leading to size explosion. Striving for more efficiency, the literature investigated mainly two classes of machines for the λ-calculus, according to what representation of λ-terms they operate on: graph reduction machines and environment machines.

- *Graph reduction machines* operate on λ-terms represented as graphs, and perform evaluation by rewriting the term graph, *i.e.* the program itself. As we will see in section 6.1, when λ-terms are represented as graphs, sharing amounts to having multiple edges that point to a same node: in this way, substitution steps can avoid to copy (the subgraph representing) the argument of a redex by simply reusing it as many times as the number of occurrences of the formal argument. Even though graph reduction is quite intuitive, efficient implementations are more complicated and less obviously correct (Sestoft, 1997b), see for example the *G-machine* (Kieburtz, 1985; Jones, 1992). On the other hand, graph reduction is well-suited to implement CbNeed evaluation: updating an evaluated argument is obtained by altering *imperatively* the graph structure.

- *Environment machines* keep the λ-term to be evaluated separate from the substitutions originated by β-steps: like with ES, substitutions are not performed right away but delayed and stored in a separate data structure called "environment" (also: store, heap), which is accessed as a kind of database (Curien, 1991).

In order to present the main intuitions underlying evaluation through abstract machines, in the rest of this section we showcase some well-known environment machines for closed evaluation: the KAM (page 48) and the MAM (page 50) for CbN evaluation, and the GLAM (page 54) for CbV evaluation. We will not present any machine for CbNeed, being slightly more involved and out of the scope of this dissertation.

The KAM

Most of the literature on environment machines —on which we focus in this dissertation— relies on so-called "local" environments, in which every piece of code in the machine is paired with its own environment, forming a "closure". A closure owes its name to the fact that all the free variables of the term in the closure are bound in its environment: this approach clearly works only for closed terms, *i.e.* computer programs. In this section we focus on the evaluation of λ-terms without free variables; only later, in chapter 5, we will generalize machines to open terms.

The *Krivine Abstract Machine* (Krivine, 2007) is probably the most famous abstract machine for the λ-calculus using local environments (see fig. 4.4).

Syntax

$$\begin{array}{rrcl} \text{Environments} & e & ::= & \epsilon \mid [x{\leftarrow}c]::e \\ \text{Closures} & c & ::= & \langle t, e\rangle \\ \text{Stack} & \pi & ::= & \epsilon \mid c::\pi \end{array}$$

State

$$\boxed{t \mid e \mid \pi}$$

Initial states

$$\underline{t} := \boxed{t \mid \epsilon \mid \epsilon}$$

Final states

$$\boxed{\lambda x.t \mid e \mid \epsilon}$$

Transitions

$$\begin{array}{lcll} \boxed{t\,s \mid e \mid \pi} & \to_{\text{App}} & \boxed{t \mid e \mid \langle s, e\rangle::\pi} & \\ \boxed{\lambda x.t \mid e \mid c::\pi} & \to_{\text{Abs}} & \boxed{t \mid [x{\leftarrow}c]::e \mid \pi} & \\ \boxed{x \mid e \mid \pi} & \to_{\text{Var}} & \boxed{t \mid e' \mid \pi} & \text{if } e(x) = \langle t, e'\rangle \end{array}$$

The Krivine Abstract Machine (KAM) / **Figure 4.4**

Syntax. A state of the KAM is composed of the term being evaluated (which we also call "code") plus its local environment (together forming a closure) and a data structure called "applicative stack" or simply "stack" which is simply a list of closures. We denote by ϵ the empty list, and by $::$ the *consing* operation (we will sometimes omit the symbol $::$ to lighten up the notation).

Execution. The stack component basically contains the arguments of the applications that are encountered by "App" transitions: when evaluating an application of the form ts, the argument s is stored in the stack together with the current local environment e, and the machine proceeds to evaluate t. An "Abs" transition basically performs a β-step, but delays the substitution in the environment just like $\to_{\beta\mathtt{ES}}$-reductions in $\lambda\mathtt{lsub}$ (see 45). Finally, a "Var" transition substitutes one occurrence of the variable x with the corresponding entry in the environment just like a $\to_{\mathtt{lsub}}$-reduction in $\lambda\mathtt{lsub}$. "Var" transitions need to perform the lookup $e(x)$ of x in the environment e; in order to define it, we first introduce the domain of e:

Definition 4.4 \ Domain of an environment

Let e be an enviroment. We define its domain $\mathsf{dom}(e)$ by induction on the structure of e:

$$\begin{array}{rcl} \mathsf{dom}(\epsilon) & \coloneqq & \{\} \\ \mathsf{dom}([x{\leftarrow}c]::e) & \coloneqq & \{x\} \cup \mathsf{dom}(e). \end{array}$$

Definition 4.5 \ Lookup $e(x)$

Let e be an environment, and $x \in \mathsf{dom}(e)$. We denote by $e(x)$ the lookup of x in e, defined as follows by induction on the structure of e:

$$e(x) \coloneqq \begin{cases} c & \text{if } e = [x{\leftarrow}c]::e' \\ e'(x) & \text{if } e = [y{\leftarrow}c]::e' \text{ for } x \neq y \end{cases}$$

If $x \notin \mathsf{dom}(e)$, we say that $e(x)$ is undefined.

Nameless. Usually local environments are employed in combination with nameless λ-terms, introduced in section 2.4: when variables do not carry a name, variables do not need to be recorded in environment entries, and therefore the syntactic category of ES collapses to the one of closures. Then an enviroment is a list of closures, and the operation of lookup in the environment for a variable $\#i$ consists in taking the i-th entry from the environment, considered as a list. For a study of closures over nameless terms, see the calculus of closures $\lambda\rho$ by Curien (1991).

Kinds of environments. Most of the literature on abstract machines relies on local environments, however other styles of environments exist: in a recent work, Accattoli and Barras

(2017) compare local, "global", and "split" environments.

- Global environments are used by a minority of works, and instead of coupling every term with its own environment (forming a closure), they employ a single environment that records the substitutions originated from all the pieces of code in the machine. In this dissertation we prefer global environments over local ones, mainly for two reasons: on the one hand, working with global environments is simpler because it avoids using closures; on the second hand, global environments provide the same kind of sharing as λ-graphs, a graphical representation of shared λ-terms that we will introduce in section 6.1 and that we will employ in part III.

- Local environments actually enable faster machines but only in the case relevant for programming languages (weak evaluation of closed terms), not in the general case of proof assistants (open/strong evaluation, section 5.2); in addition, local environments do not naturally account for sharing as it is needed for instance by CbNeed evaluation (Accattoli and Barras, 2017).

- Finally, "split" environments are a style that combines the best of the two approaches, but which we will not consider in this dissertation—we refer the reader to Accattoli and Barras, 2017 for a comparison of the three styles, both from complexity and implementations point of view.

The MAM

The KAM defined in fig. 4.4 employs local environments, but we are now going to consider a variant of the KAM with global environments; it is called *Milner Abstract Machine* (see Accattoli and Barras, 2017), and its definition is in fig. 4.5.

Syntax

$$\begin{array}{rrcl} \text{Environments} & e & ::= & \epsilon \mid [x \leftarrow t] :: e \\ \text{Stack} & \pi & ::= & \epsilon \mid \square t :: \pi \end{array}$$

State

$$\boxed{t \mid \pi \mid e}$$

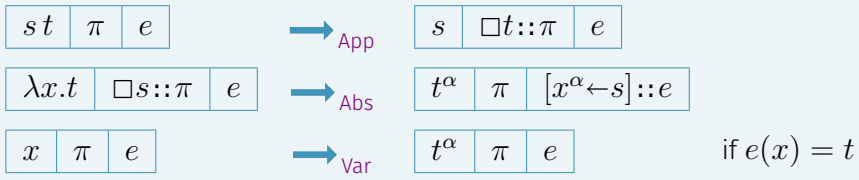
Initial states

$$\underline{t} := \boxed{t \mid \epsilon \mid \epsilon}$$

Final states

$$\boxed{\lambda x.t \mid \epsilon \mid e}$$

Transitions



The Milner Abstract Machine (MAM) / **Figure 4.5**

The main difference with respect to the KAM is that by using global environments the separate syntactical category of closures disappears, collapsing on terms. A state of the MAM has still three components: a λ-term, a stack π, and a global environment e. The global environment is simply a list of explicit substitutions: each ES in the environment impacts on both the first two components of the machine, and also recursively on the remaining part of the environment on its left. The stack is basically a list of terms, used to store the arguments encountered while evaluating a nested application term like in the KAM. However, in fig. 4.5 we defined the stack as a list of entries of the form "$\Box t$": following Danvy et al. (2010), we call these entries context frames, stack frames, or just frames.

Inside-out contexts. Since we will reuse the notion of frames in the abstract machines that follow, let us now provide a general definition of frames and stacks. The stack component in an abstract machine is a representation of the current evaluation context, but "turned inside-out like a returned glove".[1] To understand what an inside-out context is, let us consider for example the general context $D := t\,((\lambda x.\langle\cdot\rangle)\,s)$. D is built from the grammar of evaluation contexts by using the production rules tC, Cs, $\lambda x.C$, and finally $\langle\cdot\rangle$. Each use of a production rule corresponds to a layer of the context, and a frame is simply a single layer of an evaluation context. The grammar of context frames follows, with an entry for each production rule in the grammar of general contexts (cf. fig. 4.1):

Context frames

$$f \;::=\; \Box t \mid t\Box \mid \lambda x.\Box \mid \Box[x \leftarrow t] \mid t[x \leftarrow \Box]$$

An inside-out context—which we denote by $\mathcal{O}$—is then simply a list of context frames:

Inside-out contexts

$$\mathcal{O} \;::=\; \epsilon \mid f :: \mathcal{O}$$

We call these contexts "inside-out" because they lay out frames in reverse order with respect to a usual evaluation context. For instance, the inside-out context corresponding to D is $\lambda x.\Box :: \Box s :: t\Box :: \epsilon$.

MAM components. Let us now go back to the components of the MAM:

[1] A term together with an inside-out context is an example of the datatype known as *zipper* and due to Huet (1997).

- The stack is an inside-out context composed only of frames of the form $\square t$ and it is therefore the inside-out version of weak head contexts $H ::= \langle\cdot\rangle \mid H\,t$, correctly suggesting that the MAM simulates CbN evaluation.
- An environment instead is a list of ES, where each entry $[x{\leftarrow}t]$ is basically a frame of the form $\square[x{\leftarrow}t]$; hence an environment is the inside-out version of environment contexts E defined in fig. 4.1.

We can formalize the correspondence between classical contexts and inside-out contexts by defining the following readback functions:

Definition 4.6 \ Readback of inside-out contexts

We define the readback $\mathcal{O}_{\downarrow}$ of an inside-out context $\mathcal{O}$ to a usual context by using the auxiliary function $\mathrm{flip}(C \mid \mathcal{O})$, by induction on the structure of $\mathcal{O}$:

$$\begin{array}{lcl}
\mathcal{O}_{\downarrow} & := & \mathrm{flip}(\langle\cdot\rangle \mid \mathcal{O}) \\
\mathrm{flip}(C \mid \epsilon) & := & C \\
\mathrm{flip}(C \mid \square t :: \mathcal{O}) & := & \mathrm{flip}(Ct \mid \mathcal{O}) \\
\mathrm{flip}(C \mid t\square :: \mathcal{O}) & := & \mathrm{flip}(tC \mid \mathcal{O}) \\
\mathrm{flip}(C \mid \lambda x.\square :: \mathcal{O}) & := & \mathrm{flip}(\lambda x.C \mid \mathcal{O}) \\
\mathrm{flip}(C \mid \square[x{\leftarrow}t] :: \mathcal{O}) & := & \mathrm{flip}(C[x{\leftarrow}t] \mid \mathcal{O}) \\
\mathrm{flip}(C \mid t[x{\leftarrow}\square] :: \mathcal{O}) & := & \mathrm{flip}(t[x{\leftarrow}C] \mid \mathcal{O})
\end{array}$$

MAM decoding. We can now define the decoding of a MAM state to a term:

Definition 4.7 \ MAM decoding

Let $\mathcal{S} := (t, \pi, e)$ be a state of the MAM. We define its decoding $\mathcal{S}_{\downarrow}$ to a λ-term with ES as follows:

$$(t, \pi, e)_{\downarrow} := e_{\downarrow}\langle \pi_{\downarrow}\langle t\rangle\rangle.$$

As one can see, a state of the KAM corresponds to a term where the explicit substitutions are only found at top-level. One can also decode a MAM state to a λ-term without ES, by turning each explicit substitution in the environment to an actual substitution. In order to do so, we introduce the substitution induced by an environment:

Definition 4.8 \ Substitution of environment σ_e

We define the (simultaneous) substitution σ_e induced by an environment e, by induction over the structure of e:

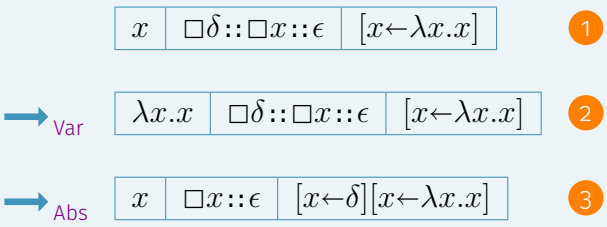
$$\begin{array}{lcl}
\sigma_{\epsilon} & := & \{\} \\
\sigma_{[x{\leftarrow}t]e} & := & \{x{\leftarrow}t\} \circ \sigma_e.
\end{array}$$

The decoding of a MAM state (t, π, e) to a λ-term is then simply $\pi_{\downarrow}\langle t\rangle\,\sigma_e$.

Defining a decoding function from machine states to λ-terms is the first step to prove that a given abstract machine simulates a strategy in a calculus; we will go into more details in section 4.4.

Variable hygiene. Another difference between local environments and global environments is the need for α-renaming. With local environments α-renaming is not necessary, because each term is enclosed in its own closure which binds the local variables in a unambiguous way. With global environments, we need to avoid the problem of *variable shadowing* by ensuring that variable names that are bound in the environment do not get reassigned to a different ES.[2] The following example shows an execution that does not perform α-renaming, leading to an incorrect result:

Example 4.9 \ Execution without α-renaming



The states 1 and 2 decode to the λ-term $I\delta I$ (having CbN normal form I), but the state 3 decodes to the diverging λ-term $\delta\delta$ because the ES $[x \leftarrow \delta]$ has shadowed $[x \leftarrow I]$.

While in the classical λ-calculus it is common to work implicitly modulo α-equality, for abstract machines we will not consider terms up to α, because different ways of handling α-equivalence characterize different approaches to abstract machines. In fact, α-renaming a term concretely amounts to performing a copy of the term, which is not a constant time operation as it requires time proportional to the size of the term that is being copied.

There are at least two alternative ways to enforce variable hygiene by performing α-renaming:

1. One way is to α-rename at each β transition, which is the only transition that appends a new ES in the environment: the variable that is being subsituted is refreshed in such a way that clearly there cannot be multiple ES in the environment with the same variable name. This is accomplished on the right-hand side of the "Abs" transition, where we denote by x^α a fresh variable with respect to the current state, and by t^α the term $t\{x \Leftarrow x^\alpha\}$ obtained by replacing each free occurrence of x with x^α. This option corresponds directly to the graph rewriting tradition, where the body of an abstraction is copied right during a β-rewrite step in order to avoid incorrectly altering all the shared instances of an abstraction (see page 77).

2. Another way is to follow Barendregt's variable convention, enforcing that all bound variables are always distinct. This can be accomplished by α-renaming at each substitution

[2] Local environments actually allow to avoid renamings, but the simplification is an illusion because then an added complexity pops up elsewhere—see Accattoli and Barras (2017).

step: please have a look at the right-hand side of the "Var" transition, where here we denote by t^α any term obtained from t by refreshing all its bound variables.

This option is more faithful to the λ-calculus tradition, but is also justified by the operational semantics provided by the interpretation of the λ-calculus in linear logic proof nets: in that setting, the action of copy is performed only during so-called "exponential" steps, that correspond directly to subsitution steps $\to_{\texttt{lsub}}$ in calculi with ES.

We denoted by $\bullet^\alpha$ the operation of renaming: note that it is not actually necessary to perform it in both transitions "Abs" and "Var" (see fig. 4.5), one can choose either one of the two according to the explanation above. In any case, it is fundamental to require that the variables in the domain of the environment of an abstract machine are all distinct, and it will be one of the invariants necessary to prove that a machine is correct:

Definition 4.10 \ Well-named environment

Let $e = [x_1 \leftarrow t_1] \cdots [x_k \leftarrow t_k]$ be an environment. We say that e is *well-named* if $x_i \neq x_j$ for all $i \neq j$.

In all the machines we consider in this dissertation, we will require the initial state to be well-named, and well-namedness will then be proved to be preserved under machine execution by a corresponding invariant.

The GLAM

Another well-known abstract machine is the *SECD* machine (SECD stands for Stack, Environment, Control, Dump) which was originally described by Landin, 1964, and can be considered an ancestor of the *Java Virtual Machine*. The SECD machine implements left-to-right CbV; an abstract machine which instead implements right-to-left CbV is the *Leroy Abstract Machine*, introduced in Accattoli, Barenbaum, and Mazza, 2014b and inspired by Leroy's ZINC Machine (Leroy, 1990). We present the Leroy Abstract Machine with global environments (GLAM) in fig. 4.6.

Syntax

$$\begin{array}{rrcl} \text{Environments} & e & ::= & \epsilon \mid [x \leftarrow v] :: e \\ \text{Stack} & \pi & ::= & \epsilon \mid t\Box :: \pi \mid \Box v :: \pi \end{array}$$

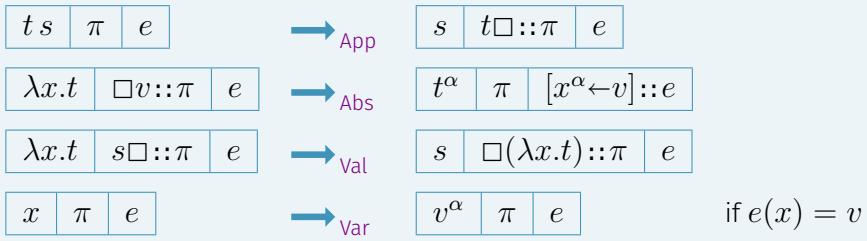
State

t	π	e

Initial state

$$\underline{t} := \begin{array}{|c|c|c|} \hline t & \epsilon & \epsilon \\ \hline \end{array}$$

Transitions



The Leroy Abstract Machine (GLAM) / **Figure 4.6**

The GLAM is usually presented in a slightly different way than how presented in fig. 4.6: the stack is usually split in two data structures, a "stack" and a "dump". Here instead we maintain the two components "merged" (in the terminology of Accattoli, Barenbaum, and Mazza, 2014b) and we reuse the notion of context frames defined on page 51.

Syntax. As usual, a state of the GLAM includes a code, a stack, and an environment. The first difference is that here an environment contains only *values*, because in CbV the arguments must be fully evaluated to values before a redex can fire. Note that here values collapse to abstractions, since we consider only closed evaluation.

It is easy to see that the GLAM stack is the inside-out version of a right v-context, defined on page 28 by the grammar:

$$R ::= \langle\cdot\rangle \mid tR \mid Rv.$$

This correctly suggests that the GLAM implements right-to-left CbV, *i.e.* the λ_{Plot} calculus defined in section 3.1. As done previously, we define the decoding of machine states as $(t, \pi, e)_\downarrow := e_\downarrow\langle\pi_\downarrow\langle t\rangle\rangle$.

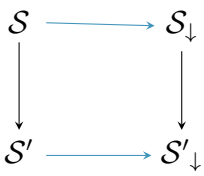
Execution. The "Abs" and "Var" transitions are exactly the same ones of the MAM (cf. fig. 4.5); the difference is in the slightly different "App" transition, and the new "Val" transition. The "App" transition focuses evaluation on the argument of an application (instead of focusing in head position like the MAM), because it evaluates arguments right-to-left. The "Val" transitions is applicable when the code — which originated from the right part of an application — has been evaluated to a value, and the machine can move on to evaluate the left part of that application.

That the GLAM correctly implements λ_{Plot} is proven for instance in Accattoli and Coen, 2015; such a proof requires the tools that we outline in the next section.

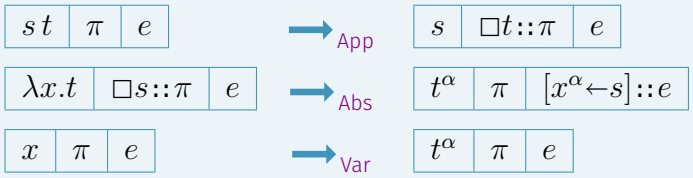
4.4 Simulation

To prove that a given abstract machine correctly implements a calculus, one first defines a decoding function from each machine state $\mathcal{S}$ to a term $\mathcal{S}_\downarrow$. The ideal would be that $\bullet_\downarrow$ is a

strong bisimulation between the machine and the calculus, in such a way that each transition of the machine corresponds to a reduction in the calculus, and viceversa:



Such a strong bisimulation almost never holds: the most evident mismatch is that not every machine transition corresponds to some reduction in the calculus. Let us consider for example the MAM from fig. 4.5, even though the same argument holds for the GLAM and the other machines in this dissertation. We recall the transitions of the MAM below in fig. 4.7:



Transitions of the MAM / **Figure 4.7**

The MAM, like more complex abstract machines, contains three kinds of transitions:

- search transitions, like the "App" transition of fig. 4.7: these transitions do not really alter the underlying term being evaluated, but only change the current evaluation context that is used to search the next redex;
- beta transitions, like the "Abs" transition of fig. 4.7: these transitions perform a β-step but delay the actual substitution by appending a new ES to the environment;
- substitution transitions, like the "Var" transition of fig. 4.7: these transitions replace exactly one occurrence of a variable with its value in the environment.

In this setting, it comes handy the distillation methodology of Accattoli, Barenbaum, and Mazza (2014a): while in a simulation every machine transition is required to be simulated by some steps in a calculus, in a distillation only some of the machine transitions are simulated, while the others are mapped to a *structural equivalence* $\equiv$ of the calculus.

Granularity. How to map machine transitions and what notion of structural equivalence to use mainly depend on the choice of decoding:

- *Decoding to λ-terms with ES.* In this case, the machine is distilled to some strategy in $\lambda\mathtt{lsub}$, where beta transitions are mapped to $\to_{\beta\mathtt{ES}}$ reductions (not exactly, but keep reading), substitution transitions are mapped to $\to_{\mathtt{lsub}}$ reductions, and search transitions are mapped to structural equivalence. Equivalent terms basically only differ

for the position of ES: the structural equivalence $\equiv$ on terms with ES amounts to the permutation of explicit substitutions around the term obtained by extending $=_\alpha$ with axioms like the following:[3]

$$\begin{array}{rcll} (t[x{\leftarrow}u])\, s & \equiv & (ts)[x{\leftarrow}u] & \text{if } x \notin \mathsf{fv}(s) \\ t\,(s[x{\leftarrow}u]) & \equiv & (ts)[x{\leftarrow}u] & \text{if } x \notin \mathsf{fv}(t) \end{array}$$

Permutating ES in a λ-term is required to reflect in the calculus the specific structure of terms that is enforced by an abstract machine during execution. As we have seen, the decoding of a machine state corresponds to a term where the ES are never nested and only found at top-level (*i.e.* grouped together to form the environment). Structural equivalence ensures that this canonical shape is reflected in the calculus.

- *Decoding to classical λ-terms.* In this case, the machine is distilled to some strategy in the λ-calculus without ES, where beta transitions are mapped to β-steps, and both search and substitution transitions mapped to equality, or more precisely α-equality $=_\alpha$.

Principal *vs* overhead transitions. We present the distillation technique in its more general form. To prove that a λ-calculus $\mathtt{C}$ "distills" an abstract machine $\mathtt{M}$ (or viceversa, that $\mathtt{M}$ implements $\mathtt{C}$), we first divide the transitions of the machine in two groups:

- *Principal transitions.* We call each reduction rule of the calculus $\mathtt{C}$ "principal", and we denote principal rules with the letter "p". We also assume that, for every p-rule of $\mathtt{C}$, $\mathtt{M}$ includes a transition rule corresponding to p (ideally, with the same label). Principal transitions are the ones that correspond to reductions in the calculus.
- *Overhead transitions:* the transitions that are not principal are called "overhead", and are denoted by the letter "o". These transition are the ones that will be mapped to structural equivalence during distillation.

We proceed with the formal definition of an implementation system, obtained by blending the notion of "reflective distillery" by Accattoli, Barenbaum, and Mazza (2014a) with the simpler definition of "implementation systems" by Accattoli and Guerrieri, 2017.

Definition 4.11 \ Implementation system

An implementation system $(\mathtt{M}, \mathtt{C}, \underline{\bullet}, \bullet_\downarrow)$ is given by a machine $\mathtt{M}$, a calculus $\mathtt{C}$, an encoding $\underline{\bullet}$ from λ-terms to machine states and a $\bullet_\downarrow$ decoding from machine states to $\mathtt{C}$-terms , such that:

1. *Initialization:* $\underline{t}_\downarrow = t$ for every λ-term t.
2. *Principal projection:* if $\mathcal{S} \to_{\mathsf{p}} \mathcal{S}'$, then $\mathcal{S}_\downarrow \to_{\mathsf{p}} \equiv \mathcal{S}'_\downarrow$ for every principal step p.
3. *Overhead transparency:* if $\mathcal{S} \to_{\mathsf{o}} \mathcal{S}'$, then $\mathcal{S}_\downarrow \equiv \mathcal{S}'_\downarrow$ for every overhead step o.

[3] See Figure 1 in Accattoli, Barenbaum, and Mazza, 2014a for a complete list of axioms for the structural equivalence.

4. *Determinism:* both M and C are deterministic.
5. *Equivalence:* $\equiv$ is a strong bisimulation with respect to the reduction rules of C.
6. *Progress:* if $\mathcal{S}$ is reachable and $\mathcal{S}_{\downarrow} \to_{\mathsf{p}} t$ for some principal step p, then there exists a state $\mathcal{S}'$ such that $\mathcal{S} \to_{\mathsf{o}}^{*} \to_{\mathsf{p}} \mathcal{S}'$ and $\mathcal{S}'_{\downarrow} \equiv t$.
7. *Overhead termination:* $\to_{\mathsf{o}}$ terminates.

Let us explain briefly each requirement:

1. *Initialization* requires that the readback is a left inverse of encoding.
2. *Principal projection* requires that each principal transition of M is mapped to a principal step in C (possibly up to $\equiv$).
3. *Overhead transparency* requires that overhead transitions of M are mapped to the structural equivalence of C.
4. *Determinism* requires both that the transition function of M is deterministic, and that reduction in C is a deterministic strategy.
5. *Equivalence* requires structural equivalence to commute with evaluation: this basically means that equivalence can be postponed until a normal form is reached, and then performed without compromising the result of evaluation.
6. *Progress* basically requires that M final states (to which no transition apply) decode to normal terms of C.
7. Finally, *Overhead termination* requires that M cannot get stuck on an infinite sequence of overhead transitions.

If all the requirements for an implementation system hold, then the given machine correctly implements the given strategy in the calculus, as stated in the following implementation theorem:

Theorem 4.12 \ Machine implementation

If $(\mathsf{M}, \mathsf{C}, \underline{\bullet}, \bullet_{\downarrow})$ is an implementation system, then:

- *Executions to derivations:* for any M-execution $\rho \colon \underline{t} \to^{*} \mathcal{S}$ there is a C-derivation $d \colon t \to^{*} \equiv \mathcal{S}_{\downarrow}$.
- *Derivations to executions:* for any C-derivation $d \colon t \to^{*} u$ there is a M-execution $\rho \colon \underline{t} \to^{*} \mathcal{S}$ such that $\mathcal{S}_{\downarrow} \equiv u$.
- *Principal matching:* in both previous points, for every principal step p, the number $|d|$ of reductions in d is exactly the same as the number $|\rho|_{\mathsf{p}}$ of p-transitions in ρ.

Proof. See Accattoli, Barenbaum, and Mazza, 2014a. ∎

The first two points of theorem 4.12 state that to each machine execution corresponds a derivation in the calculus, and vicecersa. The third point, instead, provides also some *quantitative* information: it says that the number of principal steps is preserved by distillation. This is a fundamental property for complexity analyses, which will be essential in the next section that discusses the computational complexity of execution.

Before turning to the next section, we point out that we will use the machinery of distillation and the implementation theorem in part II, in order to show that our "crumbling" abstract machines implement respectively λ_{Plot} (theorem 8.16) and λ_{fire} (theorem 9.16).

4.5 Efficiency

Abstract machines for the λ-calculus usually fall into the category of interpreters for programming languages. Following Danvy, 2003a we can distinguish further between two classes of abstract machines, according to the representation of λ-terms that they operate on:

- Some machines, like Krivine's machine (Krivine, 2007; Crégut, 1990), the CEK machine (Felleisen and Friedman, 1986; Flanagan et al., 1993), and the SECD machine (Landin, 1964) operate directly on the source syntax of λ-terms.

- Other machines, like the Categorical Abstract Machine (Cousineau, Curien, and Mauny, 1985) and the ZINC abstract machine (Leroy, 1990; Grégoire and Leroy, 2002), do not operate on λ-terms but have a different "instruction set", and thus require a non-trivial translation procedure (sometimes also called compilation). We call them "virtual machines".

In principle, choosing between an abstract or a virtual machine does not impact directly on the efficiency of the implementation, but more on the elegance and the clarity of the design. To study the efficiency of any implementation, one has to take into account mainly two factors:

1. The overhead of the abstract machine itself, *i.e.* the number (and cost) of evaluation steps it takes the machine to evaluate a term t *vs* the number of evaluation steps it takes to evaluate t in the λ-calculus.

2. How the abstract machine is actually mapped onto real hardware.

We will discuss Point 1 below, in the subsection about asymptotic complexity. Point 2 is much more intricate, as it forces to consider low-level details which may also depend on the computer architecture, for instance memory allocation and management *i.e.* manipulating the stack/heap, garbage collection, how to perform functions calls, *etc.*

One way to mitigate the complications behind Point 2 is to implement the abstract machine directly as an interpreter, itself written in a high-level programming language. This is the case for interactive theorem provers based on dependent types, which are always written in functional languages themselves: abstract machines for proof assistants benefit from a more high-level specification because they are much more complicated than the ones for functional programming languages, as they require strong evaluation (see chapter 5). In this way, the machine can rely on the features of the underlying language: for instance—as far as we know—no abstract machine implemented in an interactive prover performs garbage collection explicitly. In addition, being a first-order transition system, a machine can be implemented by a *tail-recursive* function, which ensures that the space available to it is not limited by the small size allocated for the execution stack, but by the whole size of the memory assigned to the process.

On the contrary, functional programming languages (which only handle the case where terms are closed and evaluation is weak) provide also compilation to machine language, typically more efficient. Compilation however requires handling many additional low-level details (Leroy, 1997; Jones, 1992), like how closures and datatypes are represented in memory, tagging (attaching type information to data, required by type polymorphism), boxing (heap-allocating complex data and handling it through a pointer), tail-call optimization (to save space in the stack), and so on. In addition, the compilers of functional programming languages are very complex pieces of software, whose optimizations may distort efficiency analyses. In this case benchmarks are fundamental to ensure that the many aspects involved interact properly.

In this dissertation we will not discuss efficiency with respect to Point 2 above: we consider machines only from an abstract point of view, and study their inherent efficiency by bounding asymptotically the number and cost of their transitions with respect to other abstract models of computation like *Random Access Machines* (RAMs). Asymptotic complexity is meant to complement the use of benchmarking, by covering all possible cases, that certainly cannot be covered via benchmarking. A downside is that complexity is discussed via the big O notation, which ignores constant factors, but hidden constants do impact performance in practice.

The Asymptotic Study of Abstract Machines

Strangely, asymptotic bounds of abstract machines are an aspect largely neglected by the literature—before Accattoli, Barenbaum, and Mazza (2014a) we are aware of only two independent works on this topic, one by Blelloch and Greiner (1995) and the other by Sands, Gustavsson, and Moran (2002). Bounding the overhead of abstract machines has then become commonplace; according to this line of research, one should prove the machine overhead to be polynomial or even linear in the number of β-steps (Accattoli, Barenbaum, and Mazza, 2014a; Accattoli, Barenbaum, and Mazza, 2015; Accattoli and Coen, 2015; Accattoli, 2016; Accattoli and Guerrieri, 2017; Accattoli and Barras, 2017).

To begin with, an useful intermediate setting for complexity analyses is the linear substitution calculus $\lambda\mathtt{lsub}$ presented in section 4.2: by distilling an abstract machine to $\lambda\mathtt{lsub}$

one can ensure that search transitions do not actually affect the complexity of evaluation (Accattoli, Barenbaum, and Mazza, 2014a).

In this dissertation, however, we relate abstract machines directly to the λ-calculus without ES (even though distilling will be necessary in our future work presented in section 15.1). By bounding the number (and the cost) of substitution and search transitions, *i.e.* "overhead" transitions, we show that they do not actually affect the overall complexity of evaluation. As a consequence, we will be able to bound the complexity of evaluation as a function of β-steps only (and the size of the initial term). As a side result, by the implementation theorem one also obtains that the number of β-steps according to the simulated strategy is a *reasonable* cost model, as explained in the next paragraph.

Reasonable cost models. As already mentioned on page 40, in order to analyse the computational complexity of λ-terms one needs first to fix a cost model, and the most natural one for the λ-calculus is taking as the cost of evaluation the number of β-steps required to evaluate a term. Not every cost model is acceptable: the so-called *invariance thesis* states that, in order to be *reasonable*, a time cost model must be polynomially related to the the cost model of Turing machines. Whether the unitary cost model was invariant for the λ-calculus has been a long-time open problem, until Accattoli and Lago (2012) proved that it is indeed invariant with respect to the λ-calculus endowed with the Leftmost-Outermost strategy. We will not stress too much about invariance in this dissertation, because the it is well-known that the unitary cost model is invariant under (weak) CbV.

The recipe for complexity analyses. Following Accattoli and Guerrieri (2017), we provide the three steps necessary to estimate the asymptotic complexity of an abstract machine $\mathtt{M}$:

1. *Number of transitions.* First of all, one should bound the length of an execution ρ by bounding the number $|\rho|_{\mathsf{o}}$ of overhead transitions in it. This step splits into two subparts:
   - Substitution *vs* beta: bounding the number $|\rho|_{\mathsf{sub}}$ of substitution transitions in ρ using the number $|\rho|_{\beta}$ of β-transitions;
   - Search *vs* substitution: bounding the number $|\rho|_{\mathsf{src}}$ of search transitions in ρ using the size of the initial state and $|\rho|_{\mathsf{sub}}$ (and therefore the number of β-transitions, by the previous point).
2. *Cost of single transitions.* Secondly, one must bound the cost of concretely implementing a single transition of $\mathtt{M}$. In order to lay out costs, it is necessary to go beyond the abstractions level, and make some low-level assumptions on how codes and data structures (stack and environment) are represented concretely in the implementation. The usual assumption is that the abstract machine is itself implemented on a Random Access Machine (RAM), an abstract machine that is polynomially related to Turing Machines. Search transitions are meant to have constant cost, because they do not perform any copy. The cost of substitution and beta transitions instead depends

on which one of the two transitions actually carries out the α-renaming. In any case, their cost is at most linear in the size of the initial state thanks to a common invariant of abstract machines called subterm property, having the following shape:

Definition 4.13 \ Subterm invariant

Let $\rho : \mathcal{S}_0 \to^* \mathcal{S}$ be the execution of an abstract machine with global environments. That machine satisfies the subterm invariant if all abstractions that occur in $\mathcal{S}$ are subterms of the initial state $\mathcal{S}_0$ (possibly up to variable renaming).

The subterm property ensures that only subterms of the initial term are duplicated and substituted along an execution. The operation of copy is supposed to have cost proportional to the size of the term to be copied.

3. *Complexity of the overhead.* Lastly, one obtains the total bound by composing the previous two points, *i.e.* by taking the number of each kind of transition, multiplying it by the respective cost of implementing it, and summing over all kinds of transitions.

Low-level assumptions. Different assumptions on the low-level implementation of an abstract machine lead to different complexity analyses: Accattoli and Barras, 2017 compare various ways of concretely implementing environment-based abstract machines, also showing the impact on complexity.

A peculiarity of abstract machines based on global environments is that they blur the historical distinction between environment-based machines and graph machines; more precisely, they easily allow either one between a traditional environment-based implementation, and a graph-based implementation. We discuss the two options below, but we anticipate the overall resulting complexity: a graph-based implementation is faster than a literal one, because the latter has (at least) an additional logarithmic factor due to the lookup in the sequential environment.

1. *Traditional environments:*

   - *Environment.* The operations demanded to environments are to push (a new ES) and to access an entry (when performing the lookup). The most literal way to implement a global environment is by means of a linked list of ES: this however makes the cost of the lookup in the environment linear in the size of the environment, which needs to be traversed in order to look up the desired variable. Since the size of the environment is usually proportional to the number of β-steps (because beta transitions extend the environment) this choice produces an overall quadratic overhead.

     Actually, more efficient data structures to implement global environment exist, for instance balanced trees or random-access lists. However, they only reduce the linear factor to a logarithmic factor, resulting in a overall quasilionear time complexity (see Accattoli and Barras, 2017).

- *Variables.* A variable is implemented either as a name (for instance, a string) or as a natural number (a de Bruijn index).
- *Stack.* The only operations demanded to stacks are to push and to pop; therefore a simple linked list suffices, for which push and pop are $O(1)$ operations. This makes the cost of search transitions constant, as desired.

2. *Implementation on λ-graphs:*

   - *Variables.* A variable bound by an ES in the environment is implemented as an *indirection node*, *i.e.* a *memory location* containing the address of another memory location, in such a way that an environment entry $[x \leftarrow t]$ means that the location associated to x contains a pointer to (the encoding of) t.
   - *Environment.* An environment e is not implemented as a data structure *per se*, but is identified with the global, implicit mapping from memory locations to their contents. In this way, e can be accessed in time $O(1)$ by just following the reference given by the variable occurrence that is being evaluated, with no need to access e sequentially.
   - *Stack.* Similarly, the stack component of an abstract machine does not need to be implemented as a separate data structure either. Recall that the stack component records the current evaluation context, which can be thought of the syntax tree surrounding the term being evaluated, but turned inside-out. As already mentioned, the stack can be implemented by a zipper over the λ-graph (Huet, 1997), by reversing the parent-child pointers during the traversal. In this way, push and pop are still $O(1)$ operations, and can each be implemented by a single destructive update of a node in the graph.

   The implementation that we have just outlined is basically the same representation of λ-terms as λ-graphs that we provide in chapter 6, where sharing simply amounts to allowing a node (*i.e.* a memory location) to have multiple incoming edges. There is however one single mismatch, caused by the slight different representation of ES variables as indirections instead of as simple directed edges. This mismatch is visible both in our implementation (chapter 11) and in the implementations provided by Accattoli and Barras, 2017, and consists in the following additional requirements for variable nodes:

   (a) *Explicit sharing:* other than *free* or *bound*, variable nodes can also behave as "explicit sharing" or "indirection" nodes, *i.e.* nodes possessing a single directed edge with endnode the node to be shared.

   (b) *Shared:* variable nodes are the only kind of node that can be shared.

   (c) *Maximally shared:* all nodes corresponding to the occurrences of a same variable are merged together.

   Nevertheless, we still consider λ-graphs essentially the same representation induced by the implementation on RAMs that we have provided above, with the only mismatch

of explicit sharing nodes. When developing our theory of sharing equality in part III, we will not consider explicit sharing nodes; however, one can translate in linear time between that representation to λ-graphs (and viceversa) by collapsing these "variables-as-sharing" on their child, if they are the child of some other node. Our results could be adapted to this other approach, but at the price of more technical definitions.

Garbage collection. As a final note: abstract machines ignore on purpose many details of concrete implementations, such as garbage collection. A detailed study of garbage collection is an orthogonal topic, that is not usually performed by existing abstract machines. In particular, garbage collection requires at most polynomial time, if not linear, and so its omission does not hide harmful blowups in complexity.

Chapter 5

Strong Evaluation

By strong evaluation we mean evaluating a term to its full β-normal form, which we call here strong normal form. Terms in strong normal form are described by the entry t_s in the following grammar:

Definition 5.1 \ Strong normal forms

$$\begin{array}{rcl} t_s & ::= & \lambda x.t_s \mid i_s \\ i_s & ::= & x \mid i_s\ t_s \end{array}$$

In general, we call "strong" any strategy able to contract redexes which are located inside the bodies of abstractions, as opposed to weak startegies which are not (cf. chapter 3).

Usual programming languages do not perform strong evaluation: the body of a function is never evaluated before enough arguments are provided. However, strong evaluation is sometimes employed during compilation in order to optimise the compiled code.

Partial evaluation, for example, is a program transformation technique that specializes a program with respect to part of its input (Consel and Danvy, 1993). Partial evaluation is useful when a program contains many occurrences of a function that is invoked on the same "static" arguments (*i.e.* available at compile-time): the optimization then consists in constructing a specialized version of that function with fewer formal parameters. It follows that partial evaluation must evaluate a function given the actual value of some formal parameters but without evaluating the expressions that depend on data that is only available at run-time.

Note however that the kinds of strong reduction performed at compile time typically consider only "linear" redexes, *i.e.* redexes where the body of the function uses the provided argument exactly once: in the linear case, the act of reducing amounts to "permutating" the code without any duplication, making the optimization safe as there cannot be any size or time explosion.

Another setting which requires strong reduction is higher-order logic programming, for instance the λProlog language. λProlog[1] is an extension of traditional Prolog where terms

[1] http://www.lix.polytechnique.fr/~dale/lProlog/

are not simply first-order terms but λ-terms. In all kinds of Prolog, computation is initiated by running a query over the relations defined in a program; these relations are specified through rules called "clauses", and the Prolog engine decides what rule to apply by trying to unify the current goal with the head of all existing clauses, one by one. Classical Prolog uses first-order unification to check if a rule is applicable, but λProlog uses *pattern unification*[2], a subset of the very powerful (undecidable) higher-order unification. Any algorithm for pattern unification must not only check that given terms unify up to syntactical identity, but also whether they are somewhat the same λ-term, *i.e.* are convertible in the λ-calculus. [3]

Finally, performing strong reduction is also required in implementations of proof assistants or automated theorem provers based on dependent type theories. Such type theories usually include a typing rule similar to the following, called conversion and mentioned in the preface:

$$\frac{\Gamma \vdash t : T' \qquad \Gamma \vdash T' \equiv T}{\Gamma \vdash t : T} \text{ (conv)}$$

The conversion rule behaves like a subtyping rule, and states that whenever a term t has type T' in a typing context Γ, and if T' is equivalent to the type T, then t has also type T. In the simplest case of pure type systems, the equivalence relation $\equiv$ is β-convertibility $=_\beta$, but in more complex type theories like the one used by Coq is a much more complex subtyping relation (see the part about future works on page 230 for more information on convertibility in Coq).

In the rest of this chapter, we explain how the literature tackles the problem of strong evaluation through abstract machines. In section 5.1 we show the only existing *reasonable* strong abstract machine, which implements the Leftmost-Outermost strategy. In section 5.2 we study open evaluation, a subcase of strong evaluation that is simpler but non-trivial.

5.1 Strong Machines

The study of abstract machines for strong evaluation is extremely technical, because they have more data structures and more transitions than the ones that we have introduced in the previous chapter for the weak, closed case. In fact, for more than 25 years there has been only one abstract machine for strong evaluation in the literature, the Normalizing Krivine Machine (NK) by Crégut (1990). NK is an augmented version of the Krivine Abstract Machine (cf. fig. 4.4) that does not only evaluate a λ-term according to CbN, but also iterates evaluation inside the bodies of abstractions producing the strong normal form of that term according to the Leftmost-Outermost startegy.

In order to highlight the difficulty of the topic, we present in fig. 5.1 (a variant of) Crégut's machine: it was named "Strong MAM" by Accattoli, 2016 because reformulated with global environments. Compare it with its weak version, the MAM in fig. 4.5.

[2] See also page 185 for more information on pattern unification.

[3] More precisely, in λProlog convertibility is required up to α, β, and also η, *i.e.* the identification of t with $\lambda x.(t\ x)$ where $x \notin \mathsf{fv}(t)$, see also page 230.

Syntax

Environments	e	$::= \epsilon \mid [x \leftarrow t]e$
Frames	f	$::= \lambda x.\Box \mid i_{\mathsf{s}}\Box \mid \Box t$
Stack	π	$::= \epsilon \mid f :: \pi$
Phase	Φ	$::=$ "◀" $\mid$ "▶"

State

t	Φ	π	e

Initial state

$\underline{t} :=$ | t | ◀ | ϵ | ϵ |

Final state

t_{s}	▶	ϵ	ϵ

Transitions

1	$t\ u$ ◀ π e	→	t ◀ $\Box u :: \pi$ e		
2	$\lambda x.t$ ◀ $\Box s :: \pi$ e	→	t^{α} ◀ π $[x^{\alpha} \leftarrow u]e$		
3	$\lambda x.t$ ◀ π e	→	t ◀ $\lambda x.\Box :: \pi$ e	if 2 not appl.	
4	x ◀ π e	→	t^{α} ◀ π e	if $e(x) = t$	
5	x ◀ π e	→	x ▶ π e	if $x \notin \mathsf{dom}(e)$	
6	i_{s} ▶ $\Box u :: \pi$ e	→	u ◀ $i_{\mathsf{s}}\Box :: \pi$ e		
7	t_{s} ▶ $i_{\mathsf{s}}\Box :: \pi$ e	→	$i_{\mathsf{s}}\ t_{\mathsf{s}}$ ▶ π e		
8	t_{s} ▶ $\lambda x.\Box :: \pi$ e	→	$\lambda x.t_{\mathsf{s}}$ ▶ π e		

The Strong Milner Abstract Machine (Strong MAM) / **Figure 5.1**

Syntax. Just like the MAM, a state of the Strong MAM includes a term t, a stack π which encodes the evaluation context, and a global environment e. Note that the original formulation of the Strong MAM by Accattoli, 2016 includes an additional "frame" component, that here we decided to merge with the stack component for the sake of simplication.

Compared to the abstract machines presented previously in this dissertation, the Strong MAM has a new component Φ called evaluation phase or simply phase: it can be either "evaluation" ◀ or "backtracking" ▶. Intuitively, while in the evaluation phase the machine proceeds to evaluate the code component; in the backtracking phase, instead, the code has already been evaluated to full normal form, and the machine backtracks over the stack frames, searching for the next subterm to evaluate. (Recall that t_{s} and i_{s} denote terms in strong normal form, where i_{s} is not an abstraction, see definition 5.1.)

Transitions. As one can see, the Strong MAM requires more than double the number of transitions than its ancestor the MAM. The transitions 1 2 4 correspond exactly to the transitions of the MAM, and are respectively a search, beta, and substitution transition. All the other transitions are search transitions. Transition 3 is executed when evaluating an unapplied abstraction $\lambda x.t$: in this case, evaluation is iterated inside the body t, and the entry $\lambda x.\Box$ is appended to the stack, indicating that evaluation has entered the abstraction. Transition 5 is executed when evaluating a variable x which is not bound in the environment to any substitution: x may be a globally free variable, or a variable bound by an enclosing abstraction, but in both cases the term x is in normal form; the phase is switched to backtracking, and the machine proceeds to evaluate the arguments of the variable (transition 6). When there are no more arguments to evaluate, transitions 7 and 8 continue backtracking by concluding the evaluation of respectively the argument of an application, or the body of an abstraction.

Decoding. Like the weak machines in section 4.3, we can decode a state of the Strong MAM to a term with ES in the following way: $(t, \Phi, \pi, e)_\downarrow := e_\downarrow\langle\pi_\downarrow\langle t\rangle\rangle$. Note that, since the Strong MAM implements LO evaluation, the stack is the inside-out version of LO contexts, described by the entry C_{LO} in the following grammar:

Definition 5.2 \ Leftmost-Outermost Contexts

$$\begin{array}{lcl} C_{\mathsf{LO}} & ::= & \lambda x.C_{\mathsf{LO}} \mid I_{\mathsf{LO}} \\ I_{\mathsf{LO}} & ::= & \langle\cdot\rangle \mid i_{\mathsf{s}}\, C_{\mathsf{LO}} \mid I_{\mathsf{LO}}\, t \end{array}$$

Complexity. The Strong MAM is not reasonable: in fact, it is subject to the size-explosion phenomenon (see section 4.1) since it computes the full β-normal form of a given term.

Accattoli, 2016 developed an optimized version of the Strong MAM, called *Useful MAM*, with only polynomial overhead. We will not provide the optimized version in this dissertation, because it is quite involved and relies on an auxiliary abstract machine that is executed as a subprocedure; we will however provide the basic intuition underlying the optimization.

Recall that size explosion is caused by unrestricted substitution: in fact, the substitution transition 4 does not distinguish between useful and useless steps, and always performs the substitution of a variable with its content in the environment. The solution for efficiency is to restrict transition 4 to perform only those substitution steps that are useful, hence obtaining the Useful MAM. Basically, a substitution step is useful if it exposes a new redex: a term that contains a redex can always be substituted because the cost of substitution will be compensated by a reduction, and otherwise a term without redexes can be substituted only if it is an abstraction and it will be applied to an argument.

The Useful MAM decides usefulness in the following way. Whenever a new ES $[x{\leftarrow}s]$ is appended to the environment by a transition 2, the Useful MAM executes an auxiliary machine, called "Checking Abstract Machine", that establishes the usefulness of $[x{\leftarrow}s]$ by producing a

label l. The label can be either: "redex", if s contains a β-redex, hence useful; "neutral", if s does not contain any redex and it is not an abstraction, hence useless; "abstraction" if s is an abstraction, hence potentially useful if applied. The produced label is then attached to the entry in the environment, obtaining $[x \leftarrow s]^l$. When later an occurrence of x is found by a transition 4, the Useful MAM replaces x with s only if the label l on $[x \leftarrow s]^l$ says that it is useful. Otherwise the machine backtracks using the transition 5.

The proof that such Useful MAM implements Leftmost-Outermost evaluation with polynomial overhead can be found in the aforementioned paper by Accattoli, 2016.

(No) Other machines. As the Strong/Useful MAM suggests, machines for strong evaluation require many additional transitions, data structures, and delicate optimization. While for CbN evaluation the situation is still manageable (Crégut, 1990; García-Pérez, Nogueira, and Moreno-Navarro, 2013; Accattoli, Barenbaum, and Mazza, 2015; Accattoli, 2016), for CbV and CbNeed the situation becomes quickly desperate—it is not by chance that there is not a single (reasonable) strong abstract machine for CbV/CbNeed in the literature.

For instance, the current evaluator for Coq by Bruno Barras employs a CbNeed-like strategy, but it is not known whether it is reasonable. To clarify what CbNeed consists of in presence of strong evaluation, a recent work by Balabonski et al., 2017 introduced and studied a Strong CbNeed calculus, also proving that it is conservative over classical CbNeed and that it follows the "by need" spirit (in that arguments are only evaluated when needed and at most once). Their approach is to define Strong CbNeed by iterating classical CbNeed (actually, CbNeed adapted to the case of open terms, called "symbolic" CbNeed).

The same approach of obtaining strong evaluation by iterating weak evaluation had been used previously by Grégoire and Leroy, 2002 for a CbV machine, again to improve the implementation of Coq. Their machine proceeds by levels: it first evaluates a term at top-level (*i.e.* outside of all abstractions), and then it is re-launched recursively under each abstraction.

Note that in order to evaluate by levels, one cannot simply iterate one of the usual abstract machines for functional programming languages (say, the MAM or the KAM from section 4.3): when evaluating under an abstraction, the variable bound by that abstraction behaves as free, and therefore the abstract machines must be adapted so to handle open λ-terms. Unfortunately, the naïve handling of open terms with the techniques for FPL gives abstract machines with exponential overhead (Accattoli and Coen, 2015; Accattoli and Guerrieri, 2017). The open setting thus provides a non-trivial intermediate setting: it is close in spirit to the strong case, but clearly easier to study. Nevertheless, it is strictly harder than the closed one.

The aforementioned work by Grégoire and Leroy, 2002, for instance, lacks an estimation of the efficiency of their open CbV machine: in fact, it actually suffers of exponential overhead, even if the authors then in practice implement a slightly different machine with polynomial overhead.

The asymptotic study of the case of Open CbV is in Accattoli and Coen, 2015; Accattoli and Guerrieri, 2017, and is the topic of the next section.

5.2 Open Evaluation

As already mentioned, before the recent wave of abstract machines focused on complexity analyses the literature mostly neglected the open setting, apart from few exceptions (Crégut, 1990; Grégoire and Leroy, 2002; García-Pérez, Nogueira, and Moreno-Navarro, 2013) which however did not address complexity. Nowadays, the literature contains abstract machines that implement all the common strategies (CbN, CbV, CbNeed) in the open setting with only bilinear overhead (Accattoli and Coen, 2015; Accattoli and Barras, 2017).

Since the focus of this dissertation in CbV, in this section we provide the example of an efficient machine performing Open CbV evaluation. Note that naïvely extending the GLAM to open terms leads to a machine with exponential overhead. Accattoli and Coen, 2015 introduce two optimized versions of the Open GLAM, the GLAMOUr and the Unchaining GLAMOUr: they are actually sophisticated machines that rely on careful optimizations, which we will outline in the rest of the section.

The Fast GLAMOUr. In the rest of this section we discuss the Fast GLAMOUr machine by Accattoli and Guerrieri, 2017, an open and reasonable variant of the GLAM (cf. fig. 4.6).

Syntax

Environments $e ::= \epsilon \mid [x \leftarrow f]e$

Stack $\pi ::= \epsilon \mid t\Box :: \pi \mid \Box f :: \pi$

State

$t \mid \pi \mid e$

Initial states

$\underline{t} := t \mid \blacktriangleleft \mid \epsilon \mid \epsilon$

Final state

$f \mid \Phi \mid \epsilon \mid e$

Transitions

1	$t\,s \mid \blacktriangleleft \mid \pi \mid e$	$\to$	$s \mid \blacktriangleleft \mid t\Box :: \pi \mid e$	
2	$\lambda x.t \mid \blacktriangleleft \mid \Box y :: \pi \mid e$	$\to$	$t\{x \Leftarrow y\} \mid \blacktriangleleft \mid \pi \mid e$	
3	$\lambda x.t \mid \blacktriangleleft \mid \Box f :: \pi \mid e$	$\to$	$t^{\alpha} \mid \blacktriangleleft \mid \pi \mid [x^{\alpha} \leftarrow f] :: e$	if $f \notin \mathcal{V}$
4	$\lambda x.t \mid \blacktriangleleft \mid s\Box :: \pi \mid e$	$\to$	$s \mid \blacktriangleleft \mid \Box(\lambda x.t) :: \pi \mid e$	
5	$x \mid \blacktriangleleft \mid \Box f :: \pi \mid e$	$\to$	$v^{\alpha} \mid \blacktriangleleft \mid \Box f :: \pi \mid e$	if $e(x)$ is abs.
6	$x \mid \blacktriangleleft \mid \pi \mid e$	$\to$	$x \mid \blacktriangleright \mid \pi \mid e$	otherwise
7	$i \mid \blacktriangleright \mid \Box f :: \pi \mid e$	$\to$	$if \mid \blacktriangleright \mid \pi \mid e$	
8	$f \mid \blacktriangleright \mid s\Box :: \pi \mid e$	$\to$	$s \mid \blacktriangleleft \mid \Box f :: \pi \mid e$	

The Fast GLAMOUr / **Figure 5.2**

Syntax. Just like the GLAM, a state of the Fast GLAMOUr includes a term t, a stack π which encodes the evaluation context, and a global environment e. Note that the original formulation of the Fast GLAMOUr by Accattoli and Guerrieri, 2017 includes an additional "dump" component, that here we decided to merge with the stack component for the sake of uniformity with previous machines. Getting rid of the dump has the disadvantage that our reformulation requires the notion of evaluation phase, like Crégut's machine, and it has more transitions.

The other main difference with respect to the GLAM of fig. 4.6 is that the syntactic category of values has been replaced with the one of fireballs, as one would expect when moving from λ_{Plot} evaluation to λ_{fire} evaluation. Recall that f and i denote respectively fireballs and inert terms, as defined in fig. 3.2.

Transitions. As one can see, the Fast GLAMOUr is quite more complex than the its ancestor the GLAM. A source of difficulty comes from the optimizations needed to make the machine reasonable; we will discuss these optimizations below, in the section about complexity. The transitions 1 2 3 4 5 correspond exactly to the transitions of the GLAM, with the only difference that the beta transition of the GLAM has been split in the two beta transitions 2 + 3 due to an optimization (see below): transition 2 is applied when the argument of the redex is a variable, and transition 3 when the argument is any other fireball.

All the other transitions are search transitions and control the backtracking phase. Transition 6 is applied when evaluating a variable x, and in two cases: when $e(x)$ is not an abstraction, meaning that x is either a globally free variable, or bound in the environment to an inert term; or when $e(x)$ is an abstraction, but the stack π does not begin with an entry of the form $\Box f$. We will discuss the reason for this side-condition below.

After a 6 transition, the machine enters the backtracking phase and basically builds up an inert term by repeatedly applying fireball arguments to the head variable. Transition 7 picks the fireball arguments from the stack and appends them to the inert i. When there are no more fireball arguments in the stack, a 8 transition stores in the stack the inert that has just been built — which is a right part of an application — and continues evaluation of the left part of an application. In the original formulation of Accattoli and Guerrieri, 2017, the transitions 6 7 8 are merged in a single transition called "c_3".

Decoding. Like for all previous machines, we decode a state of the Fast GLAMOUr to a term with ES in the following way: $(t, \Phi, \pi, e)_{\downarrow} := e_{\downarrow}\langle \pi_{\downarrow}\langle t \rangle\rangle$. Note that, since the FAST GLAMOUr implements λ_{fire} evaluation, the stack is the inside-out version of f-contexts, described by the entry R in the grammar on page 34.

Complexity. The good complexity of the Fast GLAMOUr relies critically on the following two optimizations:

1. *No substitution of variables.* A potential performance issue comes from the seemingly innocuous subsitution of variables. If variables are treated just like values, and can thus be added to ES in the environment and substituted, then the overhead of the machine becomes quadratic (this is what happens in the GLAMOUr machine of Accattoli and Coen, 2015).

   The cause for the quadratic overhead is the presence in the environment of chains of ES like the following:

   $$[x_1 \leftarrow x_2][x_2 \leftarrow x_3] \cdots [x_k \leftarrow \lambda y.t].$$

   This is a chain of variable substitutions ending up in an abstraction. Let us suppose for instance that a machine is evaluating the code x_1 with respect to the environment above; then the machine needs to perform k substitution steps before substituting the abstraction $\lambda y.t$: since k can be proportional to the number of β-steps, this phenomenon causes the number of substitution steps to become quadratic in the number of β-steps. The relationship between substituting variables and a linear or quadratic overhead is studied in-depth by Accattoli and Coen, 2014.

   The Unchaining GLAMOUr machine by Accattoli and Coen, 2015 solves this issue by adding labels and a further *unchaining optimization*. Another solution, which is the one adopted for the Fast GLAMOUr, is to split a β-transition in two distinct transitions, handling this situation with the new transition 2 that renames the variable in a body t without altering the environment. In this way, the Fast GLAMOUr disallows chains of renamings like the one above by avoiding that ES substituting variables are ever appended to the environment. This latter solution is a known optimization in the literature of abstract machines (Sands, Gustavsson, and Moran, 2002; Friedman et al., 2007; Wand, 2007).

2. *On-demand substitution of abstractions.* Recall that, for performance reasons, in Crégut's machine (fig. 5.1) the substition transition had to be restricted in such a way that only useful terms could be substituted. In that case, checking whether a term is useful was a non-trivial task that required an auxiliary subprocedure to scan the term and store a label. The Unchaining GLAMOUr of Accattoli and Coen, 2015 uses a similar idea of labeling, which is however made unnecessary by the optimization in the previous point, and by the stronger invariants that CbV evaluation provides. To check whether a substitution is useful in the Fast GLAMOUr, it suffices to check whether the variable x is bound in the environment to an abstraction, and whether x is applied by checking the stack (see transition 5). In fact, the CbV strategy forces the environment to contains only terms in normal form, *i.e.* fireballs, and by the optimization discussed above it cannot contain variables that mask malicious chains. Therefore the check for usefulness of an ES $[x \leftarrow f]$ can be performed in constant time by simply checking that f is syntactically an abstraction.

The proof that the Fast GLAMOUr implements λ_{fire} evaluation with only bilinear overhead can be found in the aforementioned paper by Accattoli and Guerrieri, 2017. Note that the optimizations sketched above are fundamental to obtain bilinear complexity.

We have seen above that porting an abstract machine to the open case is a delicate task, and it usually results in a machine that is quite more complex then the original, closed one. In part II we will revisit Open CbV evaluation by defining a new kind of abstract machines, that we call "crumble machines" because based on an alternative compilation of λ-terms that we call "crumbling". As we will see, open crumble machines are much simpler than the existing machines for open evaluation: as a consequence, abstract machines for Strong CbV evaluation become within grasp (see section 15.1).

Chapter 6

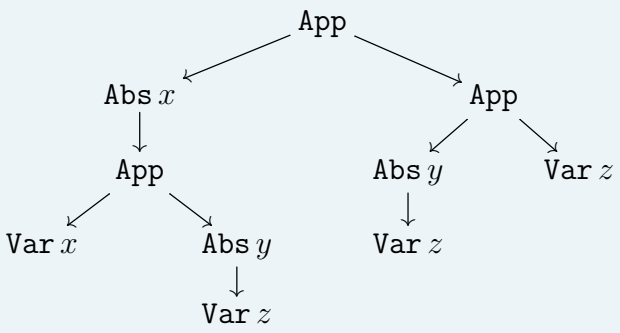
Terms as Graphs

In section 4.2 we added sharing directly to the syntax of the λ-calculus. In this chapter instead we take a *graphical* approach: we are going to introduce λ-graphs, a natural way of representing shared λ-terms as graphs. λ-graphs are fundamental because they are both the data structure that we implement our abstract machines on (part II) and also the one required by our sharing equality algorithm (part III).

λ-graphs are a pretty straightforward representation of shared λ-terms, but before turning to the next chapter we will also compare them to alternative or more complex graph-based representations for sharing: vertical sharing, proof nets, and sharing graphs.

6.1 λ-graphs

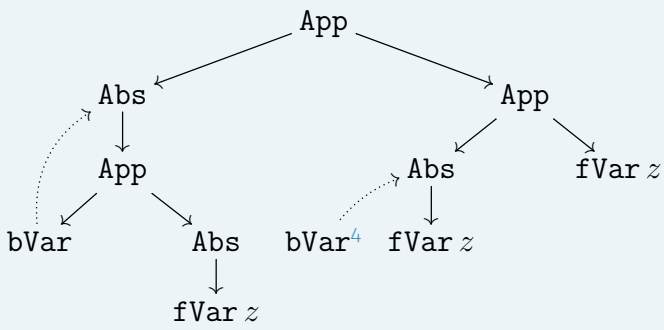
One can see a λ-term as a syntax tree in a natural way: consider for example the following tree, which represents the named λ-term $(\lambda x.x\ (\lambda y.z))\ ((\lambda y.z)\ z)$.



Named λ-tree / **Figure 6.1**

As expected, the syntax tree above contains three kinds of nodes, which are in a one-to-one correspondence with the constructors of λ-terms: `App`, `Abs`, or `Var` nodes.

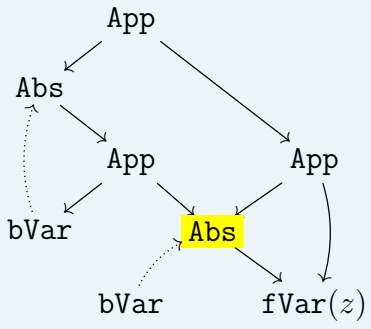
More faithful to the (locally) nameless representation[1] however is the following syntax tree, which represents the same λ-term:



Locally nameless λ-tree / **Figure 6.2**

In the syntax tree of fig. 6.2 free and bound variables are represented by different kinds of nodes: free variable nodes (labelled by `fVar`) carry a name, while bound variable nodes (labelled by `bVar`) do not carry a name but have a backward (dotted) edge towards the abstraction node that binds them. This backward edge, which we call binding edge, is a way of representing scopes that dates back to Bourbaki in *Eléments de Théorie des Ensembles*, but also supported by the strong relationship between λ-calculus and linear logic proof nets.

In the graphical setting we can realize sharing by simply allowing nodes to have more than one parent, as for instance the highlighted abstraction node in fig. 6.3: in this way, the two identical subgraphs present in fig. 6.2 can now be represented by a single piece of graph. Note that sharing can also happen under abstractions, *e.g.* the highlighted node is shared under the other abstraction node: however, nodes in the scope of an abstraction are shareable only when they respect a structural property that we will formally define in definition 6.8.

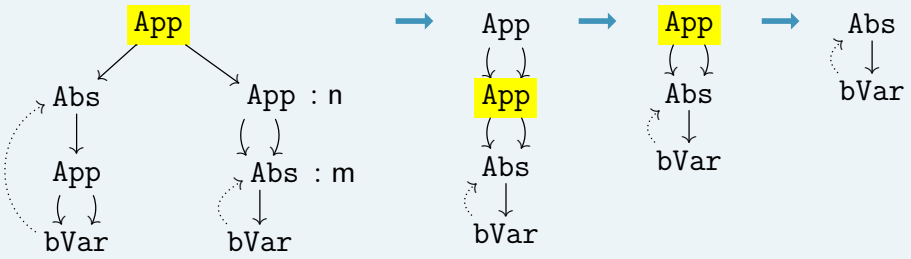


λ-graph with sharing / **Figure 6.3**

Shared evaluation. Representing λ-terms as graphs dates back to the PhD thesis of Wadsworth, 1971; in that same thesis, Wardsworth introduced the Call-by-Need evaluation strategy[5], which

[1] See section 2.4.
[4] This bound variable node is not used, so it may be as well omitted.
[5] See page 26.

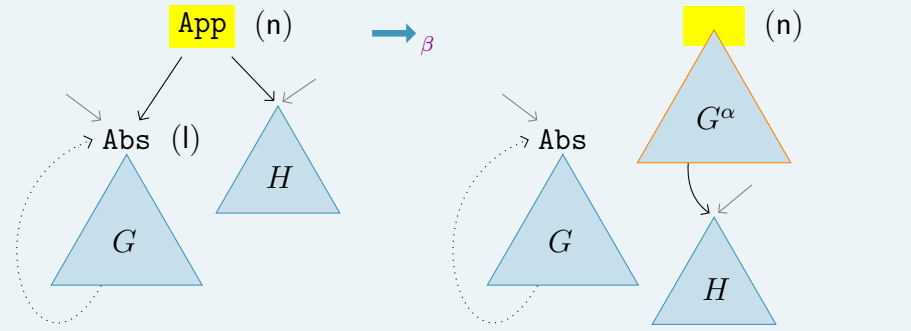
afterwards became widespread. λ-graphs are particularly suited to implement CbNeed because they can naturally express sharing and the destructive update necessary to cache evaluated arguments. Consider for example the graph reduction sequence in fig. 6.4 according to the CbNeed strategy.



Example of graph rewriting sequence / **Figure 6.4**

The first λ-graph in the sequence corresponds to the λ-term $(\lambda y.yy)\,(I\,I)$ where $I := \lambda x.x$, or more precisely to the term with ES $(\lambda y.yy)\,(z\,z)[z\leftarrow I]$ where the sharing variable z corresponds to the node **m**. We reduce each graph in the sequence by contracting the redex corresponding to the highlighted node, rewriting the graph itself as to mimic β-reduction. The redex in the first graph is contracted without evaluating its argument **n**: the resulting second graph corresponds to the term $yy[y\leftarrow zz][z\leftarrow I]$, and the node **n** occurs shared since it is both the left and right child of the root. Then **n** is evaluated in place, being replaced by **m** and yielding the third graph, which corresponds to the term $yy[y\leftarrow I]$. Finally we obtain the fourth graph, whose root is the abstraction **m** and which is in normal form.

β as a graph rewrite rule is depicted in fig. 6.5—cf. β in the λ-calculus, page 19.



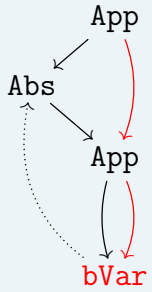
Graph β-rewriting / **Figure 6.5**

Essentially, β in fig. 6.5 replaces the highlighted node **n** with the root node of the subgraph G^α, a copy of G where every node labeled with $\texttt{bVar}(\mathsf{l})$ has been replaced with the root node of H. Note that all these nodes might be part of a bigger graph: **l** may as well be shared, *i.e.* it may have parents other than **r**. Therefore one must copy (part of) the body G of **l** before instantiating $\texttt{bVar}(\mathsf{l})$, in order to preserve sharing with other arguments.

Incidentally, the operation of copying a function body during a β-step is totally analogous

to what happens in the plain λ-calculus. In linear logic proof nets — a generalization of λ-graphs that we mention in section 6.2 — a β step is instead decomposed in a finer way, and the copy operation is performed by a dedicated reduction step, called "exponential".

Correctness of binders. Not every graph corresponds to a valid λ-term: consider for example the graph in fig. 6.6.



Graph breaking domination / **Figure 6.6**

The highlighted bound variable is visible outside the scope of its binder: there is a path from the root node to the variable node, that does not pass through the abstraction node that binds the variable. At first sight one may say that the graph in fig. 6.6 represents the term $t := (\lambda x.xx)\ (xx)$, however this cannot be the case since x is free in xx but bound in $\lambda x.xx$, hence the two occurrences in t of the subexpression xx are totally different and cannot be shared. It is well-known that scopes corresponding to λ-terms are characterized by a property borrowed from control-flow graphs called domination, also called *unique binding* by Wadsworth, 1971, and checkable in linear time (Alstrup et al., 1999; Buchsbaum et al., 1998; Gabow, 1990).

Before defining formally the property of domination and the structural properties that characterize λ-graphs we introduce pre-λ-graphs, graphs where we do not yet consider variable nodes and scopes. We then formalize our notion of λ-terms with sharing in definition 6.8.

Definition 6.1 \ pre-λ-graphs

A *pre-λ-graph* is a directed labeled graph such that:

- *Applications*: an application node is labelled with $\mathtt{App}$ and has exactly two children, called *left* ($\swarrow$) and *right* ($\searrow$). We write $\mathtt{App}(\mathsf{n}, \mathsf{m})$ for a node labelled by $\mathtt{App}$ whose left child is n and whose right child is m.
- *Abstractions*: an abstraction node is labelled with $\mathtt{Abs}$ and has exactly one child, called its *body* ($\downarrow$). We write $\mathtt{Abs}(\mathsf{n})$ for a node labelled by $\mathtt{Abs}$ with body n. We denote with $\mathsf{l}, \mathsf{l}', \ldots$ generic abstraction nodes.

Notations for paths. Paths are a crucial concept, needed both to define the readback to λ-terms and to state formally the properties of λ-graphs of being acyclic and dominated. A path in a graph is determined by a start node together with a *trace*, *i.e.* a sequence of directions:

Definition 6.2 \ Paths, traces

We define *traces* as finite sequences of directions:

$$\text{Directions} \quad d ::= \swarrow \mid \downarrow \mid \searrow$$
$$\text{Traces} \quad \tau ::= \epsilon \mid \tau \cdot d$$

Let n, m be nodes of a pre-λ-graph G, and τ be a trace. We define inductively the judgement "$\tau : \mathsf{n} \rightsquigarrow \mathsf{m}$", which reads "path from n to m (of trace τ)":

- *Empty*: $\epsilon : \mathsf{n} \rightsquigarrow \mathsf{n}$.
- *Abstraction*: if $\tau : \mathsf{n} \rightsquigarrow \mathtt{Abs}(\mathsf{m})$, then $(\tau \cdot \downarrow) : \mathsf{n} \rightsquigarrow \mathsf{m}$.
- *Application*: if $\tau : \mathsf{n} \rightsquigarrow \mathtt{App}(\mathsf{m}, \mathsf{p})$, then $(\tau \cdot \swarrow) : \mathsf{n} \rightsquigarrow \mathsf{m}$ and $(\tau \cdot \searrow) : \mathsf{n} \rightsquigarrow \mathsf{p}$.

We just write $\tau : \mathsf{n}$ if $\tau : \mathsf{n} \rightsquigarrow \mathsf{m}$ for some node m when the endpoint m is not relevant.

Definition 6.3 \ Acyclic pre-λ-graph

We say that a pre-λ-graph G is *acyclic* when for every node n in G and every trace τ, $\tau : \mathsf{n} \rightsquigarrow \mathsf{n}$ if and only if $\tau = \epsilon$.

Root nodes. Pre-λ-graphs (and later λ-graphs) may have various root nodes. What is maybe less expected, is that these roots may share some parts of the graph. Consider fig. 6.3, and imagine to remove the root and its edges: the outcome is still a perfectly legal pre-λ-graph. We admit these configurations because they actually arise naturally in implementations, where the space in memory which stores many λ-terms can be seen as a single huge λ-graph.

Definition 6.4 \ Roots

Let r be a node of a pre-λ-graph G. r is a *root* if and only if the only path with endnode r has empty trace.

Defining the property of domination requires the following two additional concepts:

Definition 6.5 \ Access path (Ariola and Klop, 1996)

An *access path* to a node n of a λ-graph G is a path from a root node of G to n.

Definition 6.6 \ Path crossing a node

Let n, m be nodes of a pre-λ-graph, and τ a trace such that $\tau : \mathsf{n}$. We define inductively the judgment "$\tau : \mathsf{n}$ crosses m":

- if $\tau : \mathsf{n} \rightsquigarrow \mathsf{m}$, then $\tau : \mathsf{n}$ crosses m
- if $\tau : \mathsf{n}$ crosses m and $(\tau \cdot d) : \mathsf{n}$, then $(\tau \cdot d) : \mathsf{n}$ crosses m.

Definition 6.7 \ Domination

Let G be a pre-λ-graph, and n, m be nodes of G: we say that m dominates n when every access path to n crosses m.

Recall fig. 6.6: the path highlighted in red is an access path to the bound variable node, which does not cross the abstraction node. Therefore the abstraction node does not dominate the bound variable node, and the pre-λ-graph is not a λ-graph according to the upcoming definition.

Definition 6.8 \ λ-graphs

A pre-λ-graph G is a λ-*graph* if it satisfies the following additional structural properties:

- *DAG.* G is finite and acyclic (see definition 6.3).
- *Variables.* A λ-graph has exactly three kinds of nodes: abstraction, application, and variable nodes. Variable nodes may be either *free* or *bound*:
  - A *free* variable node has no children, and it carries a name $x \in \mathcal{V}$. A free variable node of atom x is denoted by $\texttt{fVar}(x)$.
  - A *bound* variable node has exactly one child, called its *binder* (↻). We write $\texttt{bVar}(\mathsf{l})$ for a node labelled by `Var` with binder l.
- *Dominated*: each bound variable node $\texttt{bVar}(\mathsf{l})$ of G is dominated by its binder l.

Note that the DAG requirement implies that there is a maximal length for paths, thus providing an induction principle on λ-graphs.

Readback of λ-graphs to λ-terms

The sharing in a λ-graph can be unfolded by duplicating shared sub-graphs, obtaining a λ-tree. We prefer however to adopt another approach. We define a readback procedure associating a locally nameless λ-term $[\![\mathsf{r}]\!]_G$ (without sharing) to each root node r of a λ-graph G, in such a way that shared sub-graphs simply appear multiple times. The readback needs to traverse recursively the children of the node to read back, but since we use de Bruijn indices for bound variables, any node of the graph by itself does not uniquely identify a λ-

term: in fact, its readback depends on the chosen access path. That path determines the abstraction nodes encountered, and thus the indices to assign to bound variable nodes.

We thus define formally $\mathtt{index}(\mathsf{l} \mid \tau : \mathsf{n})_G$, the index of an abstraction node l according to a path $\tau : \mathsf{n}$ crossing l (recall that l, l' denote abstraction nodes):

Definition 6.9 \ $\mathtt{index}(\mathsf{l} \mid \tau : \mathsf{n})_G$

Let n, l be nodes of a λ-graph G, and $\tau : \mathsf{n}$ a path crossing l. We define $\mathtt{index}(\mathsf{l} \mid \tau : \mathsf{n})$ by structural induction on the derivation of the judgement "$\tau : \mathsf{n}$ crosses l":

$$\begin{array}{rcll} \mathtt{index}(\mathsf{l} \mid \tau : \mathsf{n} \rightsquigarrow \mathsf{l}) & := & 0 & \\ \mathtt{index}(\mathsf{l} \mid (\tau \cdot d) : \mathsf{n} \rightsquigarrow \mathsf{l}') & := & \mathtt{index}(\mathsf{l} \mid \tau : \mathsf{n}) + 1 & \text{if } \mathsf{l} \neq \mathsf{l}' \\ \mathtt{index}(\mathsf{l} \mid (\tau \cdot d) : \mathsf{n} \rightsquigarrow \mathsf{m}) & := & \mathtt{index}(\mathsf{l} \mid \tau : \mathsf{n}) & \text{otherwise.} \end{array}$$

Definition 6.10 \ Readback to λ-terms $[\![\tau : \mathsf{r} \rightsquigarrow \mathsf{n}]\!]_G$

Let G be a λ-graph. For every root r and path $\tau : \mathsf{r} \rightsquigarrow \mathsf{n}$, we define the readback $[\![\tau : \mathsf{r} \rightsquigarrow \mathsf{n}]\!]_G$ of n relative to the access path $\tau : \mathsf{r} \rightsquigarrow \mathsf{n}$, by cases on n:

1. $[\![\tau : \mathsf{r} \rightsquigarrow \mathtt{bVar}(\mathsf{l})]\!] := \#i$ where $i := \mathtt{index}(\mathsf{l} \mid \tau : \mathsf{r})$.
2. $[\![\tau : \mathsf{r} \rightsquigarrow \mathtt{fVar}(x)]\!] := x$.
3. $[\![\tau : \mathsf{r} \rightsquigarrow \mathtt{Abs}(\mathsf{m})]\!] := \lambda\, [\![(\tau \cdot \downarrow) : \mathsf{r} \rightsquigarrow \mathsf{m}]\!]$.
4. $[\![\tau : \mathsf{r} \rightsquigarrow \mathtt{App}(\mathsf{n}_1, \mathsf{n}_2)]\!] := [\![(\tau \cdot \swarrow) : \mathsf{r} \rightsquigarrow \mathsf{n}_1]\!]\, [\![(\tau \cdot \searrow) : \mathsf{r} \rightsquigarrow \mathsf{n}_2]\!]$.

We will usually omit G from $[\![\tau : \mathsf{r} \rightsquigarrow \mathsf{n}]\!]_G$ when unambiguous. Also, we just write $[\![\mathsf{r}]\!]$ instead of $[\![\epsilon : \mathsf{r}]\!]$.

Some remarks about definition 6.10:

- The hypothesis that r is a root node is necessary to ensure that the readback $[\![\tau : \mathsf{r}]\!]$ is well-defined. In fact Point 1 of the definition uses $\mathtt{index}(\mathsf{l} \mid \tau : \mathsf{r})$, which is defined only if $\tau : \mathsf{r} \rightsquigarrow \mathtt{bVar}(\mathsf{l})$ crosses l. When r is a root node, this is the case by the requirement "domination" of definition 6.8.
- The definition is recursive, but it is not immediately clear what is the measure of termination. In fact, the readback calls itself recursively on longer paths. Still, the definition is well-posed because paths do not use binding edges, and because λ-graphs are finite and acyclic (definition 6.3).

We will come back to λ-graphs in part III, where we discuss how to decide whether different λ-graphs represent the same λ-term without actually performing the readback. The central notion will be the one of "sharing equivalences", relations on graphs which are roughly bisimulations. Then we are going to prove that sharing equivalence correctly characterize the

equality of the readback to λ-terms, and provide our algorithm for sharing equality that runs in time proportional to the λ-graphs to be checked.

Before concluding this section on readbacks, we prove the following proposition that connects readback and traces; it will be useful later, in section 12.3.

Proposition 6.11 \ Readback vs. traces

Let r, r' be roots of a λ-graph G. $[\![\mathsf{r}]\!] = [\![\mathsf{r}']\!]$ holds if and only if, for every trace τ:

1. *Trace Equivalence:* $\tau : \mathsf{r}$ if and only if $\tau : \mathsf{r}'$;
2. *Trace Propagation:* if $\tau : \mathsf{r}$ and $\tau : \mathsf{r}'$, then $[\![\tau : \mathsf{r}]\!] = [\![\tau : \mathsf{r}']\!]$.

Proof.

$(\Rightarrow)$ We assume that $[\![\mathsf{r}]\!] = [\![\mathsf{r}']\!]$, and proceed by structural induction on τ:

- *Empty Trace.* Clearly $\epsilon : \mathsf{r}$ and $\epsilon : \mathsf{r}'$. The fact that $[\![\epsilon : \mathsf{r}]\!] = [\![\epsilon : \mathsf{r}']\!]$ follows from the hypothesis, since $[\![\epsilon : \mathsf{r}]\!] = [\![\mathsf{r}]\!]$ and $[\![\epsilon : \mathsf{r}']\!] = [\![\mathsf{r}']\!]$.
- *Trace Cons.* Let $\tau := \tau' \cdot d$, and assume without loss of generality that $\tau : \mathsf{r}$. We need to prove that $\tau : \mathsf{r}'$ and that $[\![\tau : \mathsf{r}]\!] = [\![\tau : \mathsf{r}']\!]$.
  By inversion $\tau' : \mathsf{r}$, and by *i.h.* $\tau' : \mathsf{r}'$ and $[\![\tau' : \mathsf{r}]\!] = [\![\tau' : \mathsf{r}']\!]$. We proceed by cases on d:
  * Case $d = \downarrow$. Necessarily $\tau' : \mathsf{r} \rightsquigarrow \mathtt{Abs}(\mathsf{n})$ and $\tau' : \mathsf{r}' \rightsquigarrow \mathtt{Abs}(\mathsf{m})$ for some n, m. By the definition of readback to λ-terms, $[\![\tau' : \mathsf{r} \rightsquigarrow]\!] = \lambda[\![\tau : \mathsf{r} \rightsquigarrow \mathsf{n}]\!]$ and $[\![\tau' : \mathsf{r}' \rightsquigarrow]\!] = \lambda[\![\tau : \mathsf{r}' \rightsquigarrow \mathsf{m}]\!]$, which imply $[\![\tau : \mathsf{r} \rightsquigarrow \mathsf{n}]\!] = [\![\tau : \mathsf{r}' \rightsquigarrow \mathsf{m}]\!]$ by the definition of equality.
  * Case $d = \swarrow$. Necessarily $\tau' : \mathsf{r} \rightsquigarrow \mathtt{App}(\mathsf{n}_1, \mathsf{n}_2)$ and $\tau' : \mathsf{r} \rightsquigarrow \mathtt{App}(\mathsf{m}_1, \mathsf{m}_2)$ for some $\mathsf{n}_1, \mathsf{n}_2, \mathsf{m}_1, \mathsf{m}_2$. By the definition of readback to λ-terms, $[\![\tau' : \mathsf{r}]\!] = [\![\tau : \mathsf{r} \rightsquigarrow \mathsf{n}_1]\!]\,[\![(\tau' \cdot \searrow) : \mathsf{r} \rightsquigarrow \mathsf{n}_2]\!]$ and $[\![\tau' : \mathsf{r}']\!] = [\![\tau : \mathsf{r}' \rightsquigarrow \mathsf{m}_1]\!]\,[\![(\tau' \cdot \searrow) : \mathsf{r}' \rightsquigarrow \mathsf{m}_2]\!]$, which imply $[\![\tau : \mathsf{r} \rightsquigarrow \mathsf{n}_1]\!] = [\![\tau : \mathsf{r}' \rightsquigarrow \mathsf{m}_1]\!]$ by the definition of equality.
  * Case $d = \searrow$. Similar to the case above.

$(\Leftarrow)$ The statement follows from the hypothesis *Trace Propagation* by taking $\tau := \epsilon$. ∎

6.2 More Kinds of Sharing

The type of sharing present in λ-graphs is also known as horizontal or subterm sharing: it is essentially the same sharing mechanism available in calculi with explicit substitutions

(section 4.2), environment-based abstract machines (section 4.3), or linear logic proof nets (see below): the details are different but all these approaches provide different incarnations of the same notion of sharing.

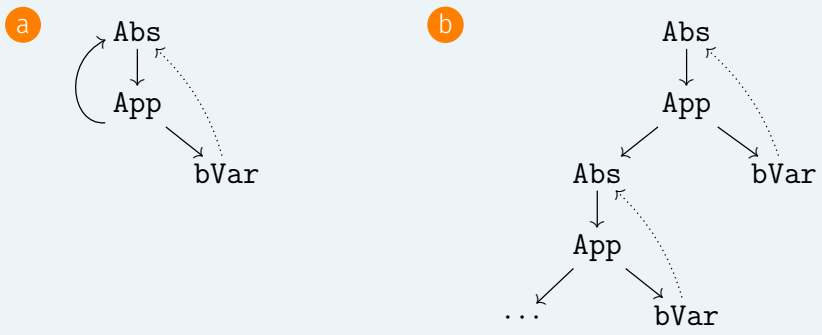
λ-graphs can be tweaked slightly in order to express other kinds of sharing, for instance "vertical" and "twisted" sharing (see Blom, 2001) which allow to encode infinite λ-terms in a finite way. Moreover, powerful generalizations of λ-graphs exist, like *linear logic proof nets* and *sharing graphs*. We sketch these alternative kinds of sharing in the following sections, but we will not employ them further in this dissertation.

Vertical Sharing

Vertical sharing corresponds in functional programming languages to cyclic definitions, which model recursive functions. We can add vertical sharing to the λ-calculus by extending its syntax with μ-expressions, having the form "$\mu x.t$" where μ binds the x in t. Consider for instance the following μ-term:

$$t \;\coloneqq\; \mu f.\lambda x.fx$$

This term can be seen as representing an "infinite" λ-term, obtained intuitively by continually unfolding the μ-expression using the rule $\mu x.s \mapsto s\{x \leftarrow \mu x.s\}$, in an infinite process that yields the term $u \coloneqq \lambda x.(\lambda y.\,(\cdots)\,y)\,x$. The graphs corresponding to these terms are in fig. 6.7, the graph for t on the left and the one for its unfolding u on the right.



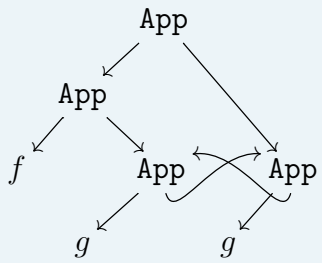
Cyclic λ-graph and its unfolding / **Figure 6.7**

Clearly the graphs in fig. 6.7 are not just plain λ-graphs: to represent μ-expressions we need to relax the acyclicity requirement and allow cyclic subgraphs (fig. 6.7a), and to represent unfoldings we need to allow infinite subgraphs (fig. 6.7b).

Twisted Sharing

Horizontal and vertical sharing can be combined by means of the `letrec` construct: the expression "`letrec` $x = t$ `in` s" is basically equivalent to "`let` $x = \mu x.t$ `in` s". *Twisted* sharing, instead, is an even more expressive combination of horizontal and vertical sharing. It corresponds to the feature of programming languages to define blocks of mutually recursive

functions: for instance the term "$\texttt{letrec}\ x = g\ y\ \texttt{and}\ y = g\ x\ \texttt{in}\ f\ x\ y$", represented as the cyclic λ-graph in fig. 6.8.



Graph with twisted sharing / **Figure 6.8**

A λ-calculus with such `letrec`-expressions having multiple mutual entries is called "cyclic λ-calculus" by Ariola and Klop, 1997, or $\lambda_{\texttt{letrec}}$ by Grabmayer and Rochel, 2014.

Handling cyclic λ-graphs is slightly more difficult than plain λ-graphs because of additional subtleties. For instance, the usual notion of domination (see definition 6.7) does not suffice to characterize $\lambda_{\texttt{letrec}}$-terms: it becomes necessary to add to the graph an explicit concept of scope. To do so, Grabmayer and Rochel, 2014 assign to each node **n** a so-called "abstraction prefix", which is a list of the `Abs`-nodes in whose scope **n** resides. Enriching cyclic graphs with abstraction prefixes is also necessary to define a correct notion of bisimulation for $\lambda_{\texttt{letrec}}$-graphs, which is otherwise ill-behaved as we will show on page 181.

Linear Logic Proof Nets

Linear logic was introduced by Jean-Yves Girard, 1987 and has had an immense impact on computer science since then. The success of linear logic is due to its rigorous account of sharing: not all objects are sharable, and duplicating an object is permitted only if that object is equipped by the ! (read "bang") modality. This allows to decompose computation in two parts:

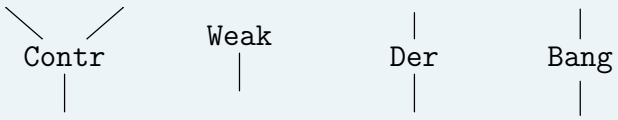
1. a *linear* process where each argument is used once and exactly once;
2. an explicit process of *copying* and *discarding*, in which linearity is broken but in a controlled way.

To show how proof nets relate to λ-graphs, let us first consider one of the simplest fragments of linear logic: it is called *MLL* for multiplicative linear logic, and its connectives are the multiplicative conjunction $\otimes$ (read "tensor") and disjunction $\parr$ (read "par"). MLL proof nets are formed by the *links* in fig. 6.9.

MLL links / **Figure 6.9**

Note that each type of link has a fixed number of connections, hence sharing is disallowed altogether. Other than that, the nodes of λ-graphs and the links of MLL are quite similar. The links $\parr$ and $\otimes$ quite literally correspond respectively to `Abs` and `App` nodes in λ-graphs. In addition, MLL proof nets contain `Ax` and `Cut` links that do not have an exact counterpart in λ-graphs; however these links are simply "identity connections", whose only role is to preserve some structural properties of the net, and they can thus be safely ignored.

Unfortunately, MLL proof nets only represent "linear" λ-terms, *i.e.* terms where each bound variable occurs exactly once (and similarly, there cannot be multiple occurrences of a free variable): in this way, no duplication or discarding can happen during reduction. To recover the full expressive power of λ-terms and exit the linear framework, it is necessary to add to MLL the *bang* "exponential" link. In this way we obtain the logic *MELL*—which stands for multiplicative exponential linear logic—whose nets are formed by the links of MLL plus the ones in fig. 6.10.



MELL links / **Figure 6.10**

A mismatch between MELL proof nets and λ-graphs is that all nodes of a λ-graph can be shared (variables included), while in proof nets only banged sub-nets can be shared, and only by means of binary contractions. Moreover, a notion of *boxes* is necessary to delimit the action of `Bang` links. This gap can be eliminated by tweaking the representation of proof nets: a solution is to use so-called *nouvelle syntaxe* and introducing n-ary `Bang` links that collapse contractions, weakenings, and derelictions (Regnier, 1992; Accattoli, Barenbaum, and Mazza, 2014a).

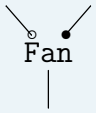
In this way, λ-graphs become essentially the same representation induced by the translation of λ-calculus into linear logic proof nets: the only missing feature of λ-graphs are explicit sharing `Bang` links and boxes, where `Bang`s are basically the explicit sharing nodes that we have discussed in the section on low-level implementation details, on page 64. For more details on the correspondence between the λ-calculus with sharing and proof nets, see for instance Accattoli, 2018.

Sharing Graphs

Sharing graphs are a variant of λ-graphs providing a much deeper form of sharing, and were introduced by Lamping, 1990 to implement his algorithm for optimal reduction in the λ-calculus. The intuition behind optimal reduction is to perform reduction while strictly avoiding any copying that could later cause a duplication of work. Optimal reduction cannot be performed through usual *sequential* strategies, but requires a notion of *parallel* reduction of all the redexes of a same family at the same time (Levy, 1978). This cannot be achieved

with the sharing present in λ-graphs, which merely allows to reuse the first-order structure of subterms: it becomes crucial a form of *higher-order sharing*, where not only subterms are shared but also *term contexts*, *i.e.* terms with holes that can be filled by each sharing instance in a different way.

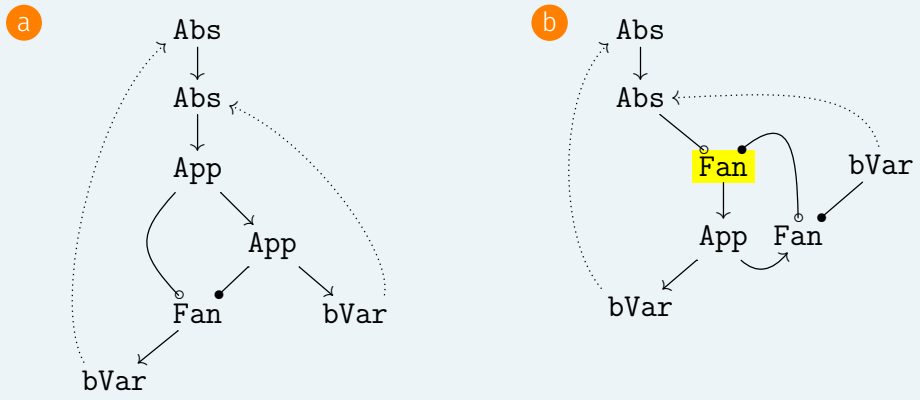
Higher-order sharing can be obtained by extending λ-graphs with special sharing nodes called *fans*, having three ports, two of which marked with ○ and ● (note: the port marked here by ● is traditionally marked by $*$).



Fan link / **Figure 6.11**

Fans correspond more-or-less to contraction nodes in linear logic proof nets (cf. fig. 6.10) and can behave in two ways. *Fan-in* nodes introduce sharing, and dually *fan-out* nodes "unshare": each fan-out node is paired with a fan-in node, which delimites its scope and annihilates its sharing effect. When reading back a λ-term from a sharing graph, one visits the graph on the proviso that after entering a fan-in node by a certain port, they have to exit by its matching fan-out using the same port.

Consider for example the two sharing graphs in fig. 6.12, both denoting the same λ-term $\lambda x.\, \lambda y.\, x\,(x\,y)$:



Examples of sharing graphs / **Figure 6.12**

As we see in the sharing graph in fig. 6.12b, fan nodes enable certain kinds of cyclic structures: the shared part with highlighted root node contains a cycle, meaning that it is reusing a different instance of itself. Cyclic sharing graphs can be totally valid sharing graphs, and in fact the graph in fig. 6.12b can actually be obtained during reduction, even when starting from an initial graph that is acyclic.

Correct sharing graphs. Cyclicity complicates the question of how to establish the correct pairing between fan-ins and fan-outs during reduction in a local way. Lamping's algorithm used a vague notion of enclosure to delimit the interaction of fans, which was then made formal by Asperti[6] by means of *levels*: each node is decorated with an additional integer number, which specifies the level at which the node lives and can interact with other nodes. To operate on levels during reduction, one adds to sharing graphs two additional kinds of nodes: *croissants*, which open or close a level, and *brackets*, which temporarily close a level or restore a temporarily closed one. Like fans, brackets need to be paired as well, and one needs additional structural constraints that we will not describe here.

Optimal reduction is not invariant. As we can see, the machinery of sharing graphs and optimal reduction is quite involved, in contrast to λ-graphs which represent λ-terms straightforwardly. In this dissertation however we do not strive for optimal reduction, because we are not interested in the efficiency of evaluation in the sense of finding the smartest or shortest evaluation strategy: we study the asymptotic complexity of evaluation through abstract machines, proving that their running time is proportional to the number of β-steps required by the fixed sequential strategy that the machine implements. In the case of optimal evaluation, instead, the number of parallel β steps—even thought less than the number of β steps required by any sequential strategy—does not provide an invariant cost model[7]. As shown by Asperti and Mairson, 1998, the definition of optimal reduction hides hyper-exponential computations: the cost of reducing a λ-term through an optimal reduction algorithm cannot be bound by any elementary function of the number of parallel β-steps.

Higher-order sharing equality. As a final remark, we point out that in part III we will study the problem of *sharing equality*, *i.e.* how to check whether two λ-graphs represent the same unshared λ-term. This problem for λ-graphs has a very low computational complexity, and in fact we provide a linear-time algorithm for checking sharing equality. On the contrary, the problem of sharing equality for sharing graphs has never been studied, and we doubt it is solvable even in polynomial time. The reason of our pessimism is in the phenomenon called *superposition*, the property of sharing graphs of representing succintly a super-exponential number of entirely different λ-terms (Asperti and Mairson, 1998).

[6] See *e.g.* Asperti and Guerrini, 1998.
[7] For "cost model", see page 61.

Part II

Computation

This part of the dissertation is about Computation, *i.e.* the efficient evaluation of λ-terms. We introduce crumbled forms, our alternative representation of λ-terms which decomposes nested applications. The focus of this part is on the impact of crumbled forms on the design and asymptotic overhead of abstract machines for CbV evaluation. An interesting feature of crumbling machines is that, unlike traditional abstract machines, the "mechanism" (*i.e.* the machine components and transitions) is somewhat decoupled from evaluation strategy (*e.g.* right-to-left *vs* left-to-right). As we will see, crumbled forms also induce abstract machines with less data structures and less transitions, and the crumbling transformation does not introduce any asymptotic overhead. Moreover, these facts smoothly scale up to open terms, a more delicate setting as discussed in section 5.2.

Why studying crumbled forms. As already mentioned in the introduction, our ultimate goal is to provide an abstract machine for Strong CbV evaluation, but as discussed in chapter 5 strong evaluation is much harder to implement than closed or open evaluation, expecially with respect to strategies like CbV and CbNeed. One of the critical points is that a strong machine requires further data structures, and managing these additional data structures leads to a proliferation of machine transitions. This makes it daunting to even formulate a tentative machine, let alone prove it correct or efficient.

We devised crumbled forms exactly to make the transition from Open CbV to Strong CbV feasible, and in fact we have promising results that through crumbled forms one can successfully formulate an abstract machine for Strong CbV evaluation. We present our candidate machine for Strong CbV in the conclusions (section 15.1) but leave its full technical development for future research.

This part of the dissertation, instead, succeeds in exploring the subtleties of CbV in frameworks that are well-understood, such as the closed and open cases, and show that there is no slowdown in turning to a crumbled representation.

Conference paper. This part covers the results published in the article (Accattoli et al., 2019a), whose full proofs can be found in the accompanying technical report (Accattoli et al., 2019b). In these papers we actually discuss a slightly more general case, *i.e.* crumbled forms extended with *booleans* and the `if` $\bullet$ `then` $\bullet$ `else` $\bullet$ construct, so to show that our approach scales painlessly to calculi that bear more similarities with programming languages and proof assistants. In this dissertation however we decided to present a version of these results simplified to the pure λ-calculus, because we believe that the slight complications caused by these constructs are not particularly informative, if not of the fact that crumbled forms are very flexible. Please refer to the aforementioned articles for the more general case.

Outline. This part is structured as follows:

- Chapter 7 is an introductory chapter that also provides the intuitions behind crumbling. First, we define the syntax of crumbled forms, together with a translation function $\underline{\bullet}$ from λ-terms and a readback $\bullet_{\downarrow}$. We provide an operational semantics by introducing

crumbled evaluation contexts, and prove some preliminary results that will be required in the next chapters. Moreover, we compare crumbled forms to already existing transformations, *i.e. administrative normal forms* and *continuation-passing style* transformations. In addition, we clarify a point raised by Kennedy, about a potential inefficiency of such representations: our approach does not suffer from Kennedy's slowdown.

- In chapter 8 we introduce an abstract machine evaluating crumbled forms, which we call Crumble GLAM, and show that it implements Plotkin's CbV calculus λ_{Plot} defined in section 3.1. Moreover, we study the overhead of the machine, and show that it is linear in the number of β-steps and in the size of the initial term, exactly as the best machines for CbV executing ordinary λ-terms. Therefore, the crumbling transformation does not introduce any asymptotic overhead. The study is detailed and based on a careful and delicate spelling of the invariants of the machine.

- In chapter 9 we lift crumbled forms and environments to the open case. The crumbling technique smoothly scales up to Open CbV: we provide an abstract machine, the Open Crumble GLAM, implementing the fireball calculus λ_{fire} defined in section 3.2, and we show that it only has a linear overhead as in the closed case. Two aspects of this study are worth to be pointed out. First, the technical development follows almost identically the one for the closed case, once the subtler invariants of the new machine have been found. Second, the *substitution of abstractions on demand*, a technical optimisation typical of open/strong cases that we have discussed on page 72 becomes superfluous, as it is smoothly subsumed by the crumbling transformation.

- In chapter 10 we introduce variants of the Crumble GLAMs from previous chapters, called Pointed Crumble GLAMs. These machines refine the ones already provided in the previous chapter by also spelling out the search transitions, necessary to consider them proper abstract machines; this is obtained by enriching crumbled environment with an additional pointer, that indicates the environment entry that is currently being evaluated.

- Lastly, we provide in chapter 11 an OCaml implementation of the Pointed Crumble GLAMs, together with a discussion of the data structures required to respect the compexity analyses carried in the previous chapters.

Chapter 7

Crumbled CbV

Adding explicit sharing to the λ-calculus (see section 4.2) enables a special representation of terms where every term constructor is associated with a new sharing point. Such a special form — that is the topic of this part — is roughly obtained by (recursively) decomposing iterated applications by introducing an ES in between any two of them. For instance:

Example 7.1 \ Crumbling

Take the λ-term $((\lambda x.x\,(xx))\,y)\,((\lambda z.z)\,y)\,y$. Its crumbling is:

$$\alpha y\,[\alpha{\leftarrow}\beta\gamma]\,[\beta{\leftarrow}(\lambda x.\,x\delta\,[\delta{\leftarrow}xx])\,y]\,[\gamma{\leftarrow}(\lambda z.z)\,y]$$

where—for the sake of readability—we used the greek letters $\alpha, \beta, \gamma, \delta$ to denote the new sharing variables introduced during crumbling.

A few observations:

- The crumbling transformation affects function bodies: for example the abstraction $\lambda x.x(xx)$ turns into $\lambda x.\,x\delta[\delta{\leftarrow}xx]$.
- ES are grouped together in environments, unless forbidden by abstractions.
- ES are flattened out, *i.e.* they are not nested unless nesting is forced by abstractions.

This chapter is devoted to the study of such a representation, which we call crumbled as it "disintegrates" a λ-term by means of ES. Our *crumbling transformation* closely resembles—while not being exactly the same—the transformation into *a(dministrative) normal form* (ANF), introduced by Flanagan et al., 1993 (building on previous work by Sabry and Felleisen, 1993), itself a variant of the *continuation-passing style* (CPS) transformation. We discuss similarities and differences in section 7.4.

A delicate point is to preserve crumbled forms during evaluation. ES often come together with *commutation* rules to move them around the term structure, like the ones shown in fig. 4.3. These rules are used to unveal redexes during evaluation or to preserve specific syntactic forms, but they may introduce significant overhead that, if not handled carefully, can

even lead to asymptotic slowdowns as shown by Kennedy, 2007. One of our contributions is to show that crumbled forms can be evaluated and preserved with no need of commutation rules, therefore avoiding Kennedy's potential slowdown (again, see section 7.4).

7.1 Crumbled Environments

In section 4.3 we showed how abstract machines typically rely on environments. Crumbled forms also rely on packing ES together, as pointed out before, but depart from the ordinary case because environments may appear also under abstractions.

The notion of environment induced by crumbled forms, named here *crumbled environments*, is particularly interesting. Crumbled environments indeed play a double role: they both store delayed substitutions, as also do ordinary environments, but also *encode evaluation contexts*. In ordinary abstract machines, the evaluation context is usually stored in data structures such as the *stack* and the *dump*: roughly, they implement the search for the redex in the ordinary applicative structure of λ-terms. For crumbled forms, the evaluation context is encoded directly in the crumbled environment, and so the other structures disappear.

Operations on crumbled environments. There are two subtle implementative aspects of crumbled environments, that set them apart from ordinary ones.

1. Ordinary environments are presented with a sequential structure but they are only accessed randomly (that is, not sequentially)—in other words, their sequential structure does not play a role. Crumbled environments, as the ordinary ones, are accessed randomly, to retrieve delayed substitutions, but they are also explored sequentially—since they encode evaluation contexts—in order to search for redexes. Therefore, their implementation has to reflect the sequential structure.

2. The second subtlety is that crumbled machines also have to concatenate environments, an operation never performed by ordinary machines. Concatation has to be concretely implemented as efficiently as possible, *i.e.* in constant time: the mentioned slowdown of Kennedy, 2007 is due exactly to the concatenation of environments, and as we will see it amounts to a quadratic overhead in evaluating terms in ANF.

To address these points, we provide in chapter 11 a prototype OCaml implementation of crumbled environments, that can be compared with the one of global environments by Accattoli and Barras, 2017 that does not concretely implement the sequential structure. In particular, our implementation concatenates environments in constant time and does not suffer from Kennedy's slowdown: essentially, Kennedy's slowdown amounts to the fact that his implementation concatenates ANF environments in linear rather than constant time (see section 7.4); this improvement is one of the contributions of this work.

7.2 Introducing Crumbled Environments

In this section we provide an informal explanation of the crumbling transformation $\underline{\bullet}$ of λ-terms into crumbles; the formal definition is in the next section.

Decomposing applications. The idea is to forbid iterated applications without loosing expressive power. To do so, one can write terms such as $(ts)u$ or $t(su)$ respectively as $(\lambda x.(xu))(ts)$ and $(\lambda x.(tx))(su)$ where x is a fresh variable. It is usually preferred to use let expressions rather than introducing β-redexes, so that one would rather write $\texttt{let}\ x = ts\ \texttt{in}\ (xu)$ and $\texttt{let}\ x = su\ \texttt{in}\ (tx)$, or, with explicit substitutions (a.k.a environment entries), rather write:

$$(xu)[x\leftarrow ts] \quad \text{and} \quad (tx)[x\leftarrow su].$$

If the crumbling transformation $\underline{\bullet}$ is applied to the whole λ-term—recursively on t, s, and u in our examples—then all applications have the form vv', *i.e.* they only involve values. If moreover CbV evaluation is adopted, then such a crumbled form is stable by evaluation, as variables can only be replaced by values.

Simulation and no evaluation contexts. Let us now have a look at a slightly bigger example and discuss the recursive part of the crumbling transformation. Let $I = \lambda x.x$ be the identity and consider the term $t := ((\lambda y.yy)\, I)\, ((II)\, I)$ whose right-to-left CbV evaluation is:

$$t \to_{\beta/v} ((\lambda y.yy)\, I)\, (II) \to_{\beta/v} ((\lambda y.yy)\, I)\, I \to_{\beta/v} (II)\, I \to_{\beta/v} II \to_{\beta/v} I.$$

The crumbling transformation decomposes all applications, taking special care of grouping all the environment entries together, flattening them out (that is, avoiding to have them nested one into the other), and reflecting the evaluation order in the arrangement of the environment. For instance, the crumbled representation $\underline{t}$ of the λ-term t above is

$$\underline{t} = (wz)[w\leftarrow(\lambda y.yy)I][z\leftarrow xI][x\leftarrow II],$$

and evaluation takes always place at the far right of the environment, as follows:

Example 7.2 \ Example of crumbled small-step evaluation

$$\begin{array}{lll}
\underline{t} & \to_{\beta/v} & (wz)[w\leftarrow(\lambda y.yy)I][z\leftarrow xI][x\leftarrow \underline{I}] \\
& \to_{\beta/v} & (wz)[w\leftarrow(\lambda y.yy)I][z\leftarrow \underline{I}] \\
& \to_{\beta/v} & (wI)[w\leftarrow \underline{II}] \\
& \to_{\mathtt{sub}} & \underline{II}
\end{array}
\qquad
\begin{array}{ll}
\to_{\mathtt{sub}} & (wz)[w\leftarrow(\lambda y.yy)I][z\leftarrow \underline{II}] \\
\to_{\mathtt{sub}} & (wI)[w\leftarrow \underline{(\lambda y.yy)\ I}] \\
\to_{\beta/v} & (wI)[w\leftarrow \underline{I}] \\
\to_{\beta/v} & I
\end{array}$$

In example 7.2 $\to_{\beta/v}$ steps correspond exactly to steps in the ordinary evaluation of t and $\to_{\mathtt{sub}}$ steps simply eliminate the explicit substitution when its content is a value. Note how the transformation makes the redex always appear at the end of the environment, so that the need for searching for it—together with the notion of evaluation context—disappears.

Let us also introduce some terminology. Values and applications of values are *crumbled terms*. The transformation, called *crumbling translation*, turns a λ-term into a crumbled term plus an environment—such a pair is called a crumble.

Turning to Micro-Step Evaluation. Example 7.2 covers what happens when the crumbling transformation is paired with small-step evaluation. Abstract machines, however, employ the finer mechanism of micro-step evaluation (the difference is discussed in section 4.3), where the substitutions due to β-redexes are delayed and represented as new environment entries, and moreover substitution is decomposed as to act on one variable occurrence at a time. In particular, such a more parsimonious evaluation never removes environment entries because they might be useful later on—garbage collection is assumed to be an orthogonal and independent process. To give an idea of how micro steps work in this setting, let us focus on the evaluation of the subterm $t' := (wz)[w\leftarrow(\lambda y.yy)I]$ of example 7.2 (because micro-step evaluations are long and tedious), that proceeds as follows:

Example 7.3 \ Example of crumbled micro-step evaluation

$$\begin{array}{lllll} t' & \to_{\beta/v} & (wz)[w\leftarrow yy][y\leftarrow I] & \to_{\mathtt{lsub}} & (wz)[w\leftarrow yI][y\leftarrow I] \\ & \to_{\mathtt{lsub}} & (wz)[w\leftarrow II][y\leftarrow I] & \to_{\beta/v} & (wz)[w\leftarrow x][x\leftarrow I][y\leftarrow I] \\ & \to_{\mathtt{lsub}} & (wz)[w\leftarrow I][x\leftarrow I][y\leftarrow I] & \to_{\mathtt{lsub}} & (Iz)[w\leftarrow I][x\leftarrow I][y\leftarrow I] \end{array}$$

Note that in example 7.3 now $\to_{\beta/v}$ steps introduce new environment entries, and also that the redex is not always at the end of the environment, but it is always followed on the right by an environment whose entries are all abstractions, so that the search for the next redex simply becomes a straightforward sequential visit from right to left of the environment—the evaluation context has been coded inside the sequential structure of the environment.

Abstraction Bodies and the Concatenation of Environments. There is a last point to explain. We adopt weak evaluation, but the crumbling transformation is in a way *strong*, as it also transforms the bodies of abstractions into crumbles. Let us see another example. The crumbled representation of $s := (\lambda x.((xx)(xx)))(II)$ then is:

$$\underline{s} = ((\lambda x.((yz)[y\leftarrow xx][z\leftarrow xx]))w)[w\leftarrow II].$$

Micro-step evaluation goes as follows:

$$\begin{array}{lll} \underline{s} & \to_{\beta/v} & ((\lambda x.((yz)[y\leftarrow xx][z\leftarrow xx]))w)[w\leftarrow w'][w'\leftarrow I] \\ & \to_{\mathtt{lsub}} & ((\lambda x.((yz)[y\leftarrow xx][z\leftarrow xx]))w)[w\leftarrow I][w'\leftarrow I] \\ & \to_{\mathtt{lsub}} & ((\lambda x.((yz)[y\leftarrow xx][z\leftarrow xx]))I)[w\leftarrow I][w'\leftarrow I]. \end{array}$$

At this point, the reduction of the β-redex (involving λx) has to combine the crumble of the redex itself with the body of the abstraction, by concatenating the environment of the former (here $[w\leftarrow I][w'\leftarrow I]$) at the end of the environment of the latter ($[y\leftarrow xx][z\leftarrow xx]$), interposing the entry created by the redex itself $[x\leftarrow I]$, thus producing the new crumble:

$$(yz)[y\leftarrow xx][z\leftarrow xx][x\leftarrow I][w\leftarrow I][w'\leftarrow I].$$

The key conclusion is that evaluation needs to *concatenate* crumbled environments, which is an operation that ordinary abstract machines instead never perform.

7.3 The Crumbling Transformation

Now that we have the necessary intuitions about crumbling, we are ready to define formally the language of crumbled forms and the crumbling transformation.

Crumbled forms. The syntax of crumbled forms is in fig. 7.1. Basically, λ-terms are replaced by *crumbles*, which are pairs of a bite and an environment, where in turn:

- a *crumbled term* is either a *crumbled value* (*i.e.* a variable or an abstraction over a crumble) or an application of crumbled values, and
- an *environment* is a finite sequence of explicit substitutions of crumbled terms for variables.

$$\text{Bites } b ::= v \mid vw \qquad \text{Crumbled values } v, w ::= x \mid \lambda x.c$$
$$\text{Environments } e ::= [x_1 \leftarrow b_1] \cdots [x_k \leftarrow b_k] \qquad \text{Crumbles } c, d ::= (b, e)$$

Crumbled forms / **Figure 7.1**

Note that:

- By a slight abuse of notation, we denote crumbled values with the same letters that denote values in the λ-calculus. Nevertheless, the two are very different syntactic categories — see next point.
- *Abstractions*: the body of abstractions is itself a crumble—the forthcoming crumbling transformation is indeed "strong", as it also transforms the body of abstractions.
- *Crumbles are not closures*: the definition of crumbles may remind the one of *closures* in abstract machines with local environments (cf. page 48), but the two concepts are different. The environment e of a crumble (b, e), indeed, does not in general bind all the free variables of the crumbled term b.

As one can see, the syntax for environments is slightly different than the one we provided previously (cf. fig. 4.4): in this part we will freely consider environments as lists extendable both on the left and on the right, and whose concatenation is obtained by simple juxtaposition. As usual, we denote by ϵ the empty environment. We introduce the explicit operation of appending "@", but only use it between crumbles and environments:

Definition 7.4 \ Appending @

Let (b, e) be a crumble, and e' be a crumbled environment. The *appending* of e' to (b, e) is defined as:

$$(b, e) \mathbin{@} e' \coloneqq (b, ee').$$

Free variables, α-renaming, and all that. All syntactic expressions are *not* considered up to α-equivalence. Free variables of crumbled forms are defined via the auxiliary notion of *domain* of environments (this is motivated by the fact that global environment are used here):

Definition 7.5 \ Variables of crumbled forms

We extend the notions from chapter 2 of free variables $\mathsf{fv}(\bullet)$ and domain $\mathsf{dom}(\bullet)$ to crumbled forms, as follows:

$$\mathsf{fv}(x) \coloneqq \{x\} \qquad \mathsf{fv}(vw) \coloneqq \mathsf{fv}(v) \cup \mathsf{fv}(w) \qquad \mathsf{fv}(\lambda x.c) \coloneqq \mathsf{fv}(c) \setminus \{x\}$$

$$\begin{aligned} \mathsf{fv}(\epsilon) &\coloneqq \emptyset & \mathsf{fv}(e[x \leftarrow b]) &\coloneqq \mathsf{fv}(e) \setminus \{x\} \cup \mathsf{fv}(b) \\ \mathsf{dom}(\epsilon) &\coloneqq \emptyset & \mathsf{dom}(e[x \leftarrow b]) &\coloneqq \mathsf{dom}(e) \cup \{x\} \end{aligned}$$

$$\mathsf{dom}(b, e) \coloneqq \mathsf{dom}(e) \qquad \mathsf{fv}(b, e) \coloneqq \mathsf{fv}(b) \setminus \mathsf{dom}(e) \cup \mathsf{fv}(e)$$

As for all abstract machines based on global environments, we require that crumbled forms are *well-named* to avoid variable shadowing:

Definition 7.6 \ Well-named

Let $e = [x_1 \leftarrow b_1] \cdots [x_k \leftarrow b_k]$ be an environment: we say that e is *well-named* if the x_i's are all pairwise distinct. A crumble (b, e) is *well-named* if e is well-named.

A consequence of well-namedness is that the lookup of a variable in a crumbled environment is well-defined:

Definition 7.7 \ Lookup in a crumbled environment

Let $e = [x_1 \leftarrow b_1] \cdots [x_k \leftarrow b_k]$ be a well-named environment. We denote the lookup of x_i in e by $e(x_i) \coloneqq b_i$. If $y \notin \{x_1, \dots, x_k\}$, we say that $e(y)$ is undefined.

Crumbling. We are now ready to provide the formal definition of the crumbling translation of λ-terms to crumbled forms:

Definition 7.8 \ Crumbling translation $\underline{t}$

We transform a λ-term into a crumbled term via the following *crumbling translation* $\underline{\bullet}$, and using an auxiliary translation $\overline{\bullet}$:

$$\begin{array}{rcl@{\qquad}rcl} \underline{x} & := & (x,\epsilon) & \overline{x} & := & (x,\epsilon) \\ \underline{\lambda x.t} & := & (\lambda x.\underline{t},\epsilon) & \overline{\lambda x.t} & := & (\lambda x.\underline{t},\epsilon) \\ \underline{su} & := & (vv', ee') & \overline{su} & := & (z, [z{\leftarrow}vv']ee') \quad (*) \end{array}$$

$(*)$ where $\overline{s} =: (v,e)$ and $\overline{u} =: (v',e')$, and z is globally fresh.

In the definition above, $\overline{\bullet}$ is an auxiliary translation function that handles separately the case of nested applications. In fact, translating a value or the application of two values simply results in a crumble with empty environment, since these terms do not contain nested applications at toplevel – note however that the translation of an abstraction still performs the translation recursively inside its body. The translation of an arbitrary nested application su is more involved: when for instance u is an application too, the auxiliary translation generates a fresh variable z to act as a sharing point in place of u.

To clarify definition 7.8, let us have a look at the following example:

Example 7.9 \ Crumbling

Let $\delta := \lambda x.xx$ and $I := \lambda x.x$. Then:

$$\begin{aligned} \underline{\delta\delta I} &= (zI_b, [z{\leftarrow}\delta_b\delta_b]) \\ &= (z\,(\lambda x.x,\epsilon), [z{\leftarrow}(\lambda x.xx,\epsilon)(\lambda x.xx,\epsilon)]) \end{aligned}$$

$$\begin{aligned} \underline{\delta\delta(xx)} &= (zw, [z{\leftarrow}\delta_b\delta_b][w{\leftarrow}xx]) \\ &= (zw, [z{\leftarrow}(\lambda x.xx,\epsilon)(\lambda x.xx,\epsilon)][w{\leftarrow}xx]). \end{aligned}$$

where

- $I_b := \lambda x.\underline{x} = \lambda x.(x,\epsilon)$,
- $\delta_b := \lambda x.\underline{xx} = \lambda x.(xx,\epsilon)$,
- $\overline{\delta\delta} = (z, [z{\leftarrow}\delta_b\delta_b]) = (z, [z{\leftarrow}(\lambda x.xx,\epsilon)(\lambda x.xx,\epsilon)])$,
- $\overline{xx} = (w, [w{\leftarrow}xx])$.

Note that the crumbling translation $\underline{\bullet}$ is not surjective, *i.e.* not every crumble is the translation of a λ-term. For instance, the crumble $c := (xx, [x{\leftarrow}y])$ is not in the image of the translation. This is a consequence of two facts:

1. The fresh variables introduced by the crumbling translation are *linear*, *i.e.* they occur only once in the translated crumble; in the crumble c provided above, instead, the variable x occurs twice. Linearity is going to play a major role in the proof of an essential

property of our abstract machines, namely the "contextual decoding" one (see proposition 7.26/5).

2. The crumbling environments induced by the crumbling translation do not contain "renaming" entries of the form $[x \leftarrow y]$. This property does not have direct consequences on the correctness of our crumbling machines, but will be crucial in the complexity analyses of sections 8.3 and 9.3.

There is however a left inverse for the crumbling translation from λ-terms to crumbles, called readback and defined as follows:

Definition 7.10 \ Readback to λ-terms $c_\downarrow$

We define the readback $\bullet_\downarrow$ of bites and crumbles, by structural induction:

$$x_\downarrow := x \qquad (\lambda x.c)_\downarrow := \lambda x.c_\downarrow \qquad (vw)_\downarrow := v_\downarrow w_\downarrow$$
$$(b, \epsilon)_\downarrow := b_\downarrow \qquad (b, e[x \leftarrow b'])_\downarrow := (b, e)_\downarrow \{x \leftarrow b'_\downarrow\}$$

In order to prove that $\bullet_\downarrow$ is a left inverse of $\underline{\bullet}$ (see proposition 7.17), we first need a number of intermediate lemmas.

First, two structural properties of the readback:

Proposition 7.11 \ Readback of application

$(vw, e)_\downarrow = (v, e)_\downarrow (w, e)_\downarrow$ for every crumbled environment e and crumbled values v, w.

Proof. We proceed by induction on the structure of e:

- If $e = \epsilon$, then $(vw, e)_\downarrow = vw_\downarrow = v_\downarrow w_\downarrow = (v, \epsilon)_\downarrow (w, \epsilon)_\downarrow$.
- If $e = e'[x \leftarrow b]$, then $(vw, e)_\downarrow = (vw, e')_\downarrow \{x \leftarrow b_\downarrow\}$. By *i.h.* $(vw, e')_\downarrow = (v, e')_\downarrow (w, e')_\downarrow$, and therefore $(vw, e')_\downarrow \{x \leftarrow b_\downarrow\} = ((v, e')_\downarrow (w, e')_\downarrow)\{x \leftarrow b_\downarrow\} = (v, e')_\downarrow \{x \leftarrow b_\downarrow\} (w, e')_\downarrow \{x \leftarrow b_\downarrow\} = (v, e)_\downarrow (w, e)_\downarrow$.

Proposition 7.12 \ Readback of append

Let c, d be crumbles such that $c_\downarrow = d_\downarrow$. Then $(c \mathbin{@} e)_\downarrow = (d \mathbin{@} e)_\downarrow$ for every environment e.

Proof. By induction on e:

- if $e := \epsilon$, then we conclude because $c \mathbin{@} e = c$ and $d \mathbin{@} e = d$;
- if $e := e'[x \leftarrow b]$, by *i.h.* $(c \mathbin{@} e')_\downarrow = (d \mathbin{@} e')_\downarrow$, and we conclude because $(c \mathbin{@} e)_\downarrow = (c \mathbin{@} e')_\downarrow \{x \leftarrow b_\downarrow\}$ and $(d \mathbin{@} e)_\downarrow = (d \mathbin{@} e')_\downarrow \{x \leftarrow b_\downarrow\}$.

The following remark simply states some evident properties of variables and the crumbling translation:

Proposition 7.13 \ Crumbling translation, readback, free variables, and values

For any λ-term t, any λ-value v, any bite b, and any crumble c:

1. $\mathsf{fv}(\underline{t}) = \mathsf{fv}(t)$; in particular, t is closed if and only if $\underline{t}$ is so.
2. $\mathsf{fv}(c_{\downarrow}) \subseteq \mathsf{fv}(c)$.
3. The crumbling translation commutes with the renaming of free variables.
4. The readback maps crumbled values to values.

Proof.

1. We proceed by induction on the structue of t:
   - If $t = x$, then $\underline{t} = (x, \epsilon)$ and clearly $\mathsf{fv}((x, \epsilon)) = \{x\} = \mathsf{fv}(x)$.
   - If $t = \lambda x.s$, then $\underline{t} = (\lambda x.\underline{s}, \epsilon)$ and so $\mathsf{fv}(\lambda x.\underline{s}, \epsilon) = \mathsf{fv}(\underline{s}) \setminus \{x\} = \mathsf{fv}(s) \setminus \{x\}$ by *i.h.*
   - If $t = su$, then then $\overline{t} = (vw, ee')$ where $(v, e) := \overline{s}$ and $(w, e') := \overline{u}$. By cases on s and u:
     - If s and u are both values, then $e = e' = \epsilon$, $\underline{t} = (vw, \epsilon)$ and $\overline{t} = (z, [z \leftarrow vw])$. Then $\mathsf{fv}(\underline{t}) = \mathsf{fv}(v) \cup \mathsf{fv}(w) = \mathsf{fv}(s_{\downarrow}) \cup \mathsf{fv}(u_{\downarrow})$ by *i.h.*
     - If s and u are both applications, then $e, e' \neq \epsilon$ and $x \in \mathsf{dom}(e)$ and $y \in \mathsf{dom}(e')$. By *i.h.* $\mathsf{fv}(e) = \mathsf{fv}(s)$ and $\mathsf{fv}(e') = \mathsf{fv}(u)$.
       Note that, by the definition of $\mathsf{fv}(\bullet)$, $\mathsf{fv}(vw, ee') = (\mathsf{fv}(v) \cup \mathsf{fv}(w) \setminus \mathsf{dom}(e) \cup \mathsf{fv}(e)) \setminus \mathsf{dom}(e') \cup \mathsf{fv}(e')$. By *i.h.* $(\mathsf{fv}(v) \cup \mathsf{fv}(w) \setminus \mathsf{dom}(e) \cup \mathsf{fv}(e)) \setminus \mathsf{dom}(e') \cup \mathsf{fv}(e') = (\mathsf{fv}(v) \cup \mathsf{fv}(w) \setminus \mathsf{dom}(e) \cup \mathsf{fv}(s)) \setminus \mathsf{dom}(e') \cup \mathsf{fv}(u)$, which is simply $\mathsf{fv}(s) \cup \mathsf{fv}(u)$ because $\mathsf{dom}(e)$ and $\mathsf{dom}(e')$ are disjoint since the crumbling variables are choosen as globally fresh during crumbling, and $s \perp e'$ for the same reason, because $\mathsf{dom}(e')$ are fresh crumbling variables.
     - The cases when only one among s and u is an application are similar to the two cases proved above.
2. We prove at the same time the corresponding statement for bites, *i.e.* that $\mathsf{fv}(b_{\downarrow}) \subseteq \mathsf{fv}(b)$. By induction on the structure of b:
   - If $b = x$, then $\mathsf{fv}(x_{\downarrow}) = \mathsf{fv}(x) = \{x\}$.
   - If $b = \lambda x.e$, then $\mathsf{fv}((\lambda x.e)_{\downarrow}) = \mathsf{fv}(\lambda x.e_{\downarrow}) = \mathsf{fv}(e_{\downarrow}) \setminus \{x\}$. By *i.h.* $\mathsf{fv}(e_{\downarrow}) \subseteq \mathsf{fv}(e)$, and we conclude.
   - If $b = vw$, then $\mathsf{fv}((vw)_{\downarrow}) = \mathsf{fv}(v_{\downarrow}) \cup \mathsf{fv}(w_{\downarrow}) \subseteq \mathsf{fv}(v) \cup \mathsf{fv}(w)$ by *i.h.*

   Let now $c = (b, e)$. We proceed by induction on the structure of e:
   - If $e = \epsilon$, then $\mathsf{fv}(c_{\downarrow}) = \mathsf{fv}(b_{\downarrow})$ and $\mathsf{fv}(c) = \mathsf{fv}(b)$. Conclude by *i.h.*

- If $e = e'[x\leftarrow b']$, then $c_\downarrow = (b, e')_\downarrow\{x\leftarrow b'_\downarrow\}$. By the properties of substitution, $\mathsf{fv}((b, e')_\downarrow\{x\leftarrow b'_\downarrow\}) \subseteq \mathsf{fv}((b, e')_\downarrow) \setminus \{x\} \cup \mathsf{fv}(b'_\downarrow)$. By *i.h.* $\mathsf{fv}((b, e')_\downarrow) \subseteq \mathsf{fv}((b, e'))$ and $\mathsf{fv}(b'_\downarrow) \subseteq \mathsf{fv}(b')$, and we conclude because $\mathsf{fv}(c) = \mathsf{fv}((b, e')) \setminus \{x\} \cup \mathsf{fv}(b')$ by definition.

3. Obvious because the translation does not distinguish between free variables.

4. Follows trivially from the definition of readback. ∎

Disjointedness. We introduce the new notion of *disjoint forms*, noted with the symbol $\perp$ and fundamental to prove that readbacks of crumbled forms are somewhat independent when their variables are also disjoint (see theorem 7.25).

To define $\perp$ uniformly over various kinds of crumbled objects, we use the generic symbol P in the definition below to denote any kind of crumbled form introduced so far:

Definition 7.14 \ Disjoint forms $\perp$

Let P and P' be two crumbles or environments (contexts): P and P' are *disjoint*, in symbols $P \perp P'$, if $\mathsf{fv}(P) \cap \mathsf{dom}(P') = \emptyset$.

The following proposition further clarifies the aim of $\perp$:

Proposition 7.15 \ Readback *vs* disjointedness

For every crumble c and environment e: if $c \perp e$, then $(c \mathbin{@} e)_\downarrow = c_\downarrow$.

Proof. By structural induction on e:

- If $e := \epsilon$ then $c \mathbin{@} e = c$ and hence $(c \mathbin{@} e)_\downarrow = c_\downarrow$.
- Otherwise $e := e'[x\leftarrow b]$. By *i.h.*— which can be applied since $c \perp e$ implies $c \perp e'$, because $\mathsf{dom}(e') \subseteq \mathsf{dom}(e) - (c \mathbin{@} e')_\downarrow = c_\downarrow$. From $c \perp e$ it follows that $x \notin \mathsf{fv}(c) \supseteq \mathsf{fv}(c_\downarrow)$ (by proposition 7.13), so $c_\downarrow\{x\leftarrow b_\downarrow\} = c_\downarrow$. Therefore, $(c \mathbin{@} e)_\downarrow = c \mathbin{@} e'_\downarrow\{x\leftarrow b_\downarrow\} = c_\downarrow\{x\leftarrow b_\downarrow\} = c_\downarrow$. ∎

The following two propositions are auxiliary properties that relate the readbacks of disjoint forms, and are necessary to prove the important theorem 7.25 below.

Proposition 7.16 \ Decompositions of readback

Let c be a crumble, b be a bite, $e.e'$ be environments, and x be a variable such that $x \notin \mathsf{dom}(e)$. Then:

1. $(c \mathbin{@} [x\leftarrow b]e)_\downarrow = (c \mathbin{@} e)_\downarrow\{x\leftarrow(b, e)_\downarrow\}$ when also $x \notin \mathsf{fv}(e)$.

2. $(c \mathbin{@} [x{\leftarrow}b]e)_\downarrow = c_\downarrow\{x{\leftarrow}(b,e)_\downarrow\}$ when also $x \notin \mathsf{fv}(e)$ and $c \perp e$.
3. $(x, e[x{\leftarrow}b]e')_\downarrow = (b,e')_\downarrow$.

Proof. 1. Let $c = (b', e')$. We proceed by structural induction on e:

- If $e = \epsilon$, then $(b,e)_\downarrow = b_\downarrow$ and $(b', e'[x{\leftarrow}b]e)_\downarrow = (b', e'[x{\leftarrow}b])_\downarrow = (b',e')_\downarrow\{x{\leftarrow}b_\downarrow\} = (b', e'e)_\downarrow\{x{\leftarrow}(b,e)_\downarrow\}$.
- Otherwise $e = e''[z{\leftarrow}b'']$ and then $(b,e)_\downarrow = (b,e'')_\downarrow\{z{\leftarrow}b''_\downarrow\}$, so

$$\begin{aligned}(b', e'[x{\leftarrow}b']e)_\downarrow &= (b', e'[x{\leftarrow}b']e''[z{\leftarrow}b''])_\downarrow \\ &= (b', e'[x{\leftarrow}b']e'')_\downarrow\{z{\leftarrow}b''_\downarrow\} \\ &= (b', e'e'')_\downarrow\{x{\leftarrow}(b',e'')_\downarrow\}\{z{\leftarrow}b''_\downarrow\} && (1.)\\ &= (b', e'e'')_\downarrow\{z{\leftarrow}b''_\downarrow\}\{x{\leftarrow}(b',e'')_\downarrow\{z{\leftarrow}b''_\downarrow\}\} && (2.)\\ &= (b', e'e)_\downarrow\{x{\leftarrow}(b',e)_\downarrow\}\end{aligned}$$

where

(a) by *i.h.*, where $x \notin \mathsf{dom}(e'') \cup \mathsf{fv}(e'')$ follows from the hypothesis $x \notin \mathsf{dom}(e) \cup \mathsf{fv}(e)$ because $\mathsf{dom}(e'') \subseteq \mathsf{dom}(e)$ and $\mathsf{fv}(e) = \mathsf{fv}(e'') \setminus \{z\} \cup \mathsf{fv}(b'')$.

(b) because $x \notin \mathsf{fv}(b''_\downarrow) \subseteq \mathsf{fv}(b'')$ by proposition 7.13 and the hypothesis that $x \notin \mathsf{fv}(e)$.

2. According to point 1,

$$(c \mathbin{@} [x{\leftarrow}b]e)_\downarrow = (c \mathbin{@} e)_\downarrow\{x{\leftarrow}(b,e)_\downarrow\} = c_\downarrow\{x{\leftarrow}(b,e)_\downarrow\}$$

where the last equality holds by proposition 7.15, since $c \perp e$.

3. By proposition 7.15, $(x,e)_\downarrow = (x,\epsilon)_\downarrow$ because $x \notin \mathsf{dom}(e)$. By proposition 7.12, $(x, e[x{\leftarrow}b])_\downarrow = (x, [x{\leftarrow}b])_\downarrow = x\{x{\leftarrow}b_\downarrow\} = b_\downarrow = (b,\epsilon)_\downarrow$. Again by proposition 7.12, $(x, e[x{\leftarrow}b]e')_\downarrow = (b,e')_\downarrow$.

Note that proposition 7.16/2 does not hold without the hypothesis $c \perp e$. Indeed, take $c := (y, \epsilon)$ and $e := [y{\leftarrow}zz]$ with $x \neq y$: for any term b, one has $(c \mathbin{@} [x{\leftarrow}b]e)_\downarrow = zz \neq y = c_\downarrow\{x{\leftarrow}(b,e)_\downarrow\}$.

As anticipated, we now prove that $\bullet_\downarrow$ is a left inverse of $\underline{\bullet}$:

Proposition 7.17 \ Readback is a left inverse for crumbling translation

$\underline{t}_\downarrow = t$ for every λ-term t.

Proof. We prove the required statement mutually with the corresponding statement for the auxiliary translation, *i.e.* that $\bar{t}_{\downarrow} = t$ for every λ-term t. We proceed by induction on the structure of t:

- *Variable, i.e.* $t := x$; then $\underline{t} = \bar{t} = (x, \epsilon)$, thus $\underline{t}_{\downarrow} = x = t$ and $\bar{t}_{\downarrow} = x = t$.
- *Abstraction, i.e.* $t := \lambda x.s$; then, $\underline{t} = \bar{t} = (\lambda x.\underline{s}, \epsilon)$. By *i.h.*, $\underline{s}_{\downarrow} = s$, hence $\underline{t}_{\downarrow} = \lambda x.\underline{s}_{\downarrow} = t$ and similarly $\bar{t}_{\downarrow}$.
- *Application, i.e.* $t = su$; then $\underline{t} = (vv', ee')$ and $\bar{t} = (z, [z \leftarrow vv']ee')$, where $(v, e) := \bar{s}$ and $(v', e') := \bar{u}$. By the definition of the crumbling translation, the variables in $\mathsf{dom}(e)$ and $\mathsf{dom}(e')$ are choosen in such a way to be globally fresh: it follows that $(v, e) \perp e'$, and $v' \perp e$. By proposition 7.11, $(vv', ee')_{\downarrow} = (v, ee')_{\downarrow}(v', ee')_{\downarrow}$. By proposition 7.15, $(v, ee')_{\downarrow} = (x, e)_{\downarrow}$ and $(v', ee')_{\downarrow} = (v', e')_{\downarrow}$. We conclude using the *i.h.*. The case for the auxiliary translation is similar, and it follows from proposition 7.16/3. ∎

Crumbled contexts. To evaluate crumbled forms we need two kinds of contexts, both for environments and crumbles:

Environment contexts $E ::= e[x \leftarrow \langle\cdot\rangle]$ Crumble contexts $C ::= \langle\cdot\rangle \mid (b, E)$

Crumbled contexts / **Figure 7.2**

Note that the environment contexts defined in fig. 7.2 are completely different from the environment contexts introduced for λ-calculi with ES on page 44, even though we denote them by the same letter E. We used the latter contexts to reduce a term in an environment, while we will use the former to reduce inside the rightmost ES of a crumbled environment. By means of C contexts, instead, we can locate a sub-crumble of a given crumble: either the whole crumble (with $\langle\cdot\rangle$) or a coda (with (b, E)).

Crumbles can be plugged into both notions of contexts. Let us point out that the following definition of plugging is slightly unusual as it does a little bit more than just replacing the hole, because simply replacing would not provide a well-formed syntactic object: plugging indeed extracts the environment from the plugged crumble and concatenates it with the environment of the context. Such an unusual operation—that may seem ad-hoc—is actually one of the key technical points in order to obtain a clean proof of the implementation theorem for Crumble GLAMs.

Definition 7.18 \ Plugging in crumbled contexts

Let $E = e[x\leftarrow\langle\cdot\rangle]$ be an environment context, C be a crumble context, and $c = (b', e')$ be a crumble. The *plugging* $E\langle c\rangle$ of c in E and the *plugging* $C\langle c\rangle$ of c in C are defined by

$$(e[x\leftarrow\langle\cdot\rangle])\langle(b', e')\rangle := e[x\leftarrow b']e' \qquad \langle\cdot\rangle\langle c\rangle := c \qquad (b, E)\langle c\rangle := (b, E\langle c\rangle).$$

Let us explain the definition of plugging by means of the following example:

Example 7.19 \ Plugging

In example 7.9 we have seen that $\underline{\delta\delta I} = (zI_b, [z\leftarrow\delta_b\delta_b])$ where $I_b := \lambda x.(x, \epsilon)$ and $\delta_b := \lambda x.(xx, \epsilon)$. Then, we have that $\underline{\delta\delta I} = C\langle c\rangle$ with $C := (zI_b, [z\leftarrow\langle\cdot\rangle])$ and $c := (\delta_b\delta_b, \epsilon)$.

The basic notions for crumbled forms can be naturally extended to crumbled contexts:

- The *appending* of an environment context E to a crumble (b, e') is defined as $(b, e')\ @\ E := (b, e'E)$.
- The notions of $\mathsf{fv}(\bullet)$ and $\mathsf{dom}(\bullet)$ are extended to crumble contexts by:

$$\begin{aligned} \mathsf{fv}(\langle\cdot\rangle) &:= \emptyset & \mathsf{fv}(b, e[x\leftarrow\langle\cdot\rangle]) &:= \mathsf{fv}(b, e) \setminus \{x\} \\ \mathsf{dom}(\langle\cdot\rangle) &:= \emptyset & \mathsf{dom}(b, e[x\leftarrow\langle\cdot\rangle]) &:= \mathsf{dom}(e) \cup \{x\} \end{aligned}$$

- An environment context $E := e[x\leftarrow\langle\cdot\rangle]$ is *well-named* if e is well-named and $x \notin \mathsf{dom}(e)$; a crumble context C is *well-named* if $C := \langle\cdot\rangle$ or $C := (b, E)$ and E is well-named.

We extend the definition of readback to crumble contexts as follows:

Definition 7.20 \ Readback of crumble contexts

Let C be a crumble context. We define its readback $C_\downarrow$ by cases:

$$\langle\cdot\rangle_\downarrow := \langle\cdot\rangle \qquad (b, e[x\leftarrow\langle\cdot\rangle])_\downarrow := (b, e)_\downarrow\{x\leftarrow\langle\cdot\rangle\}.$$

Note that the unfolding of a crumble context is not necessarily a context, because the hole may be duplicated or erased by the unfolding. For instance:

Example 7.21 \ Invalid context readback

Let $C := (x\,x, [x\leftarrow\langle\cdot\rangle])$. Then $C_\downarrow = \langle\cdot\rangle\langle\cdot\rangle$ is not a context.

However, the unfolding of contexts coming from the encoding of λ-terms is always a valid context: this property is called "contextual decoding", and we will prove it in proposition 7.26/5

together with other properties of crumbling. Now, instead, we provide a few technical propositions that are required by theorem 7.25, which relates readback with plugging and composition of contexts.

Proposition 7.22 \ Append inside plugging

For all crumble context C, crumble c, and environment e, one has $C\langle c\rangle \mathbin{@} e = C\langle c \mathbin{@} e\rangle$.

Proof. By cases according to the definition of the crumble context C. If $C \coloneqq \langle\cdot\rangle$, then $C\langle c\rangle \mathbin{@} e = c \mathbin{@} e = C\langle c \mathbin{@} e\rangle$. Otherwise $C \coloneqq (b, e'[x{\leftarrow}\langle\cdot\rangle])$; let $c \coloneqq (b', e'')$; then, $c \mathbin{@} e = (b', e''e)$ and hence $C\langle c\rangle \mathbin{@} e = (b, e'[x{\leftarrow}b']e'') \mathbin{@} e = (b, e'[x{\leftarrow}b']e''e) = (b, e'[x{\leftarrow}b']) \mathbin{@} e''e = C\langle c \mathbin{@} e\rangle$.

Proposition 7.23 \ Readback of appended entry

For every crumble c:

1. $c \mathbin{@} [x{\leftarrow}b]_{\downarrow} = c_{\downarrow}\{x{\leftarrow}b_{\downarrow}\}$.
2. $c \mathbin{@} [x{\leftarrow}\langle\cdot\rangle]_{\downarrow} = c_{\downarrow}\{x{\leftarrow}\langle\cdot\rangle\}$.

Proof. Let $c \coloneqq (b', e)$: then, $(b', e) \mathbin{@} [x{\leftarrow}b]_{\downarrow} = (b', e[x{\leftarrow}b])_{\downarrow} = (b', e)_{\downarrow}\{x{\leftarrow}b_{\downarrow}\}$. Similarly, $(b', e) \mathbin{@} [x{\leftarrow}\langle\cdot\rangle]_{\downarrow} = (b', e[x{\leftarrow}\langle\cdot\rangle])_{\downarrow} = (b', e)_{\downarrow}\{x{\leftarrow}\langle\cdot\rangle\}$.

Like contexts in the usual λ-calculus, crumbled contexts can be composed:

Definition 7.24 \ Composition of crumble contexts

It is also possible to plug a crumble context C' into another crumble context C, as follows:

$$C\langle C'\rangle \coloneqq \begin{cases} C & \text{if } C' = \langle\cdot\rangle \\ C' & \text{if } C = \langle\cdot\rangle \\ (b, e[x{\leftarrow}b']e'[y{\leftarrow}\langle\cdot\rangle]) & \text{if } C = (b, e[x{\leftarrow}\langle\cdot\rangle]) \text{ and } C' = (b', e'[y{\leftarrow}\langle\cdot\rangle]) \end{cases}$$

Theorem 7.25 \ Readback *vs* crumble contexts

Let c be a crumble and C, C' be crumble contexts such that $C_{\downarrow}$ and $C'_{\downarrow}$ are weak contexts.

1. *Plugging:* If $C\langle c\rangle$ is well-named, and $C \perp c$, then $C\langle c\rangle_{\downarrow} = C_{\downarrow}\langle c_{\downarrow}\rangle$.
2. *Composition:* If $C\langle C'\rangle$ is well-named and $C \perp C'$, then $C\langle C'\rangle_{\downarrow} = C_{\downarrow}\langle C'_{\downarrow}\rangle$ and it is a weak context.

Proof.

1. We proceed by induction on the structure of C. If $C \coloneqq \langle\cdot\rangle$, then $C_\downarrow = \langle\cdot\rangle$ and so $C\langle c\rangle_\downarrow = c_\downarrow = C_\downarrow\langle c_\downarrow\rangle$. Otherwise $C \coloneqq (b, e[x{\leftarrow}\langle\cdot\rangle])$ with $c \coloneqq (b', e')$; since $C\langle c\rangle = (b, e[x{\leftarrow}b']e')$ is well-named by hypothesis, it follows that $x \notin \mathsf{dom}(e')$; together with $C \perp c$ it implies that $(b, e) \perp e'$; finally $C\langle c\rangle_\downarrow = (b, e)_\downarrow\{x{\leftarrow}(b', e')_\downarrow\} = C_\downarrow\langle c_\downarrow\rangle$ by proposition 7.16/2 and because by hypothesis C unfolds to a context.

2. First of all, note that the composition of two weak contexts is a context, thus $C_\downarrow\langle C'_\downarrow\rangle$ is a context since $C_\downarrow$ and $C'_\downarrow$ are so by hypothesis. It remains to prove that $C\langle C'\rangle_\downarrow = C_\downarrow\langle C'_\downarrow\rangle$. We proceed by cases:

   - If $C \coloneqq \langle\cdot\rangle$, then $C_\downarrow = \langle\cdot\rangle$ and $C\langle C'\rangle = C'$, thus $C\langle C'\rangle_\downarrow = C'_\downarrow = C_\downarrow\langle C'_\downarrow\rangle$.
   - If $C' \coloneqq \langle\cdot\rangle$, then $C'_\downarrow = \langle\cdot\rangle$ and $C\langle C'\rangle = C$, so $C\langle C'\rangle_\downarrow = C_\downarrow = C_\downarrow\langle C'_\downarrow\rangle$.
   - Finally, if $C \coloneqq (b, e[x{\leftarrow}\langle\cdot\rangle])$ and $C' = (b', e'[y{\leftarrow}\langle\cdot\rangle])$, then $C\langle C'\rangle = (b, e[x{\leftarrow}b']e'[y{\leftarrow}\langle\cdot\rangle])$ and so

$$\begin{aligned} C\langle C'\rangle_\downarrow &= (b, e[x{\leftarrow}b']e')_\downarrow\{y{\leftarrow}\langle\cdot\rangle\} \\ &= (b, e)_\downarrow\{x{\leftarrow}(b', e')_\downarrow\}\{y{\leftarrow}\langle\cdot\rangle\} && \text{(a)} \\ &= (b, e)_\downarrow\{x{\leftarrow}(b', e')_\downarrow\{y{\leftarrow}\langle\cdot\rangle\}\} && \text{(b)} \\ &= C_\downarrow\langle C'_\downarrow\rangle \end{aligned}$$

     where:

     (a) by proposition 7.16/2, which is applicable because $(b, e) \perp e'$, by $C \perp C'$, $x \notin \mathsf{dom}(e')$, and $C\langle C'\rangle$ is well-named;

     (b) because $y \notin \mathsf{fv}((b, e)_\downarrow)$, which follows from $\mathsf{fv}((b, e)_\downarrow) \subseteq \mathsf{fv}(b, e)$ (proposition 7.13/2), from $y \neq x$ by the well-namedness of $C\langle C'\rangle$, and from the hypothesis $C \perp C'$. ∎

Properties of translation. Proposition 7.26 below provides the properties of the translation that are used to prove the invariants of the Crumble GLAM (propositions 8.8 and 9.8).

Proposition 7.26 \ Properties of the crumbling translation

For every λ-term t:

1. *Freshness:* $\underline{t}$ is well-named.
2. *Closure:* if t is closed, then $\mathsf{fv}(\underline{t}) = \emptyset$.
3. *Disjointedness:* $\underline{t} \perp \underline{t}$.

4. *Abstractions:* every abstraction occurring in $\underline{t}$ is the translation of a λ-term.
5. *Contextual decoding:* if $\underline{t} = C\langle c\rangle$, then $C_\downarrow$ is a right v-context.

Proof.

1. It follows immediately from the freshness condition in the definition of translation.
2. By proposition 7.13.
3. It follows immediately from the freshness condition in the definition of translation.
4. By induction on t and by cases on the rules defining the translation.
5. We prove the required statement mutually with the corresponding statement for the auxiliary translation, *i.e.* that $C_\downarrow$ is a right v-context whenever $\overline{t} = C\langle c\rangle$. We proceed by induction on the structure of t. Cases:
   - *Abstraction, i.e.* $t = \lambda x.s$. Then $\underline{t} = (\lambda x.\underline{s}, \epsilon)$ and so the only possible crumble context C such that $\underline{t} = C\langle c\rangle$ for some crumble c is $C = \langle\cdot\rangle$ and so $C_\downarrow = \langle\cdot\rangle$, which is a right v-context. The proof for $\overline{t}$ is similar.
   - *Variable, i.e.* $t = x$. Similar to the case above.
   - *Application, i.e.* $t = su$. Then $\underline{t} = (vv', ee')$ where $(v, e) := \overline{s}$ and $(v', e') := \overline{u}$. We discuss separately three subcases:
     - *Empty, i.e.* $C = \langle\cdot\rangle$: trivial.
     - $C = (xv', [x{\leftarrow}\langle\cdot\rangle])\langle C'\rangle$ where C' is a crumble context of $\underline{s}$, *i.e.* $\underline{s} = C'\langle c\rangle$ for some C'. The readback of the crumble context $C'' := (xv', [x{\leftarrow}\langle\cdot\rangle])$ is $C''_\downarrow = xv'_\downarrow\{x{\leftarrow}\langle\cdot\rangle\} = \langle\cdot\rangle v'_\downarrow$, which is a right v-context because $v'_\downarrow$ is a λ-value by proposition 7.13/4. By *i.h.*, $C'_\downarrow$ is a right v-context. By the freshness conditions in the definition of translation, $C'' \perp C'$; according to theorem 7.25/2, $C_\downarrow = C''\langle C'\rangle_\downarrow = C''_\downarrow\langle C'_\downarrow\rangle$, which is a right v-context since the composition of right v-context is a right v-context (proposition 3.2).
     - $C = (vy, e[y{\leftarrow}\langle\cdot\rangle])\langle C'\rangle$ where C' is a crumble context of $\underline{u}$, *i.e.* $\underline{u} = C'\langle c\rangle$ for some C'. The readback of the crumble context $C'' := (vy, e[y{\leftarrow}\langle\cdot\rangle])$ is $C''_\downarrow = (v, e[y{\leftarrow}\langle\cdot\rangle])_\downarrow(y, e[y{\leftarrow}\langle\cdot\rangle])_\downarrow$ by proposition 7.11. Moreover, $(v, e[y{\leftarrow}\langle\cdot\rangle])_\downarrow = (v, e)_\downarrow\{y{\leftarrow}\langle\cdot\rangle\} = s\{y{\leftarrow}\langle\cdot\rangle\} = s$ by proposition 7.17 and because $y \notin \mathsf{fv}(s)$ since y is a fresh variable introduced by crumbling; $(y, e[y{\leftarrow}\langle\cdot\rangle])_\downarrow = (y, e)_\downarrow\{y{\leftarrow}\langle\cdot\rangle\} = y\{y{\leftarrow}\langle\cdot\rangle\} = \langle\cdot\rangle$ because $y \notin \mathsf{dom}(e)$ since y and e are generated by independent recursive calls of the translation function. Thus we obtain that $C''_\downarrow = s\langle\cdot\rangle$, clearly a right v-context. By *i.h.*, $C'_\downarrow$ is a right v-context. By the freshness

conditions in the definition of translation, $C'' \perp C'$; according to theorem 7.25/2, $C_\downarrow = C''\langle C'\rangle_\downarrow = C''_\downarrow\langle C'_\downarrow\rangle$, which is a right v-context since the composition of right v-context is a right v-context (proposition 3.2). The statement for the auxiliary translation $\overline{t}$ can be proved in a similar way.

Size of crumbled forms. We naturally extend to crumbled forms the size defined for λ-terms in definition 2.6:

Definition 7.27 \ Size of crumbled forms $|c|$

We define the size $|\bullet|$ of crumbled forms, crumbles and environments by structural induction:

$$|(b,e)| \coloneqq |b| + |e| \qquad |\epsilon| \coloneqq 0 \qquad |e[x{\leftarrow}b]| \coloneqq 1 + |e| + |b|$$
$$|x| \coloneqq 0 \qquad |\lambda x.c| \coloneqq 1 + |c| \qquad |vw| \coloneqq 1 + |v| + |w|$$

The following proposition shows that the translation $\underline{t}$ of a λ-term t has size proportional to the size of t: this fact will be useful in the next chapters, when we will need to bound the size of the crumble being copied by machine transitions in order to study the complexity of crumbled evaluation.

Proposition 7.28 \ Bound on the size of translated

$|\underline{t}| \leq 2|t|$ for every λ-term t.

Proof. We prove the statement mutually with the corresponding statement for the auxiliary translation, and by induction on the structure of t:

- *Variable, i.e.* $t \coloneqq x$. Then $\underline{t} = \overline{t} = (x, \epsilon)$, hence $|\underline{x}| = 0 \leq 2|x|$.
- *Abstraction, i.e.* $t \coloneqq \lambda x.s$. Then $\underline{t} = \overline{t} = (\lambda x.\underline{s}, \epsilon)$; by *i.h.*, $|\underline{s}| \leq 2|s|$ and hence $|\underline{t}| = 1 + |\underline{s}| \leq 1 + 2|s| \leq 2(1 + |s|) = 2|t|$.
- *Application, i.e.* $t \coloneqq su$. Then $\underline{t} = (vv', ee')$ and $\overline{t} = (z, [z{\leftarrow}vv']ee')$ with $(v, e) \coloneqq \overline{s}$ and $(v', e') \coloneqq \overline{u}$. By *i.h.* $|\overline{s}| = |v| + |e| \leq 2|s|$ and $|\overline{u}| = |v'| + |e'| \leq 2|u|$. Therefore $|\underline{t}| = 1+|v|+|v'|+|e|+|e'| \leq 1+2|s|+2|u| \leq 2(1+|s|+|u|) = 2|t|$. Similarly, $|\overline{t}| = 2+|v|+|v'|+|e|+|e'| \leq 2+2|s|+2|u| = 2(1+|s|+|u|) = 2|t|$.

7.4 Related Works

In this section we discuss some already existing transformations of λ-terms that are related to crumbling.

Both *Administrative Normal Forms* (ANF) and *Continuation-Passing Style* (CPS) are well-known transformations, often applied to compilation in order to obtain desirable theoretical properties. The ANF is a variant of the CPS; roughly, the difference is that the ANF transformation does not change the type of a term, when terms are typed (in this dissertation, however, we work without types).

Historically, the discovery of the ANF had an immediate practical impact, to the point that some compilers for functional languages dropped the CPS for the ANF transformation soon afterwards—see the retrospective by Flanagan et al., 1993. However, the literature on ANF is scarce. Beyond the already cited original papers, Danvy has also studied them and their relationship to CPS translation, but usually calling them *monadic normal forms* (Danvy, 1994; Hatcliff and Danvy, 1994; Danvy, 2003b) because of their relationship with Moggi's monadic λ-calculus (Moggi, 1991). That terminology however sometimes describes a more liberal notion of terms, for instance the one by Kennedy, 2007. Kennedy's paper is also another relevant piece in the literature on ANF, and we discuss it below.

Kennedy's slowdown. Kennedy, 2007 compares three different calculi: a monadic calculus (with ES), a calculus of ANFs, and the image of a CPS transformation. ANFs are just canonical shapes of monadic terms where the topmost term and the body of each abstraction is a crumble, *i.e.* a term together with a list of explicit substitutions that map variables to terms (instead of crumbles). Kennedy rightly observes that ANFs are not preserved by standard β/v-reduction and thus, after every β/v-step, some "search" steps are required to reach the ANF shape. In fact, in the monadic calculus β/v-redexes can be hidden by explicit substitutions which need to be commuted to reveal the β/v-redex. Kennedy provides an example (see example 7.29) where the number of commutations is not bounded linearly by the number of β/v-steps and blames the inefficiency of his compiler on that. In his example, the number of commutations is *quadratic* in the number of β/v-steps, since the i^{th} β/v-step is immediately followed by i commutation steps.

Example 7.29 \ Kennedy's quadratic example

Here we show the example of evaluation in the monadic calculus by Kennedy, 2007 where the number of commutation steps is quadratic in the number of β/v-steps. Note: $\to^i$ stands for the composition of i $\to$-steps. The i^{th} β/v-step is immediately followed by i commutation steps $\to_{\mathsf{let}}$ that just append two lists of substitutions moving one substitution at a time.

$$
\begin{aligned}
t &\coloneqq (z_1x_0)[z_1{\leftarrow}\lambda x_1.by_1[y_1{\leftarrow}z_2x_1]][z_2{\leftarrow}\lambda x_2.by_2[y_2{\leftarrow}z_3x_2]]\dots[z_n{\leftarrow}\lambda x_n.by_n[y_n{\leftarrow}bx_n]]\\
&\to_{\beta/v}{}^{i}\cup\to_{\mathsf{let}}{}^{i(i+1)/2}\\
&\qquad (z_1x_0)[z_1{\leftarrow}\lambda x_1.by_1[y_1{\leftarrow}z_2x_1]][z_2{\leftarrow}\lambda x_2.by_2[y_2{\leftarrow}z_3x_2]]\dots\\
&\qquad \dots[z_{n-i-1}{\leftarrow}\lambda x_{n-i-1}.by_{n-i-1}[y_{n-i-1}{\leftarrow}z_{n-i}x_{n-i-1}]]\\
&\qquad \dots[z_{n-i}{\leftarrow}\lambda x_{n-i}.by_{n-i}[y_{n-i}{\leftarrow}by_{n-i+1}][y_{n-i+1}{\leftarrow}by_{n-i+2}]\dots[y_n{\leftarrow}bx_{n-i}]]\\
&\to_{\beta/v}\\
&\qquad (z_1x_0)[z_1{\leftarrow}\lambda x_1.by_1[y_1{\leftarrow}z_2x_1]][z_2{\leftarrow}\lambda x_2.by_2[y_2{\leftarrow}z_3x_2]]\dots\\
&\qquad \dots[z_{n-i-1}{\leftarrow}\lambda x_{n-i-1}.by_{n-i-1}\\
&\qquad \dots\dots[y_{n-i-1}{\leftarrow}by_{n-i}[y_{n-i}{\leftarrow}by_{n-i+1}][y_{n-i+1}{\leftarrow}by_{n-i+2}]\dots[y_n{\leftarrow}bx_{n-i-1}]]\\
&\qquad \dots]\\
&\to_{\mathsf{let}}{}^{i+1}\\
&\qquad (z_1x_0)[z_1{\leftarrow}\lambda x_1.by_1[y_1{\leftarrow}z_2x_1]][z_2{\leftarrow}\lambda x_2.by_2[y_2{\leftarrow}z_3x_2]]\dots\\
&\qquad \dots[z_{n-i-1}{\leftarrow}\lambda x_{n-i-1}.by_{n-i-1}[y_{n-i-1}{\leftarrow}by_{n-i}][y_{n-i}{\leftarrow}by_{n-i+1}]\dots\\
&\qquad \dots\dots[y_n{\leftarrow}bx_{n-i-1}]\\
&\qquad \dots]
\end{aligned}
$$

In the Crumble and Open Crumble GLAM instead, the commutation steps are integrated in the beta rule simply by concatenating the two lists in constant time.

The quadratic example by Kennedy stands in the ANF fragment. Therefore Kennedy too hastily concludes that the quadratic blowup also affects the ANF calculus.

However, Kennedy misses the fact that the explicit substitutions in ANFs form a list and that the commutations steps altogether just implement the append function of two lists. Since append can be implemented in constant time, the complexity of evaluation in the ANF calculus is just linear (and not quadratic) in the number of β/v-steps; this is the same complexity that we achieve for the Crumble and Open Crumble GLAM in the upcoming chapters.

Conditionals. Another problem with ANF pointed out by Kennedy, 2007 is the fact that the ANF does not smoothly scale up when the λ-calculus is extended to further constructs such as conditionals or pattern matching. Essentially, the ANF requires conditionals and pattern matching to be out of ES, that is, to never have an expression such as $u[x \leftarrow \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ s]$. Unfortunately, these configurations can be created during evaluation. To preserve the ANF, one is then led to add so-called *commuting conversions* such as:

$$u[x \leftarrow \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ s] \to \mathsf{if}\ v\ \mathsf{then}\ (u[x{\leftarrow}t])\ \mathsf{else}\ (u[x{\leftarrow}s]). \qquad \text{(CC-If)}$$

Clearly, there is an efficiency issue: the commutation causes an undesirable duplication of the subterm u. A way out is to use a continuation-like technique, which makes Kennedy conclude that then there is no point in preferring ANF to CPS.

This is where our crumble representation departs from the ANF, as we do not require conditionals and pattern matching to be out of ES (see (Accattoli et al., 2019b)). First of all, let us point out that Kennedy only studies the closed case, but we are interested in open and strong evaluation with the final goal of improving the implementation of proof assistants. In that setting, commutations of conditionals and pattern matching such as those hinted at by Kennedy are not valid, as they are not validated by dependent type systems like those of Coq or Agda. For example, adding the CC-If rule above when the conditional is dependently typed breaks the property of *subject reduction*, as typed terms reduce to ill-typed terms. Consider the term:

$$(x+1)[x\colon (\mathsf{if}\ \mathit{true}\ \mathsf{then}\ \mathit{nat}\ \mathsf{else}\ \mathit{bool}) \leftarrow \mathsf{if}\ \mathit{true}\ \mathsf{then}\ 0\ \mathsf{else}\ \mathit{false}]\colon \mathit{nat}$$

that has type nat because the type of x is convertible to nat. By applying rule CC-If, we obtain:

$$\mathsf{if}\ \mathit{true}\ \mathsf{then}\ ((x+1)[x\leftarrow 0])\ \mathsf{else}\ (\underline{(x+1)[x\leftarrow \mathit{false}]})$$

which is clearly ill-typed. But commuting conversions are malicious even without types: although valid on closed untyped terms, on open terms they may alter the behaviour of programs, as they can create new redexes. For example, consider the following term in which rule CC-If introduces a divergent subterm:

$$(x\delta)[x\leftarrow(\mathsf{if}\ y\ \mathsf{then}\ I\ \mathsf{else}\ \delta)] \to \mathsf{if}\ y\ \mathsf{then}\ ((x\delta)[x\leftarrow I])\ \mathsf{else}\ (\underline{(x\delta)[x\leftarrow\delta]})$$

where $I := \lambda z.z$ is the identity, $\delta := \lambda x.xx$ is the duplicator, and the underlined subterm reduces to the looping combinator $\delta\delta$. The problem in the open case is much more general, as not even the CPS transformation would work: its properties do not naturally scale up to open terms. We provide a counterexample to the simulation property in the open case in the paragraph below.

To sum up, commuting conversions are not valid in our framework, nor is it possible to do a CPS transformation. Therefore, we accept that conditionals and pattern matching may appear in ES (in contrast to Kennedy) and so depart from the ANF. Since these constructs do not play any special role, we decided to omit them from this dissertation and stick to the ordinary λ-calculus. To handle these constructs, we simply treat them exactly as applications: we add sharing points in between any two iterated constructs, but allow them to appear inside ES. Therefore, our results immediately scale up to the case with conditionals and pattern matching, as we show thoroughly in Accattoli et al., 2019b.

Inadequate CPS. Danvy and Filinski, 1992 show that their CPS transformation scales up to open λ-terms (their Theorem 2). On open λ-terms, however, they consider Plotkin's CbV operational semantics λ_{Plot}, which is adequate only for closed terms, as discussed in section 3.2. When one considers one of the equivalent adequate CbV semantics studied by Accattoli and Guerrieri, 2016; Accattoli and Guerrieri, 2018 for the open case, for instance the fireball calculus λ_{fire}, then the properties of the CPS no longer hold, in particular it does not commute with evaluation, as the following example shows.

Take the following open λ-term $t := (\lambda x.\lambda y.y)(zz)v$, where v is a value, say a distinguished variable. In λ_{Plot} the λ-term t is β/v-normal, but in λ_{fire} we have:

$$t := (\lambda x.\lambda y.y)(zz)v \to_{\beta/f} (\lambda y.y)v \to_{\beta/f} v$$

Now, consider the CPS translation $\mathtt{cps}(t)$ of t, according to the definition in Danvy and Filinski, 1992. We use λ for standard ("dynamic", in Danvy's terminology) abstraction, and Λ and $@$ for "static" abstraction and application, respectively. If a generalized version of Theorem 2 in Danvy and Filinski, 1992 held in the open case, one would expect that $@(\mathtt{cps}(t))I$ (where $I := \lambda z.z$) evaluates to v, as v is a variable. But, even using an unrestricted β-reduction that goes under abstraction as evaluation, we obtain (by reducing all static redexes first, followed by all dynamic redexes):

$$\begin{aligned}
&@(\mathtt{cps}(t))I\\
&= (\Lambda k.@(\Lambda x.@(\Lambda y.@y(\lambda w.\lambda a.@(\Lambda b.@b(\lambda c.\lambda d.@(\Lambda e.@ec)(\Lambda e.de)))(\Lambda b.ab)))\\
&\quad (\Lambda y.@(\Lambda w.@(\Lambda a.@az)(\Lambda a.@(\Lambda b.@bz)(\Lambda b.(ab)(\lambda c.@wc))))(\Lambda w.(yw)(\lambda a.@xa))))\\
&\quad (\Lambda x.@(\Lambda y.@yv)(\Lambda y.(xy)(\lambda w.@Kw))))I\\
&\to_\beta{}^* (zz)(\lambda x.((\lambda y.\lambda w.w(\lambda a.\lambda b.ba))x)(\lambda y.yv(\lambda w.Iw)))\\
&\to_\beta{}^* (zz)(\lambda x.v)
\end{aligned}$$

where $(zz)(\lambda x.v)$ is not even β-equivalent to v. The CPS translation—like Plotkin's calculus—gets stuck trying to evaluate zz, whereas the term reduces to v in λ_{fire}.

Summing up, Danvy and Filinski's CPS transformation does not fully scale up to open λ-terms: to prove scalability, one should use an adequate CbV evaluation for open λ-terms (such as the one of the fireball calculus), instead of Plotkin's one. It is worth noting that this problem affects also other CPS translations, such as the ones defined by Plotkin, 1975 or by Lassen, 2005. Likely, this is the reason why Lassen, 2005 states his Theorem 4.6 (the analogous of Theorem 2 in Danvy and Filinski, 1992) only for closed λ-terms.

Static Single Assignment. *Static Single Assignment* (SSA) is yet another transformation, used in compilers to represent flow properties of programs (Cytron et al., 1991). As the name suggests, in SSA each variable can only assigned once; this means that if different branches of a program need to assign to the same variable x, a new variable is introduced for every occurrence of x, and then a new form of assigned called ϕ-function is added at join points, so to re-identify the different instances of x.

SSA can be viewed as ANF plus ϕ-function to implement joint points: in this way, commuting conversions are not necessary, and neither the consequent code duplication mentioned above. The similarity between crumbled forms and SSA forms is that in both settings variables can be assigned only once, even thought the crumbled translation does not apply to imperative programs. Moreover, crumbled environment do not need ϕ-functions because no variable can ever be assigned twice in different branches, due to the well-named requirement (no duplicates in the domain of crumbled environments) that is enforced during evaluation.

Chapter 8

Closed Crumbling Evaluation

In this chapter we will describe how to evaluate crumbled forms with a micro-step operational semantics in order to implement Closed CbV.

An important aspect of crumbling evaluation is that it actually blurs the distinction between a linear calculus and an abstract machine that we have outlined on page 40: in fact, it allows to use the sequential structure of the environment as the only data structure needed to search for redexes. For this reason, the operational semantics for crumbled forms that we present in the following sections is in the style of a linear calculus, because spelling out the straightforward search for redexes is not really informative. Nonetheless, we do call it an *abstract machine*, both because of the blurred distinction in the crumbled case and also because we manage names explicitly. More precisely, we call it a *crumble abstract machine*, namely the *Crumble GLAM*, that expresses its halfway status. In section 10.1 we also spell out the actual abstract machine, by making search transitions explicit.

8.1 The Crumble GLAM

Evaluation. In order to define the Crumble GLAM, we need some new notations and terminology. We introduce a restricted version of environments and crumbles, which we call respectively λ-*environments* and λ-*crumbles*, because we require them to be only made up of abstractions. They are formally defined and noted as follows:

λ-crumbled forms

λ-ENVIRONMENTS $e_\lambda ::= \epsilon \mid e_\lambda[x \leftarrow \lambda y.c]$ $\qquad$ λ-CRUMBLES $c_\lambda ::= (\lambda y.c, e_\lambda)$

Essentially, a λ-environment stands for the already evaluated coda of the environment described in the paragraph about micro-steps in section 7.2, and λ-crumbles are fully evaluated crumbles, that is, the final states of the machine, as we show below.

The rewrite rules act on crumbles whose environments are λ-environments. The root steps are:

Rule at top-level

$$((\lambda x.c)\, v, e_\lambda) \mapsto_\beta (c \mathbin{@} [x{\leftarrow}v])^\alpha \mathbin{@} e_\lambda$$

$$(x, e_\lambda) \mapsto_{\mathsf{sub}/v} (e_\lambda(x), e_\lambda) \quad \text{if } x \in \mathsf{dom}(e_\lambda)$$

$$(xv, e_\lambda) \mapsto_{\mathsf{sub}/\mathit{left}} (e_\lambda(x)\, v, e_\lambda) \quad \text{if } x \in \mathsf{dom}(e_\lambda)$$

Contextual closure

$$C\langle c\rangle \to_r C\langle d\rangle \quad \text{if } c \mapsto_r d \text{ for } r \in \{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$$

Evaluation

$$\to \;:=\; \to_\beta \;\cup\; \to_{\mathsf{sub}}$$
$$\to_{\mathsf{sub}} \;:=\; \to_{\mathsf{sub}/v} \;\cup\; \to_{\mathsf{sub}/\mathit{left}}$$

Evalution of the Crumble GLAM / **Figure 8.1**

As usual, given a crumble c we use c^α for a crumble obtained by α-renaming the names in the domain of c with fresh ones so that c^α is well-named. Actually, in rule $\to_\beta$ the α-renaming of the root-step has to pick names that are fresh also with respect to the crumble context enclosing it. This point may seem odd but it is necessary to avoid name clashes, and it is trivially obtained in our concrete implementation, where variable names are memory locations and picking a fresh name amounts to allocating a new location, that is of course new globally. The evaluation for the Crumble GLAM is defined by:

$$\to \;:=\; \to_\beta \;\cup\; \to_{\mathsf{sub}/v} \;\cup\; \to_{\mathsf{sub}/\mathit{left}} .$$

Let us explain the rules:

- *Rule* $\to_\beta$ (let us forget about the α-renaming for the moment—see the next paragraph): the rule basically does what is explained in the paragraph about abstractions in section 7.2. Namely, the environment of the body c of the abstraction and the external environment e_λ are concatenated (via the appending operation $@$) interposing the entry $[x{\leftarrow}v]$ created by the redex itself.

- *Rule* $\to_{\mathsf{sub}/v}$: the variable x is substituted by the corresponding crumbled value in the environment e_λ, if any. In the closed case, a forthcoming invariant actually guarantees that $e_\lambda(x)$ is always defined.

- *Rule* $\to_{\mathsf{sub}/\mathit{left}}$: similar to the previous case, it substitutes on the left part of an application.

Note that, according to the definitions of plugging and root-steps, the rules $\to_\beta$, $\to_{\mathsf{sub}/v}$ and $\to_{\mathsf{sub}/\mathit{left}}$ impose a *right-to-left* evaluation, since the environment on the right of a redex is a λ-environment, *i.e.* it is only made of abstractions, which means that it has already

been evaluated; this will be made evident in the harmony property for Crumble GLAM, proposition 8.5 below. Adopting right-to-left evaluation implies that the Crumble GLAM does not need a rule $\to_{\mathsf{sub}/\mathit{right}}$ symmetrical to $\to_{\mathsf{sub}/\mathit{left}}$, whose root step would be $(vx, e_\lambda) \mapsto_{\mathsf{sub}/\mathit{right}} (v\, e_\lambda(x), e_\lambda)$ with $x \in \mathsf{dom}(e_\lambda)$: indeed, if v is a variable then $\to_{\mathsf{sub}/\mathit{left}}$ applies to the same redex (vx, e_λ), otherwise v is an abstraction and then $\to_\beta$ applies to (vx, e_λ).

Unchaining abstractions. The substitution performed by the rule $\to_{\mathsf{sub}/v}$ may seem an unneeded optimization; quite the opposite, it fixes an issue causing quadratic overhead in the machine. The culprits are the malicious chains of renamings discussed on page 72, *i.e.* environments of the form $[x_1 \leftarrow x_2][x_2 \leftarrow x_3] \cdots [x_n \leftarrow \lambda y.c]$ substituting variables for variables and finally leading to an abstraction. Accattoli and Coen, 2015 showed that the key to linear overhead is to perform substitution steps while going through the chain from right to left, which is exactly what the $\to_{\mathsf{sub}/v}$ rule does.

The cost and the place of α-renaming. Abstract machines with global environments have to α-rename at some point, as discussed on section 4.3. In our implementation renaming is implemented as a copy function, whose computational complexity is under control because of forthcoming invariants of the machine; this is all standard (Accattoli and Barras, 2017). Often the burden of renaming/copying is put on the substitution rules. It is less standard to put it on the beta rule, as we do here. In the closed case, the complexity does not change, as our analysis below shows. It turns out, however, that in the open case having renaming on the beta transition enables an asymptotically faster machine—see section 8.3.

Definition 8.1 \ Reachable crumble

We call a crumble *reachable* (in the Crumble GLAM) if it is obtained by a sequence of evaluation steps starting from the translation $\underline{t}$ of a closed λ-term t.

Let us have a look at an example of evaluation in the Crumble GLAM:

Example 8.2 \ Evaluation sequence

Consider the crumble $\underline{\delta\delta} = (\delta_b\, \delta_b, \epsilon)$, where $\delta_b = \lambda x.(xx, \epsilon)$. Then:

$$\begin{aligned}\underline{\delta\delta} &\to_\beta (xx, [x \leftarrow \delta_b]) \to_{\mathsf{sub}/\mathit{left}} (\delta_b\, x, [x \leftarrow \delta_b]) \to_\beta \\ &\to_\beta (yy, [y \leftarrow x][x \leftarrow \delta_b]) \to_{\mathsf{sub}/\mathit{left}} (yy, [y \leftarrow \delta_b][x \leftarrow \delta_b]) \to_{\mathsf{sub}/\mathit{left}} \cdots\end{aligned}$$

In example 7.9 we introduced the crumble $\underline{\delta\delta I} = (z\, I_b, [z \leftarrow \delta_b\, \delta_b])$ where $I_b = \lambda x.(x, \epsilon)$; in accordance with the crumble decomposition shown in example 7.19, we have:

$$\begin{aligned}\underline{\delta\delta I} &\to_\beta (z\, I_b, [z \leftarrow xx][x \leftarrow \delta_b]) \to_{\mathsf{sub}/\mathit{left}} (z\, I_b, [z \leftarrow \delta_b\, x][x \leftarrow \delta_b]) \to_\beta \\ &\to_\beta (z\, I_b, [z \leftarrow yy][y \leftarrow x][x \leftarrow \delta_b]) \to_{\mathsf{sub}/\mathit{left}} (z\, I_b, [z \leftarrow yy][y \leftarrow \delta_b][x \leftarrow \delta_b]) \to_{\mathsf{sub}/\mathit{left}} \cdots\end{aligned}$$

Consider now the *open* crumble $c := \underline{\delta\delta(xx)} = (zw, [z \leftarrow \delta_b\, \delta_b][w \leftarrow xx])$. The crumble c is normal because its only possible decomposition of the form $C\langle b, e_\lambda\rangle$ is for $e_\lambda = \epsilon$ (as

xx is not an abstraction), and no reduction rule applies to the rightmost entry $[w{\leftarrow}xx]$ since x is globally free.

A famous key property of Closed CbV is *harmony* (proposition 3.1): given a closed λ-term t, either it diverges or it evaluates to a (closed) λ-abstraction, *i.e.* t is β/v-normal if and only if t is a (closed) λ-abstraction. The Crumble GLAM satisfies an analogous property (proposition 8.5) by replacing abstractions with λ-crumbles.

Proposition 8.3

Let $c = (b, e_\lambda)$ be a well-named closed crumble, and b have the following property: b is an abstraction, or b is x or xv but x is not defined in e_λ. Then c is a λ-crumble.

Proof. Let $c = (b, e_\lambda)$ as above: it suffices to prove that b is an abstraction. This follows easily from the hypothesis that c is closed, since the cases where b is x or xv but x is not defined in e_λ are impossible.

Corollary 8.4

If the well-named closed crumble (b, e_λ) is normal, then it is an λ-crumble.

Proposition 8.5 \ Harmony for the Crumble GLAM

A closed crumble c is normal if and only if it is an λ-crumble.

Proof.

($\Rightarrow$) Let $c = (b, e)$ be well-named, closed, and normal. We proceed by structural induction on e:

- if $e = \epsilon$, then (b, ϵ) is an λ-crumble by corollary 8.4;
- if $e = [x{\leftarrow}b']e'$, then also the crumble (b', e') is normal. By *i.h.* (b', e') is an λ-crumble, and therefore $e = [x{\leftarrow}b']e'$ is an λ-environment. By corollary 8.4, $c = b, e$ is an λ-crumble.

($\Leftarrow$) Let $c = c_\lambda$, we need to prove that c_λ is normal. Let $c_\lambda = C\langle(v, e_\lambda)\rangle$ for some C, v, e_λ with v an abstraction. Clearly no reduction rule is applicable, because v is not a variable or an application.

Before turning to the next section, we prove a technical proposition that will come handy later.

Proposition 8.6 \ Readback to a value

For every crumbled value v and λ-environment e_λ, one has that $(v, e_\lambda)_\downarrow$ is a value. If moreover v is an abstraction, then $(v, e_\lambda)_\downarrow$ is a λ-abstraction.

Proof. By induction on the length of e_λ. There are two cases, depending if v is an abstraction or a variable.

- *Abstraction:* If $e_\lambda := \epsilon$, then clearly $(v, e_\lambda)_\downarrow = v_\downarrow$ is a λ-abstraction and hence a value. Otherwise $e_\lambda := e'_\lambda[x \leftarrow b]$ where b is an crumbled value (and hence $b_\downarrow$ is a value): thus, $(v, e_\lambda)_\downarrow = (v, e'_\lambda)_\downarrow\{x \leftarrow b_\downarrow\}$; by *i.h.*, $(v, e'_\lambda)_\downarrow$ is a λ-abstraction, thus $(v, e_\lambda)_\downarrow$ is a λ-abstraction (and so a value) by proposition 3.3.
- *Variable:* If $v := x \notin \mathsf{dom}(e_\lambda)$, then $(v, e_\lambda)_\downarrow = x$ which is a value; otherwise $v := x \in \mathsf{dom}(e_\lambda)$ with $e_\lambda := e'_\lambda[x \leftarrow \lambda y.c]e''_\lambda$, and then $(x, e_\lambda)_\downarrow = (\lambda y.c, e''_\lambda)_\downarrow$ (since $x \notin \mathsf{dom}(e'_\lambda)$) is a value by *i.h.*, according to the previous point. ∎

8.2 Implementation

To show that the Crumble GLAM correctly implements the right-to-left Closed CbV evaluation, we apply the abstract approach described in section 4.4, which we reuse as well in the following chapters for other crumble abstract machines and other calculi.

Recall that a "state" of a crumble machine is simply a crumble. We will show in theorem 8.16 that the Crumble GLAM, the right-to-left Closed CbV evaluation $\to_{\beta/v}$, the crumbling translation $\underline{\bullet}$, and the readback $\bullet_\downarrow$ form an implementation system: as a consequence, the Crumble GLAM implements right-to-left Closed CbV strategy in the λ-calculus.

To prove the implementation theorem (theorem 8.16) we are going to provide five of the six sufficient properties that it requires; the sixth one, the *termination of overhead transitions*, is subsumed by the much finer complexity analysis in section 8.3.

Execution invariants. The sufficient conditions, as usual, are proved by means of a few invariants of the machine, given by proposition 8.8 below. These invariants are essentially the properties of the translation in proposition 7.26 extended to all reachable crumbles. One of them—namely *contextual decoding*—however, is weaker because reachable crumbles do not necessarily have the same nice structure as the initial crumbles obtained by translation of a λ-term, as the next remark explains.

Remark 8.7

Not all crumble contexts unfold to contexts (example 7.21), but crumble contexts obtained by decomposing crumbles translating λ-terms do (proposition 7.26/5): this is the contextual decoding property. Unfortunately, it is not preserved by evaluation. Consider the

crumble $c := \underline{(\lambda x.x(xx))\, I} = ((\lambda x.(xy, [y{\leftarrow}xx]))I_b, \epsilon)$ with $I_b = \lambda z.(z, \epsilon)$. Clearly, $c = \langle\cdot\rangle\langle\underline{(\lambda x.x(xx))\, I}\rangle$ where $\langle\cdot\rangle_\downarrow = \langle\cdot\rangle$ is a context. After one β step, the crumble c reaches $\underline{(xy, [y{\leftarrow}xx][x{\leftarrow}I_b])} = C\langle(I_b, \epsilon)\rangle$ for $C := (xy, [y{\leftarrow}xx][x{\leftarrow}\langle\cdot\rangle])$. But C unfolds to $C_\downarrow = \langle\cdot\rangle(\langle\cdot\rangle\langle\cdot\rangle)$, which is not a λ-context.

Let us clarify the role of each invariant, that we prove below in proposition 8.8:

- *Freshness* and *Closure* are invariants about variable names, needed to ensure the basic functioning of the machine.
- *Subterm* is the key invariant for complexity analyses, as it allows to bound the size of duplicated subterms (that are always abstractions) using the size of the initial term. Usually, it is only needed for complexity analyses, while here it is needed for the implementation theorem as well. Knowing that an abstraction comes from the initial term, indeed, implies that it satisfies the strong contextual decoding invariant of proposition 7.26, fact that is used in the proof of its weak variant.
- The *Weak contextual decoding* invariant, finally, is essential to show that beta steps project on β-steps of λ_{Plot}.

Proposition 8.8 \ Invariants for the Crumble GLAM

For every reachable crumble c in the Crumble GLAM:

1. *Freshness:* c is well-named.
2. *Closure:* $\mathsf{fv}(c) = \emptyset$.
3. *Subterm:* every abstraction in c is a subterm (up to renaming) of the initial crumble.
4. *Contextual decoding (weak):* for every decomposition $C\langle b, e_\lambda\rangle$ where b is not a crumbled value, if C'' is a prefix of C then $C''_\downarrow$ is a right v-context.

Proof. By induction on the length of the reduction sequence leading to the crumble. The base cases hold by proposition 7.26 (by noting that for point 4, proposition 7.26/5 implies the weaker statement proposition 8.8/4). As for the inductive cases, we inspect each reduction rule:

1. The substitution rules sub/v and $\mathsf{sub}/\mathit{left}$ do not change the domain of the crumble, hence the claim follows from the *i.h.*. For the rule β the claim follows from the side condition.
2. The rules sub/v and $\mathsf{sub}/\mathit{left}$ do not change the domain of the crumble and only copy to the left a value from the environment, and the claim follows from the *i.h.*.

   The rule β copies to the toplevel and renames the body of an abstraction. By the properties of α-renaming $\mathsf{fv}((c \mathbin{@} [x{\leftarrow}v])^\alpha) = \mathsf{fv}(c \mathbin{@} [x{\leftarrow}v]) = \mathsf{fv}(\lambda x.c)$, and

since by *i.h.* $\mathsf{fv}(\lambda x.c) \subseteq \mathsf{dom}(e_\lambda)$, we can conclude with $\mathsf{fv}((c \ @ \ [x \leftarrow v])^\alpha) \subseteq \mathsf{dom}(e_\lambda)$.

3. The rules sub/v and $\mathsf{sub}/\mathit{left}$ introduce an abstraction, but it was already in the environment, and the claim follows from the *i.h.*. The rule β copies and renames the body of an abstraction that was already in the environment, and the claim follows from the *i.h.* since the translation commutes with the renaming of free variables (proposition 7.13/3).

4. Let $\underline{b''} \to^n C'\langle(b', e'_\lambda)\rangle \to_a C\langle(b, e_\lambda)\rangle$ (where b is not an abstraction). Cases of the reduction step $C'\langle(b', e'_\lambda)\rangle \to_a C\langle(b, e_\lambda)\rangle$:

   - Case β: $C'\langle((\lambda x.c)v, e_\lambda)\rangle \to_\beta C'\langle c^\alpha \ @ \ ([x^\alpha \leftarrow v]e_\lambda)\rangle$.
     Let C'' be a prefix of C. There are two sub-cases:
     * C'' *is a prefix of* C': by *i.h.* ${C''}_\downarrow$ is a right v-context.
     * C' *is a prefix of* C'', *i.e.* $C'' = C'\langle C'''\rangle$ and $c^\alpha = C'''\langle c'\rangle$. By proposition 7.26/4 and proposition 8.8/3 c is the translation of a λ-term, by proposition 7.13/3 c^α is so, and thus by proposition 7.26/5 ${C'''}_\downarrow$ is a right v-context. By *i.h.*, ${C'}_\downarrow$ is a right v-context as well. Since ${C''}_\downarrow = {C'}_\downarrow\langle{C'''}_\downarrow\rangle$ according to theorem 7.25/2, we obtain that ${C''}_\downarrow$ is a right v-context as composition of right v-contexts (proposition 3.2).
   - Case sub/v: it follows from the *i.h.* since C is necessarily a prefix of C'.
   - Case $\mathsf{sub}/\mathit{left}$: it follows from the *i.h.*, since $e'_\lambda = e_\lambda$ and $C = C'$.

Implementation theorem. To prove the implementation theorem (theorem 8.16) we first prove separately its requirements: determinism, transparency, projection, and progress.

Proposition 8.9 \ Determinism

$\to$ is deterministic.

Proof. Assume that there exists a crumble that may be decomposed in two ways $C\langle(b, e_\lambda)\rangle = C'\langle b', e'_\lambda\rangle$ such that they reduce respectively $C\langle(b, e_\lambda)\rangle \to_a C\langle c\rangle$ and $C'\langle(b', e'_\lambda)\rangle \to_b C'\langle d\rangle$ with rules $a, b \in \{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$.
We prove that it must necessarily be $a = b$, $C = C'$, and $c = d$ (up to alpha). Three cases:

- C strict initial segment of C', *i.e.* $C' = C\langle C''\rangle$ for some $C'' \neq \langle\cdot\rangle$. We show that this case is not possible: in fact, it follows that $e_\lambda = E\langle(b', e'_\lambda)\rangle$ for some E, thus (b', e'_λ) is an λ-crumble, and by proposition 8.5 it must be normal, contradicting the

hypothesis that (b', e'_λ) and c reduce with rule b.

- $C = C'$. By inspection of the reduction rules, $a = b$: in fact the rule β applies only when b is the application of an abstraction to a crumbled value, the rule sub/v only when b is a variable, and the rule $\mathsf{sub}/\mathit{left}$ only when b is the application of a variable to a crumbled value. It remains to show that $c = d$ (up to alpha): from the determinism of the lookup in the environment during sub/v and $\mathsf{sub}/\mathit{left}$ reductions.
- C' initial segment of C, *i.e.* $C = C'\langle C''\rangle$. Symmetric to the first case. ∎

To prove overhead transparency, we need the following two propositions that prove a disjointness property of variables.

Proposition 8.10 \ Free variables of left parts

Let $c = C\langle b, e\rangle$ be a crumble. Then $\mathsf{fv}(b), \mathsf{fv}(C) \subseteq \mathsf{dom}(e) \cup \mathsf{fv}(c)$.

Proof. By cases according to the definition of the crumble context C.
If $C := \langle\cdot\rangle$ then $\mathsf{fv}(C) = \emptyset \subseteq \mathsf{dom}(e) \cup \mathsf{fv}(c)$ and $c = (b, e)$, so $\mathsf{fv}(c) = (\mathsf{fv}(b) \setminus \mathsf{dom}(e)) \cup \mathsf{fv}(e)$ and hence $\mathsf{fv}(b) \subseteq (\mathsf{fv}(c) \setminus \mathsf{fv}(e)) \cup \mathsf{dom}(e) \subseteq \mathsf{fv}(c) \cup \mathsf{dom}(e)$.
Otherwise $C := (b, e'[x\leftarrow\langle\cdot\rangle])$ and then $\mathsf{fv}(C) = \mathsf{fv}(b') \cup (\mathsf{dom}(e') \setminus \{x\})$ and $c = (s, e'[x\leftarrow b]e)$; therefore, $\mathsf{fv}(c) = \mathsf{fv}(C) \cup (\mathsf{fv}(b) \setminus \mathsf{dom}(e)) \cup \mathsf{fv}(e)$ and hence $\mathsf{fv}(C) \subseteq \mathsf{dom}(e) \cup \mathsf{fv}(c)$ and $\mathsf{fv}(b) \subseteq \mathsf{dom}(e) \cup \mathsf{fv}(c)$. ∎

Proposition 8.11

In every reachable crumble $C\langle b, e\rangle$ one has $b \perp C$.

Proof. By proposition 8.10, proposition 8.8/1, and proposition 8.8/2. ∎

Proposition 8.12 \ Overhead transparency

Let c be a reachable crumble, and let $r \in \{\mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$. If $c \to_r d$ then $c_\downarrow = d_\downarrow$.

Proof. Let $c := C\langle(b, e_\lambda)\rangle \to_r C\langle(b', e_\lambda)\rangle =: d$, and let e'_λ, e''_λ such that $e_\lambda = e'_\lambda[x\leftarrow e_\lambda(x)]e''_\lambda$, noting that x does not occur in e''_λ by proposition 8.8/1 and proposition 8.11. We first prove that $(b, e_\lambda)_\downarrow = (b', e_\lambda)_\downarrow$:

- Case sub/v, *i.e.* $b := x$ and $b' = e_\lambda(x)$. By proposition 7.16/1, $(x, e'_\lambda[x\leftarrow e_\lambda(x)]e''_\lambda)_\downarrow = (x, e'_\lambda e''_\lambda)_\downarrow\{x\leftarrow(e_\lambda(x), e''_\lambda)_\downarrow\} = (e_\lambda(x), e''_\lambda)_\downarrow$ as c is well-named (proposition 8.8/1). By proposition 8.11, $\mathsf{fv}(e_\lambda(x)) \cap \mathsf{dom}(e'_\lambda[x\leftarrow e_\lambda(x)]) = \emptyset$, therefore $(e_\lambda(x), e''_\lambda)_\downarrow = (e_\lambda(x), e_\lambda)_\downarrow$, and we conclude with $(x, e_\lambda)_\downarrow = (e_\lambda(x), e_\lambda)_\downarrow$.

- Case sub/*left*, *i.e.* $b := x\,v$ and $b' = e_\lambda(x)\,v$. By proposition 7.11 $(x\,v, e_\lambda)_\downarrow = (x, e_\lambda)_\downarrow(v, e_\lambda)_\downarrow$, and we can use the point above to conclude.

We now prove that $C\langle(b, e_\lambda)\rangle_\downarrow = C\langle(b', e_\lambda)\rangle_\downarrow$ under the hypothesis that $(b, e_\lambda)_\downarrow = (b', e_\lambda)_\downarrow$. By cases on C: if $C := \langle\cdot\rangle$ just use the hypothesis. Otherwise $C := (b'', e[x{\leftarrow}\langle\cdot\rangle])$ and so $(b'', e[x{\leftarrow}b]e_\lambda)_\downarrow = (b'', ee_\lambda)_\downarrow\{x{\leftarrow}(b, e_\lambda)_\downarrow\} = (b'', ee_\lambda)_\downarrow\{x{\leftarrow}(b', e_\lambda)_\downarrow\} = (b'', e[x{\leftarrow}b']e_\lambda)_\downarrow$ by proposition 7.16/1. ∎

To prove β projection, we first prove the following property which is a direct consequence of the fact that value substitutions and evaluation commute in λ_{Plot}, *i.e.* proposition 3.4.

Proposition 8.13 \ β/v under λ-environments

Let c, d be crumbles, and let e_λ be a λ-environment. If $c_\downarrow \to_{\beta/v} d_\downarrow$, then $(c \mathbin{@} e_\lambda)_\downarrow \to_{\beta/v} (d \mathbin{@} e_\lambda)_\downarrow$.

Proof. By induction on the length of e_λ. If $e_\lambda := \epsilon$ then $(c \mathbin{@} e_\lambda)_\downarrow = c_\downarrow \to_{\beta/v} d_\downarrow = (d \mathbin{@} e_\lambda)_\downarrow$. Otherwise $e_\lambda := e'_\lambda[x{\leftarrow}v]$ where v is an abstraction (and hence $v_\downarrow$ is a λ-abstraction); by *i.h.*, $(c \mathbin{@} e'_\lambda)_\downarrow \to_{\beta/v} (d \mathbin{@} e'_\lambda)_\downarrow$ and hence $(c \mathbin{@} e_\lambda)_\downarrow = (c \mathbin{@} e'_\lambda)_\downarrow\{x{\leftarrow}v_\downarrow\} \to_{\beta/v} (d \mathbin{@} e'_\lambda)_\downarrow\{x{\leftarrow}v_\downarrow\} = (d \mathbin{@} e_\lambda)_\downarrow$ according to proposition 3.4. ∎

Proposition 8.14 \ β projection

Let c be a reachable crumble. If $c \to_\beta d$ then $c_\downarrow \to_{\beta/v} d_\downarrow$.

Proof. Let us assume that $C\langle(\lambda x.c)\,v, e_\lambda)\rangle \to_\beta C\langle(c \mathbin{@} [x{\leftarrow}v])^\alpha \mathbin{@} e_\lambda\rangle$. The crumble context C unfolds to a right v-context by proposition 8.8/4. Let $c' \mathbin{@} [x'{\leftarrow}v] := (c \mathbin{@} [x{\leftarrow}v])^\alpha$. We need to prove that $C\langle(\lambda x.c)\,v, e_\lambda)\rangle_\downarrow \to_{\beta/v} C\langle(c \mathbin{@} [x{\leftarrow}v])^\alpha \mathbin{@} e_\lambda\rangle_\downarrow$. By proposition 7.22 and proposition 8.13, it suffices to prove that $C\langle((\lambda x.c)\,v, \epsilon)\rangle_\downarrow \to_{\beta/v} C\langle c' \mathbin{@} [x'{\leftarrow}v]\rangle_\downarrow$:

$$
\begin{array}{lll}
C\langle((\lambda x.c)\,v, \epsilon)\rangle_\downarrow & = & C_\downarrow\langle(\lambda x.c_\downarrow)\,v_\downarrow\rangle \qquad \text{by theorem 7.25/1} \\
& =_\alpha & C_\downarrow\langle(\lambda x'.c'_\downarrow)\,v_\downarrow\rangle \qquad \text{by proposition 7.23} \\
& \to_{\beta/v} & C_\downarrow\langle c'_\downarrow\{x'{\leftarrow}v_\downarrow\}\rangle \qquad \text{by proposition 8.8/4} \\
& = & C_\downarrow\langle c' \mathbin{@} [x'{\leftarrow}v]_\downarrow\rangle \\
& = & C\langle c' \mathbin{@} [x'{\leftarrow}v]\rangle_\downarrow \qquad \text{by theorem 7.25/1} \\
& = & C\langle(c \mathbin{@} [x{\leftarrow}v])^\alpha\rangle_\downarrow.
\end{array}
$$

- The first use of theorem 7.25/1 requires that $C\langle((\lambda x.c)\,v, \epsilon)\rangle$ is well-named, which follows from proposition 8.8/1, and that $C \perp ((\lambda x.c)\,v, \epsilon)$, which is obvious because $\mathsf{dom}(((\lambda x.c)\,v, \epsilon)) = \emptyset$.
- The second use of theorem 7.25/1 requires that $C\langle c' \mathbin{@} [x'{\leftarrow}v]\rangle$ is well-named and that $C \perp (c' \mathbin{@} [x'{\leftarrow}v])$ *i.e.* that $\mathsf{fv}(C) \cap \mathsf{dom}(c' \mathbin{@} [x'{\leftarrow}v]) = \emptyset$: both follow

directly from the freshness condition about α-renaming in the β rule. ∎

Proposition 8.15 \ Progress

Let c be a closed crumble. If c is $\to$-normal then $c_\downarrow$ is β/v-normal.

Proof. Let c be normal. By proposition 8.5, c is a λ-crumble. By proposition 8.6, $c_\downarrow$ is an abstraction. By proposition 3.1, $c_\downarrow$ is β/v-normal. ∎

We have now proved all the requirements of the implementation theorem for the Crumble GLAM:

Theorem 8.16 \ Implementation

Let c be a reachable crumble in the Crumble GLAM.

1. *Initialization:* $\underline{t}_\downarrow = t$ for every closed λ-term t.
2. *Beta Projection:* if $c \to_\beta d$ then $c_\downarrow \to_{\beta/v} d_\downarrow$.
3. *Overhead Transparency:* if $c \to_{\mathsf{sub}} d$ then $c_\downarrow = d_\downarrow$.
4. *Determinism:* evaluation $\to$ is deterministic.
5. *Progress:* if c is $\to$normal then $c_\downarrow$ is β/v-normal.
6. *Overhead Termination:* $\to_{\mathsf{sub}}$ terminates.

Therefore the Crumble GLAM, the right-to-left Closed CbV evaluation $\to_{\beta/v}$, the crumbling translation $\underline{\bullet}$, and the readback $\bullet_\downarrow$ form an implementation system.

Proof. First, note that c is closed by proposition 8.8/2. Each point is proved separately:

1. See proposition 7.17.
2. See proposition 8.14.
3. See proposition 8.12.
4. See proposition 8.9.
5. See proposition 8.15.
6. Immediate consequence of forthcoming proposition 8.17 (proved independently).

To conclude, apply theorem 4.12. ∎

8.3 Complexity

According to the scheme presented in chapter 4, performing a single transition $\to$ in the Crumble GLAM consists of three operations:

1. *Unplugging*: locating the next redex, splitting the crumble to be reduced into a crumble context C and a crumble c that contains the redex to be fired;
2. *Rewriting*: applying a rewriting rule to the crumble c, obtaining a new crumble d;
3. *Plugging*: putting the new crumble back into the crumble context obtaining $C\langle d\rangle$.

The technical definition of plugging and unplugging of crumbles into a crumble context is quite involved and, if implemented literally, has no constant time complexity. However, this need not be the case: in section 10.1 we will introduce a slight variant of the Crumble GLAM called Pointed Crumble GLAM that removes the need for plugging and unplugging, and we show that the total cost of looking for the next redex on that machine is bilinear in the number of beta steps and the size of the initial crumble.

The second operation—rewriting the crumble to a new crumble—is identical between the two machines (the Crumble and the Pointed Crumble GLAM). In this subsection, we estimate its cost, ignoring the cost of operations 1 and 3. We show that the total cost of rewritings is also bilinear in the number of beta steps and the size of the initial crumble. The analysis carries over to the Pointed Crumble GLAM machine, which is then proved to be bilinear as well in section 10.1.

To estimate the cost of rewriting, we provide first an upper bound on the number of substitution steps ($|d|_{\mathsf{sub}/v}$ and $|d|_{\mathsf{sub}/\mathit{left}}$) in a derivation d as a function of the number of beta steps ($|d|_\beta$) and the size of the initial crumble. Then we obtain the total cost by providing an upper bound to the cost of implementing each kind of transition.

Estimation of the number of transitions Let d be a derivation (*i.e.* a sequence of reductions) in the Crumble GLAM and let $|d|_\beta, |d|_{\mathsf{sub}/v}, |d|_{\mathsf{sub}/\mathit{left}}$ be the number of $\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}$ steps in d, respectively. An easy observation is that an $\mathsf{sub}/\mathit{left}$ step can only be immediately followed by a β step (since $\to$ is deterministic), and therefore $|d|_{\mathsf{sub}/\mathit{left}} \leq |d|_\beta + 1$. To estimate $|d|_{\mathsf{sub}/v}$ we keep track of the number $|c|_{\mathsf{var}}$ of crumbles (x, e) that occurs in c outside abstractions and that will become sub/v redexes once e is evaluated to an a λ-environment. Formally:

$$|(b,e)|_{\mathsf{var}} := |b|_{\mathsf{var}} + |e|_{\mathsf{var}} \qquad |\epsilon|_{\mathsf{var}} := 0 \qquad |e[x{\leftarrow}b]|_{\mathsf{var}} := |e|_{\mathsf{var}} + |b|_{\mathsf{var}}$$
$$|x|_{\mathsf{var}} := 1 \qquad |\lambda x.c|_{\mathsf{var}} := 0 \qquad |vv'|_{\mathsf{var}} := 0$$

The key observation is that $|c|_{\mathsf{var}}$ is decreased by one by $\mathsf{sub}/\mathit{left}$ steps and increased only by β steps, that release a number of new $[x{\leftarrow}y]$ substitutions that is exactly $|d|_{\mathsf{var}}$ where d is the body of the abstraction being reduced. Since all abstractions are α-renaming of abstractions already present in the initial term by proposition 8.8/3, we obtain the expected bilinearity result.

More precisely, let $K(c) \coloneqq \sup\{|d|_{\mathsf{var}} : \lambda x.d \text{ occurs in } c\}$. Then

Proposition 8.17 \ Number of transitions for the Crumble GLAM

Let $d\colon c_0 \to^* c$ be a derivation in the Crumble GLAM. Then:

1. $|d|_{\mathsf{sub}/\mathit{left}} \leq |d|_\beta + 1$
2. $|c|_{\mathsf{var}} \leq |c_0|_{\mathsf{var}} - |d|_{\mathsf{sub}/v} + |d|_\beta \cdot (K(c_0)+1)$ and thus $|d|_{\mathsf{sub}/v} \leq |c_0|_{\mathsf{var}} + |d|_\beta \cdot (K(c_0)+1)$.

Proof.

1. If an $\mathsf{sub}/\mathit{left}$ step is followed by a reduction step, that step must be a β step.
2. By induction on the length of the derivation d. The base case ($|d| = 0$ and hence $c_0 = c$) is trivial. As for the inductive case, let $c_0 \to^* d \to c$ and reason by cases on the reduction rules:
   - sub/v: $|c|_{\mathsf{var}} = |d|_{\mathsf{var}} - 1$ and the result follows by *i.h.*.
   - $\mathsf{sub}/\mathit{left}$: $|c|_{\mathsf{var}} = |d|_{\mathsf{var}}$ and the result follows by *i.h.*.
   - β: let $d \coloneqq C\langle((\lambda x.e)\, v, e_\lambda)\rangle \to_\beta C\langle(e \mathbin{@} [x{\leftarrow}v])^\alpha \mathbin{@} e_\lambda\rangle =: c$.
     Clearly $|c|_{\mathsf{var}} \leq |d|_{\mathsf{var}} + |e|_{\mathsf{var}} + 1$; by proposition 8.8/3, $\lambda x.e$ is a subterm up to renaming of c_0, and therefore $|e|_{\mathsf{var}} \leq K(c_0)$. Therefore $|c|_{\mathsf{var}} \leq |d|_{\mathsf{var}} + K(c_0) + 1$ and the result follows by *i.h.*. ∎

Cost of single transitions. As usual, we denote by $|t|$, $|c|$, $|e|$ and $|b|$ the size of λ-terms, crumbles, environments and crumbled terms, respectively.

- The cost of each β transition (that needs to perform a copy of the crumble in the abstraction in order α-rename it) is bound by the size of the copied crumble. By proposition 8.8/3 the abstraction is the α-renaming of one the abstractions already present in the initial crumble. Therefore the cost of a β transition is bound by the size of the initial crumble or, more precisely, by $M(c_0)+1$ where $M(c) \coloneqq \sup\{|d| : \lambda x.d \text{ occurs in } c\}$.

- The cost of $\mathsf{sub}/\mathit{left}$ and sub/v transitions may change according to the choice of implementation. Following our discussion about costs on page 62, we employ a λ-graph representation by assuming the global environment to be implemented as a memory and variable occurrences to be implemented as pointers to memory locations, so that lookup in the environment can be performed in constant time in a Random Access Machine (RAM). As for the cost of actually performing the replacement of x with $e_\lambda(x)$ in the sub/v and $\mathsf{sub}/\mathit{left}$ rules, it can be done in constant time by copying the pointer to $e_\lambda(x)$ instead of the whole term. This is possible because the actual copy, corresponding to α-renaming, is done instead in the β step.

Cost of derivations. We first observe that the crumbling translation $\underline{\bullet}$ never produces substitutions of the form $[x \leftarrow y]$ (*i.e.* a variable for a variable) and therefore the following property holds:

Proposition 8.18

For every λ-term t: $M(\underline{t}) \leq 2|t|$, $K(\underline{t}) \leq 1$, and $|\underline{t}|_{\mathsf{var}} \leq 1$.

Proof. The inequality $M(\underline{t}) \leq 2|t|$ follows directly from proposition 7.28. To prove the remaining part of the statement, we also prove a corresponding statement for the auxiliary translation: for every t, $K(\bar{t}) \leq 1$ and if$(v, e) := \bar{t}$ then $|e|_{\mathsf{var}} = 0$. We proceed by induction on t:

- *Variable, i.e.* $t := x$. Then $\underline{t} = \bar{t} = (x, \epsilon)$, hence $|\bar{t}|_{\mathsf{var}} = 1$, $|\epsilon|_{\mathsf{var}} = 0$ and $K(\bar{t}) = K(\underline{t}) = 0$.
- *Abstraction, i.e.* $t := \lambda x.s$. Then $\underline{t} = \bar{t} = (\lambda x.\underline{s}, \epsilon)$; by *i.h.* $|\underline{s}|_{\mathsf{var}} \leq 1$ and $K(\underline{s}) \leq 1$ and hence $K(\underline{t}) = K(\bar{t}) = \max\{K(\underline{s}), |\underline{s}|_{\mathsf{var}}\} \leq 1$. Moreover, $|\underline{t}|_{\mathsf{var}} = |\epsilon|_{\mathsf{var}} = 0$ by definition.
- *Application, i.e.* $t := su$. Then $\underline{t} = (vv', ee')$ and $\bar{t} = (z, [z \leftarrow vv']ee')$ where $(v, e) := \bar{s}$ and $(v', e') := \bar{u}$. $K(\underline{t}) = K(\bar{t}) = \max\{K(v), K(v'), K(e), K(e')\} = \max\{K(\bar{s}), K(\bar{u})\}$, and therefore $K(\underline{t}) \leq 1$ follows from the *i.h.*. By *i.h.* it also follows that $|e|_{\mathsf{var}} = |e'|_{\mathsf{var}} = 0$; therefore $|\underline{t}|_{\mathsf{var}} = 0$ and $|ee'|_{\mathsf{var}} = 0$.

Let $d\colon c_0 \to^* c$ be a derivation and let $M_0 := M(c_0)$ and $K_0 := K(c_0)$. The cost $|d|$ of d is given by:

$$\begin{aligned}|d| &\leq (|d|_{\mathsf{sub}/v} + |d|_{\mathsf{sub}/\mathit{left}}) + |d|_\beta \cdot (M_0 + 1) \leq |c_0|_{\mathsf{var}} + |d|_\beta \cdot (K_0 + 2) + |d|_\beta \cdot (M_0 + 1) \\ &= |d|_\beta \cdot (M_0 + K_0 + 3) + |c_0|_{\mathsf{var}}.\end{aligned}$$

The complexity is bilinear in the number of beta steps and the size of the initial λ-term: when the derivation d starts from $c_0 = \underline{t}$ for any λ-term t, by proposition 8.18 we have $|d| \leq |d|_\beta \cdot (2|t| + 4) + 1$. Summing up:

Theorem 8.19 \ The Crumble GLAM is bilinear

For any closed λ-term t and any derivation $d\colon \underline{t} \to^* c$ in the Crumble GLAM, the transitions in d cost all together $O((|d|_\beta + 1) \cdot |t|)$ on a RAM.

Proof. By proposition 8.18, proposition 8.17, and the discussion above about costs. ∎

Chapter 9

Open Crumbling Evaluation

In this chapter we extend the Crumble GLAM defined in chapter 8 to the case of open terms, implementing Open CbV *i.e.* the fireball calculus λ_{fire} (section 3.2): in this way we obtain the Open Crumble GLAM. As we will see, the Crumble GLAM scales relatively easily to the open case: this is quite surprising, since the open setting is usually much trickier and requires many subtle optimizations (as discussed on section 5.2).

9.1 The Open Crumble GLAM

Evaluated environments. Before introducing the evaluation rules, we need to discuss the environments under which evaluation takes place. In the open case, λ-crumbles and λ-environments generalize to f-crumbles and f-environments, and are denoted as follows:

f-forms

f-crumbles: c_f	f-environments: e_f

Recall that in the Crumble GLAM the already evaluated coda of the environment is made out only of abstractions. Unfortunately, a syntactic characterization of f-environments (and f-crumbles) is more involved than the simple definition of λ-environments.

In the Crumble GLAM, to check whether an crumbled term b is in normal form with respect to an environment e_λ, it suffices to check whether b is an abstraction. In the open case, looking at the syntactic structure of the term is not enough: for example, the crumbled term $y\,x$ is normal with respect to the environment $[x{\leftarrow}I]$, but not with respect to the environment $[y{\leftarrow}I]$. Because of this additional complication, we are going to define f-environments directly in terms of their semantics, *i.e.* of their readback to λ-terms. Intuitively, fully evaluated f-environments should correspond to substitutions of fully evaluated λ-terms in λ_{fire}. And since by harmony normal forms in λ_{fire} are simply fireballs, it suffices to request that the readback of every entry in a f-environment is a fireball.

Let us now define f-environments formally:

Definition 9.1 \ Fireball crumbled forms

We say that e_f is a *f-environment* (resp. c_f is a *f-crumble*) if for any environment context E (resp. any crumble context C) and any crumble c such that $e_f = E\langle c\rangle$ (resp. $c_f = C\langle c\rangle$) the following two conditions hold:

1. *Readback to fireballs:* $c_{\downarrow}$ is a fireball, and
2. *Unchaining abstractions:* if $c_{\downarrow}$ is an abstraction, then $c = (v, e)$ for some abstraction v and some crumbled environment e.

Note that the second requirement is crucial for capturing the correct behaviour of the substitution rule $\to_{\mathsf{sub}/v}$, which removes the malicious chains of substitutions that we discussed in chapter 8.

Evaluation rules. The root rules of the Open Crumble GLAM are in fig. 9.1:

Rule at top-level

$$\begin{array}{llll} ((\lambda x.c)\, v, e_f) & \mapsto_{\beta} & (c \mathbin{@} [x{\leftarrow}v])^{\alpha} \mathbin{@} e_f & \\ (x, e_f) & \mapsto_{\mathsf{sub}/v} & (e_f(x), e_f) & \text{if } e_f(x) \text{ is abstraction} \\ (xv, e_f) & \mapsto_{\mathsf{sub}/\mathit{left}} & (e_f(x)\, v, e_f) & \text{if } e_f(x) \text{ is abstraction} \end{array}$$

Contextual closure

$$C\langle c\rangle \to_r C\langle d\rangle \quad \text{if } c \mapsto_r d \text{ for } r \in \{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$$

Evaluation

$$\begin{array}{rcl} \to & := & \to_{\beta} \cup \to_{\mathsf{sub}} \\ \to_{\mathsf{sub}} & := & \to_{\mathsf{sub}/v} \cup \to_{\mathsf{sub}/\mathit{left}} \end{array}$$

The Open Crumble GLAM / **Figure 9.1**

The rules $\to_{\beta}$, $\to_{\mathsf{sub}/v}$ and $\to_{\mathsf{sub}/\mathit{left}}$ are then obtained, as usual, by closing the respective root steps under crumble contexts. Evaluation in the Open Crumble GLAM is defined by:

$$\to \;:=\; \to_{\beta} \cup \to_{\mathsf{sub}/v} \cup \to_{\mathsf{sub}/\mathit{left}} .$$

Rule $\to_{\beta}$ is identical to the one in the closed case, and the comments about α-renaming given on page 116 still hold. The only unfamiliar part is the slightly different side condition of the rules $\to_{\mathsf{sub}/v}$ and $\to_{\mathsf{sub}/\mathit{left}}$. In fact the condition requires not only that a variable is defined in e_f (like in the closed case), but also additionally that the corresponding term in the environment is an abstraction.

Note that the substitution rules are conservative with respect to the corresponding rules in the closed case: in fact in the closed case if a variable x is defined in e_{λ} then $e_{\lambda}(x)$ is

necessarily an abstraction, because e_λ consists of only abstractions. In the open case instead e_f may contain also crumbled terms that are variables or applications, but this terms are not substituted by the substitution rules: this behaviour is justified by the property of λ_{fire} stated in proposition 3.9, *i.e.* the fact that the substitution of inert terms does not create or erase redexes, and hence can be avoided. Besides, avoiding the substitution of inert terms is a prerequisite for efficiency of the machine, that would otherwise be subjected to a substitution overhead due to the problem of *size explosion* (see section 4.1).

The harmony between the evaluation rules and the syntactic definition of normal forms is witnessed by proposition 9.6, but before that we provide some auxiliary propositions.

Propositions 9.2 and 9.3 capture an essential property of f-environments: $(x, e_f)_\downarrow$ is an abstraction if and only if $e_f(x)$ is an abstraction. This is a consequence of the *unchaining abstractions* requirement, and shows that the usefulness of a substitution can be correctly checked in constant time by a simple lookup in the environment.

Proposition 9.2 \ Readback to fireball

For every crumbled value v and f-environment e_f, one has that $(v, e_f)_\downarrow$ is a fireball. If moreover v is an abstraction, then $(v, e_f)_\downarrow$ is a λ-abstraction.

Proof. By induction on the length of e_f. There are two cases, depending if v is an abstraction or a variable.

- *Abstraction:* If $e_f := \epsilon$, then clearly $(v, e_f)_\downarrow = v_\downarrow$ is a λ-abstraction and hence a fireball. Otherwise $e_f := e'_f[x{\leftarrow}b]$ where b is an crumbled term such that $b_\downarrow$ is a fireball: thus, $(v, e_f)_\downarrow = (v, e'_f)_\downarrow\{x{\leftarrow}b_\downarrow\}$; by *i.h.*, $(v, e'_f)_\downarrow$ is a λ-abstraction, thus $(v, e_f)_\downarrow$ is a λ-abstraction (and so a fireball) by proposition 3.10.
- *Variable:* If $v := x \notin \mathsf{dom}(e_f)$, then $(v, e_f)_\downarrow = x$ which is a fireball; otherwise $v := x \in \mathsf{dom}(e_f)$ with $e_f := e'_f[x{\leftarrow}b]e''_f$, and then $(x, e_f)_\downarrow = (b, e''_f)_\downarrow$ (since $x \notin \mathsf{dom}(e'_f)$) is a fireball by definition of f-environment, as the f-environment $[x{\leftarrow}b]e''_f = E\langle(b, e''_f)\rangle$ with $E := [x{\leftarrow}\langle\cdot\rangle]$.

Proposition 9.3

Let e_f well-named. If $(x, e_f)_\downarrow$ is an abstraction, then $e_f(x)$ is an abstraction.

Proof. Assume by contradiction that x is not defined in e_f: then by proposition 7.15 $(x, e_f)_\downarrow = (x, \epsilon)_\downarrow = x$, contradicting the hypothesis that $(x, e_f)_\downarrow$ is an abstraction. Therefore x must be defined in e_f, *i.e.* $e_f := e'_f[x{\leftarrow}b]e''_f$ with $b := e_f(x)$. By the hypothesis that e_f is well-named, $x \notin \mathsf{dom}(e'_f)$; therefore $(x, e_f)_\downarrow = (b, e''_f)_\downarrow$ by proposition 7.16/3. By the definition of e_f, since $(b, e''_f)_\downarrow$ is an abstraction, then also b is an abstraction, and we conclude.

The following property relates f-environments and f-crumbles by means of a syntactical condition:

Proposition 9.4

Let $c = (b, e_f)$ be a well-named crumble, and let b have the following property: either b is an abstraction, or b is x or xv but x is not defined in e_f or $e_f(x)$ is not an abstraction. Then c is a f-crumble.

Proof. Let $c = (b, e_f)$ as above: it suffices to prove that $(b, e_f)_\downarrow$ is a fireball, and that if $(b, e_f)_\downarrow$ is an abstraction, then also b is an abstraction. By cases on the property about b in the hypothesis:

- Variable, *i.e.* $b = x$ for some x when x is not defined in e_f or $e_f(x)$ is not an abstraction. $(b, e_f)_\downarrow$ is a fireball by proposition 9.2. Let us now assume that $(b, e_f)_\downarrow$ is an abstraction, and show that it is not possible: in fact by proposition 9.3 $e_f(x)$ must then be defined and an abstraction, contradicting the hypothesis.
- Abstraction: nothing to prove because $(b, e_f)_\downarrow$ is an abstraction by proposition 9.2 and thus a fireball.
- Application of a variable to a crumbled value, *i.e.* $b = xv$ when x is not defined in e_f or $e_f(x)$ is not an abstraction. Note that $(b, e_f)_\downarrow = (x, e_f)_\downarrow(v, e_f)_\downarrow$, where both $(x, e_f)_\downarrow$ and $(v, e_f)_\downarrow$ are fireballs by proposition 9.2. If $(x, e_f)_\downarrow$ is inert there is nothing else to prove, because then $(b, e_f)_\downarrow$ is a fireball, and clearly not an abstraction. The case when $(x, e_f)_\downarrow$ is an abstraction is not possible: again by proposition 9.3 $e_f(x)$ should be defined and an abstraction, contradicting the hypothesis. ∎

Corollary 9.5

If the well-named crumble (b, e_f) is normal, then it is a f-crumble.

Proposition 9.6 \ Harmony for the Open Crumble GLAM

A crumble c is normal if and only if it is a f-crumble.

Proof.

($\Rightarrow$) Let $c = (b, e)$ be well-named and normal. We proceed by structural induction on e:

- if $e = \epsilon$, then (b, ϵ) is a f-crumble by corollary 9.5;
- if $e = [x \leftarrow b']e'$, then also the crumble (b', e') is normal. By *i.h.* (b', e') is a f-crumble, and therefore $e = [x \leftarrow b']e'$ is a f-environment. By corollary 9.5, $c = (b, e)$ is a f-crumble.

($\Leftarrow$) Let $c = c_\lambda$, we need to prove that c_λ is normal. Let $c_\lambda = C\langle(b, e_f)\rangle$ for some C, b, e_f. First of all, note that by the definition of c_λ, $(b, e_f)_\downarrow$ is a fireball, and that if

$(b, e_f)_\downarrow$ is an abstraction, then also b is an abstraction. We prove that no reduction rule is applicable to $C\langle(b, e_f)\rangle$:

- Rule β can be applied only if $b = (\lambda x.b')\, v$, but this contradicts the hypothesis that $(b, e_f)_\downarrow$ is a fireball, since $((\lambda x.b')\, v, e_f)_\downarrow = (\lambda x.b', e_f)_\downarrow\, (v, e_f)_\downarrow$ and $(\lambda x.b', e_f)_\downarrow$ is an abstraction by proposition 9.2.
- Rule sub can be applied only if b is some variable x and $e_f(x)$ is defined and an abstraction. This contradicts the hypothesis that if $(b, e_f)_\downarrow$ is an abstraction, then also b is an abstraction.
- Rule sub$/left$ can be applied only if $b = xv$ for some x, v, and $e_f(x)$ is defined and an abstraction. This contradicts the hypothesis that $(b, e_f)_\downarrow$ is a fireball, since $(xv, e_f)_\downarrow = (x, e_f)_\downarrow (v, e_f)_\downarrow$ and $(x, e_f)_\downarrow$ is an abstraction by proposition 9.2 because $e_f(x)$ is an abstraction.

We provide an example of evaluation of an open crumble, showing that the Open Crumble GLAM reduces a crumble that was instead stuck for the Crumble GLAM:

Example 9.7

Recall that $\delta_b := (\lambda x.xx, \epsilon)$. In example 8.2 we noted that the (open) crumble $\underline{\delta\delta(xx)}$ was stuck in the Crumble GLAM. Now instead it correctly reduces, never reaching a normal form:

$$\begin{aligned}\underline{\delta\delta(xx)} = (zw, [z{\leftarrow}\delta_b\, \delta_b][w{\leftarrow}xx]) &\to_\beta (zw, [z{\leftarrow}yy][y{\leftarrow}\delta_b][w{\leftarrow}xx]) \\ &\to_{\text{sub}/left} (zw, [z{\leftarrow}\delta_b\, y][y{\leftarrow}\delta_b][w{\leftarrow}xx]) \to \cdots\end{aligned}$$

9.2 Implementation Theorem

As expected, we now call a crumble *reachable* if it is the target of a Open Crumble GLAM evaluation starting from the translation $\underline{t}$ of a possibly open λ-term t.

The proof of the implementation theorem for the Open Crumble GLAM follows the same structure as for the Crumble GLAM in section 8.2, relying on similar but subtler invariants.

Proposition 9.8 \ Invariants for the Open Crumble GLAM

For every reachable crumble c:

1. *Freshness:* c is well-named.
2. *Disjointedness*: if $c = C\langle(b, e)\rangle$ then $b \perp C$.
3. *Subterm:* any abstraction in c is a subterm (up to renaming) of the initial crumble.

4. *Contextual decoding (weak):* for every decomposition $C\langle b, e_f\rangle$ where $(b, e_f)_\downarrow$ is not a fireball, if C'' is a prefix of C then ${C''}_\downarrow$ is a right v-context.

Proof. By induction on the length of the reduction sequence leading to the crumble c. The base cases hold by proposition 7.26 (by noting that for point 4, proposition 7.26/5 implies the weaker statement proposition 9.8/4). As for the inductive cases, we inspect each reduction rule:

1. The exponential rules sub/v and $\mathsf{sub}/\mathit{left}$ do not change the domain of the crumble, hence the claim follows from the *i.h.*. For β the claim follows from the side condition.

2. The rules sub/v and $\mathsf{sub}/\mathit{left}$ do not change the domain of the crumble and only copy to the left a value from the environment, and the claim follows from the *i.h.*.

   The rule β copies to the toplevel and renames the body of an abstraction. By the properties of α-renaming $\mathsf{fv}((c \mathbin{@} [x{\leftarrow}v])^\alpha) = \mathsf{fv}(c \mathbin{@} [x{\leftarrow}v]) = \mathsf{fv}(\lambda x.c)$. If the b is chosen to be in the crumble context (say C'') or in e_f of the reduction rule, then the claim follows from the *i.h.*. Let instead $(c \mathbin{@} [x{\leftarrow}v])^\alpha =: C'\langle(b, e')\rangle$ with $C = C''\langle C'\rangle$: then $\mathsf{fv}(C'\langle(b, e')\rangle) = \mathsf{fv}(\lambda x.c)$ and $\mathsf{fv}(b) \subseteq \mathsf{dom}(e') \cup \mathsf{fv}(\lambda x.c)$. $\mathsf{dom}(e') \cap \mathsf{dom}(C) = \emptyset$ by the side condition of β, and $\mathsf{fv}(\lambda x.c) \cap \mathsf{dom}(C) = \emptyset$ by *i.h.*, therefore we conclude with $\mathsf{fv}(b) \cap \mathsf{dom}(C) = \emptyset$.

3. The rules sub/v and $\mathsf{sub}/\mathit{left}$ introduce an abstraction, but it was already in the environment, and the claim follows from the *i.h.*. The rule β copies and renames the body of an abstraction that was already in the environment, and the claim follows from the *i.h.* since the translation commutes with the renaming of free variables (proposition 7.13/3).

4. Let $\underline{u} \to {}^n C'\langle(b', e'_f)\rangle \to_a C\langle(b, e_f)\rangle$ (where $(b, e_f)_\downarrow$ is not an abstraction). Cases of the reduction step $C'\langle(b', e'_f)\rangle \to_a C\langle(b, e_f)\rangle$:

   - Case β: $C'\langle((\lambda x.c)v, e_f)\rangle \to_\beta C'\langle c^\alpha \mathbin{@} ([x^\alpha{\leftarrow}v]e_f)\rangle$.
     Let C'' be a prefix of C. There are two sub-cases:
     * *C'' is a prefix of C'*: by *i.h.* ${C''}_\downarrow$ is a right v-context.
     * *C' is a prefix of C'', i.e.* $C'' = C'\langle C'''\rangle$ and $c^\alpha = C'''\langle c'\rangle$. By proposition 7.26/4 and proposition 9.8/3 c is the translation of a λ-term, by proposition 7.13/3 c^α is so, and thus by proposition 7.26/5 ${C'''}_\downarrow$ is a right v-context. By *i.h.*, ${C'}_\downarrow$ is a right v-context as well. Since ${C''}_\downarrow = {C'}_\downarrow\langle{C'''}_\downarrow\rangle$ according to theorem 7.25/2, we obtain that ${C''}_\downarrow$ is a right v-context as composition of right v-contexts (proposition 3.2).
   - Case sub/v: it follows from the *i.h.* since C is necessarily a prefix of C'.
   - Case $\mathsf{sub}/\mathit{left}$: it follows from the *i.h.*, since $e'_f = e_f$ and $C = C'$.

To prove the implementation theorem (theorem 9.16) we first prove separately its require-

ments: determinism, transparency, projection, and progress.

Proposition 9.9 \ Determinism of crumbled reduction

$\to$ is deterministic.

Proof. Assume that there exists a crumble that may be decomposed in two ways $C\langle(b, e_f)\rangle = C'\langle(b', e'_f)\rangle$ such that they reduce respectively $C\langle(b, e_f)\rangle \to_a C\langle c\rangle$ and $C'\langle(b', e'_f)\rangle \to_b C'\langle d\rangle$ with rules $a, b \in \{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$.
We prove that it must necessarily be $a = b$, $C = C'$, and $c = d$ (up to α-equivalence). Three cases:

- C strict initial segment of C', *i.e.* $C' = C\langle C''\rangle$ for some $C'' \neq \langle\cdot\rangle$. We show that this case is not possible: in fact, it follows that $e_f = E\langle(b', e'_f)\rangle$ for some E, thus (b', e'_f) is a f-crumble, and by proposition 9.6 it must be normal, contradicting the hypothesis that (b', e'_f) and c reduce with rule b.
- $C = C'$. By inspection of the reduction rules, $a = b$: in fact β applies only when b is the application of an abstraction to a crumbled value, sub/v only when b is a variable, and $\mathsf{sub}/\mathit{left}$ only when b is the application of a variable to a crumbled value. It remains to show that $c = d$ (up to alpha): this follows from the determinism of the lookup in the environment during sub/v and $\mathsf{sub}/\mathit{left}$ reductions.
- C' initial segment of C, *i.e.* $C = C'\langle C''\rangle$. Symmetric to the first case. ∎

Proposition 9.10 \ Overhead transparency

Let c reachable and $a \in \{\mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$. If $c \to_a d$ then $c_\downarrow = d_\downarrow$.

Proof. Let $c := C\langle(b, e_f)\rangle \to_a C\langle(b', e_f)\rangle =: d$, and let e'_f, e''_f be such that $e_f = e'_f[x{\leftarrow}e_f(x)]e''_f$, noting that x does not occur in e''_f by proposition 9.8/1 and proposition 9.8/2. We first prove that $(b, e_f)_\downarrow = (b', e_f)_\downarrow$:

- Case sub/v, *i.e.* $b := x$ and $b' = e_f(x)$. By proposition 7.16/1, $(x, e'_f[x{\leftarrow}e_f(x)]e''_f)_\downarrow = (x, e'_f e''_f)_\downarrow\{x{\leftarrow}(e_f(x), e''_f)_\downarrow\} = (e_f(x), e''_f)_\downarrow$ as c is well-named (proposition 9.8/1). By proposition 9.8/2, $\mathsf{fv}(e_f(x)) \cap \mathsf{dom}(e'_f[x{\leftarrow}e_f(x)]) = \emptyset$, therefore $(e_f(x), e''_f)_\downarrow = (e_f(x), e_f)_\downarrow$, and we conclude with $(x, e_f)_\downarrow = (e_f(x), e_f)_\downarrow$.
- Case $\mathsf{sub}/\mathit{left}$, *i.e.* $b := xv$ and $b = e_f(x)\, v$. Since $(xv, e_f)_\downarrow = (x, e_f)_\downarrow(v, e_f)_\downarrow$, we can use the point above to conclude.

We now prove that $C\langle(b, e_f)\rangle_\downarrow = C\langle(b', e_f)\rangle_\downarrow$ under the hypothesis that $(b, e_f)_\downarrow = (b', e_f)_\downarrow$. By cases on C: if $C := \langle\cdot\rangle$ just use the hypothesis. Otherwise $C := (b'', e[x{\leftarrow}\langle\cdot\rangle])$ and so $(b'', e[x{\leftarrow}b]e_f)_\downarrow = (b'', ee_f)_\downarrow\{x{\leftarrow}(b, e_f)_\downarrow\} = (b'', ee_f)_\downarrow\{x{\leftarrow}(b', e_f)_\downarrow\} = (b'', e[x{\leftarrow}b']e_f)_\downarrow$ by proposition 7.16/1. ∎

To prove projection (proposition 9.14) we rely on the notion of substitution σ_e induced by an environment, useful to factor out from a crumble the f-environment on its right.

Given an environment e, the simultaneous substitution σ_e associated with e is defined by induction on the length of e as follows:[1]

$$\sigma_\epsilon := \{\} \qquad\qquad \sigma_{[x\leftarrow b]e} := \{x\leftarrow(b,e)_\downarrow\} \cup \sigma_e\,.$$

The substitution induced by a f-environment is a fireball substitution:

Proposition 9.11 \ Substitution of fireballs

If $e_f := [x_1\leftarrow b_1]\dots[x_n\leftarrow b_n]$ is a f-environment, then $\sigma_{e_f} = \{x_1\leftarrow s_1,\dots,x_n\leftarrow s_n\}$ where all the s_i's are fireballs.

Proof. By induction on the length of e. If $e_f := \epsilon$, then $\sigma_{e_f} = \{\}$ and the statement is vacuously true. Otherwise $e_f := [x\leftarrow b]e'_f$ with $e'_f := [x_1\leftarrow b_1]\dots[x_n\leftarrow b_n]$ and then $\sigma_{e_f} = \{x\leftarrow(b,e'_f)_\downarrow\} \cup \sigma_{e'_f}$; by *i.h.* (since e'_f is a f-environment), $\sigma_{e'_f} = \{x_1\leftarrow s_1,\dots,x_n\leftarrow s_n\}$ where all the b''_is are fireballs; also $(b,e'_f)_\downarrow$ is a fireball by definition of f-environment, as $e_f = E\langle(b,e')\rangle$ with $E := [x\leftarrow\langle\cdot\rangle]$; therefore, $\sigma_{e_f} = \{x\leftarrow(b,e'_f)_\downarrow, x_1\leftarrow s_1,\dots,x_n\leftarrow s_n\}$ satisfies the statement.

Proposition 9.12 \ Readback *vs* append

For any crumble c and f-environment e_f, $(c \mathbin{@} e_f)_\downarrow = c_\downarrow\sigma_{e_f}$.

Proof. By induction on the length of e_f. If $e_f := \epsilon$ then $\sigma_{e_f} = \{\}$ and hence $(c \mathbin{@} e_f)_\downarrow = c_\downarrow = c_\downarrow\{\} = c_\downarrow\sigma_{e_f}$. Otherwise $e_f := [x\leftarrow b]e'_f$ where $x \notin \mathsf{dom}(e'_f)$; by *i.h.* (since e'_f is a fireball environment), $((c \mathbin{@} [x\leftarrow b]) \mathbin{@} e'_f)_\downarrow = (c \mathbin{@} [x\leftarrow b])_\downarrow\sigma_{e'_f}$ and $(b,e'_f)_\downarrow = ((b,\epsilon) \mathbin{@} e'_f)_\downarrow = (b,\epsilon)_\downarrow\sigma_{e'_f} = b_\downarrow\sigma_{e'_f}$; by the definitions of append and readback, $(c \mathbin{@} [x\leftarrow b])_\downarrow = c_\downarrow\{x\leftarrow b_\downarrow\}$; therefore, $(c \mathbin{@} e_f)_\downarrow = ((c \mathbin{@} [x\leftarrow b]) \mathbin{@} e'_f)_\downarrow = (c \mathbin{@} [x\leftarrow b])_\downarrow\sigma_{e'_f} = (c_\downarrow\{x\leftarrow b_\downarrow\})\sigma_{e'_f} = c_\downarrow(\{x\leftarrow b_\downarrow\sigma_{e'_f}\} \cup \sigma_{e'_f}) = c_\downarrow\{x\leftarrow(b,e'_f)_\downarrow\} \cup \sigma_{e'_f} = c_\downarrow\sigma_{e_f}$.

The following proposition lifts proposition 3.11 from λ_{fire} to the crumbled setting, and is fundamental in the proof of beta projection: it basically states that β/v steps are transformed to β/f steps under the influence of f-environments.

Proposition 9.13

If $c_\downarrow \to_{\beta/v} d_\downarrow$, then $(c \mathbin{@} e_f)_\downarrow \to_{\beta/f} (d \mathbin{@} e_f)_\downarrow$.

Proof. According to proposition 9.11, $\sigma_{e_f} = \{x_1\leftarrow f_1,\dots,x_n\leftarrow f_n\}$ where $f_1,\dots,f_n$ are fireballs. By proposition 9.12 and proposition 3.11, $(c \mathbin{@} e_f)_\downarrow = c_\downarrow\sigma_{e_f} \to_{\beta/f} d_\downarrow\sigma_{e_f} = (d \mathbin{@} e_f)_\downarrow$

[1] This definition of σ_e is slightly different from definition 4.8 previously provided in this dissertation, but equivalent.

Note that proposition 9.13 does not hold if we replace $\to_{\beta/v}$ with $\to_{\beta/f}$ in the hypothesis. Indeed, take $c := ((\lambda x.(x,\epsilon))y, [y{\leftarrow}(zz,\epsilon)])$ and $d := (y, [y{\leftarrow}zz])$ and $e_f := [z{\leftarrow}\lambda x.(xx,\epsilon)]$: then, $c_\downarrow = (\lambda x.x)(zz) \to_{\beta/f} zz = d_\downarrow$ but $(c \mathbin{@} e_f)_\downarrow = (\lambda x.x)((\lambda x.xx)\lambda x.xx) \not\to_{\beta/f} (\lambda x.xx)\lambda x.xx = (d \mathbin{@} e_f)_\downarrow$. The problem is essentially due to the fact that fireballs, contrary to values, are not closed by substitution: this a notable difference between the closed case (where the normal forms coincide with closed values) and the open case (where the normal forms coincide with fireballs).

Proposition 9.14 \ Beta projection

Let c be a reachable crumble. If $c \to_\beta d$ then $c_\downarrow \to_{\beta/f} d_\downarrow$.

Proof. Let us assume that $C\langle(\lambda x.c)\, v, e_f)\rangle \to_\beta C\langle(c \mathbin{@} [x{\leftarrow}v])^\alpha \mathbin{@} e_f\rangle$. C unfolds to a right v-context by proposition 9.8/4. Let $e \mathbin{@} [y{\leftarrow}v] := (c \mathbin{@} [x{\leftarrow}v])^\alpha$. Let us first consider $C\langle(\lambda x.c)\, v, \epsilon)\rangle$:

$$\begin{aligned}
C\langle((\lambda x.c)\, v, \epsilon)\rangle_\downarrow &= C_\downarrow\langle(\lambda x.c_\downarrow)\, v_\downarrow\rangle && \text{by theorem 7.25/1}\\
&=_\alpha C_\downarrow\langle(\lambda y.e_\downarrow)\, v_\downarrow\rangle\\
&\to_{\beta/v} C_\downarrow\langle e_\downarrow\{y{\leftarrow}v_\downarrow\}\rangle\\
&= C_\downarrow\langle e \mathbin{@} [y{\leftarrow}v]_\downarrow\rangle\\
&= C\langle e \mathbin{@} [y{\leftarrow}v]\rangle_\downarrow && \text{by theorem 7.25/1}\\
&= C\langle(c \mathbin{@} [x{\leftarrow}v])^\alpha\rangle_\downarrow.
\end{aligned}$$

Note that using theorem 7.25/1 in the last step requires to show that $C \perp (e \mathbin{@} [y{\leftarrow}v])$, which holds because the side condition in the β rule.
Therefore, $C\langle((\lambda x.c)\, v, \epsilon)\rangle_\downarrow \to_{\beta/v} C\langle e \mathbin{@} [y{\leftarrow}v]\rangle_\downarrow$ and, by propositions 7.22 and 9.13, $C\langle((\lambda x.c)v, e_f)\rangle_\downarrow \to_{\beta/f} C\langle e \mathbin{@} [y{\leftarrow}v]e_f\rangle_\downarrow$. ∎

Proposition 9.15 \ Progress

If c is normal then $c_\downarrow$ is $\to_{\beta/f}$-normal.

Proof. By proposition 9.6, if c is normal then it is a f-crumble *i.e.* $c = c_f$. By definition of c_f, $c_\downarrow$ is a fireball. By harmony for λ_{fire}, $c_\downarrow$ is β/f-normal. ∎

We have now proved all the requirements of the implementation theorem for the Open Crumble GLAM:

Theorem 9.16 \ Implementation

Let c be a reachable crumble in the Open Crumble GLAM.

1. *Initialization:* $\underline{t}_\downarrow = t$ for every λ-term t.
2. *Beta Projection:* if $c \to_\beta d$ then $c_\downarrow \to_{\beta/f} d_\downarrow$.

3. *Overhead transparency:* if $c \to_{\mathsf{sub}} d$ then $c_{\downarrow} = d_{\downarrow}$.
4. *Determinism:* evaluation $\to$ is deterministic.
5. *Progress:* If c is $\to$-normal then $c_{\downarrow}$ is β/f-normal.
6. *Overhead termination:* $\to_{\mathsf{sub}}$ terminates.

Therefore the Open Crumble GLAM, the right-to-left fireball evaluation $\to_{\beta/f}$, the crumbling translation $\underline{\bullet}$, and the readback $\bullet_{\downarrow}$ form an implementation system.

Proof. Each point is proved separately:

1. See proposition 7.17.
2. See proposition 9.14.
3. See proposition 9.10.
4. See proposition 9.9.
5. See proposition 9.15.
6. Immediate consequence of forthcoming proposition 9.17 (proved independently).

To conclude, apply theorem 4.12. ∎

9.3 Complexity

The complexity analysis is identical to the one in section 8.3. Indeed, once the search for the next redex is neglected, the two machines only differ by the side condition for substitution steps $\to_{\mathsf{sub}/v}$ and $\to_{\mathsf{sub}/\mathit{left}}$.

Recall that a derivation d is a sequence of reduction steps. Moreover, we use $|d|_{\beta}$, $|d|_{\mathsf{sub}/v}$, $|d|_{\mathsf{sub}/\mathit{left}}$ to count the number of respectively β, sub/v, $\mathsf{sub}/\mathit{left}$ reduction steps in d.

Proposition 9.17 \ Number of transitions for the Open Crumble GLAM

Let $d\colon c_0 \to^* c$ be a derivation in the Open Crumble GLAM. Then:

1. $|d|_{\mathsf{sub}/\mathit{left}} \leq |d|_{\beta} + 1$
2. $|c|_{\mathsf{var}} \leq |c_0|_{\mathsf{var}} - |d|_{\mathsf{sub}/v} + |d|_{\beta} \cdot (K(c_0) + 1)$ and thus $|d|_{\mathsf{sub}/v} \leq |c_0|_{\mathsf{var}} + |d|_{\beta} \cdot (K(c_0) + 1)$.

Proof.

1. If an $\mathsf{sub}/\mathit{left}$ step is followed by a reduction step, that step must be a β step.

2. By induction on the length of the derivation d. The base case ($|d| = 0$ and hence $c_0 = c$) is trivial. As for the inductive case, let $c_0 \to^* d \to c$ and reason by cases on the reduction rules:
   - sub/v: $|c|_{\mathsf{var}} = |d|_{\mathsf{var}} - 1$ and the result follows by *i.h.*.
   - sub/$left$: $|c|_{\mathsf{var}} = |d|_{\mathsf{var}}$ and the result follows by *i.h.*.
   - β: let $d := C\langle((\lambda x.e)\, v, e_f)\rangle \to_\beta C\langle(e \mathbin{@} [x \leftarrow v])^\alpha \mathbin{@} e_f\rangle =: c$.
     Clearly $|c|_{\mathsf{var}} \leq |d|_{\mathsf{var}} + |e|_{\mathsf{var}} + 1$; by proposition 9.8/3, $\lambda x.e$ is a subterm up to renaming of c_0, and therefore $|e|_{\mathsf{var}} \leq K(c_0)$. Therefore $|c|_{\mathsf{var}} \leq |d|_{\mathsf{var}} + K(c_0) + 1$ and the result follows by *i.h.*.

Cost of single transitions. The cost of each β transition (that needs to perform a copy of the body of the abstraction in order α-rename it) is bound by $M(c_0)$.

For the cost of sub/$left$ and sub/v transitions we assume as usual that variables are implemented as pointers, and therefore lookup and access in the environment can be performed in constant time. For the cost of performing the substitution of the term from the environment, we can assume that it amounts to the copy of a pointer, as in the closed case. Therefore, the cost of the derivation is:

$$\begin{aligned} |d| &\leq (|d|_{\mathsf{sub}/v} + |d|_{\mathsf{sub}/left}) + |d|_\beta \cdot (M_0 + 1) \\ &\leq |c_0|_{\mathsf{var}} + |d|_\beta \cdot (K_0 + 2) + |d|_\beta \cdot (M_0 + 1) \\ &= |d|_\beta \cdot (M_0 + K_0 + 3) + |c_0|_{\mathsf{var}} \end{aligned}$$

Again, the complexity improves when starting from a λ-term: by proposition 8.18, $|d| \leq |d|_\beta \cdot (2|t| + 4) + 1$. Summing up:

Theorem 9.18 \ The Open Crumble GLAM is reasonable & efficient

For any derivation $d\colon \underline{t} \to^* c$ in the Open Crumble GLAM, the transitions in d cost all together $O((|d|_\beta + 1) \cdot |t|)$ on a RAM.

Proof. By proposition 8.18 (which is a property of the translation $\underline{\bullet}$, independently from the abstract machine), proposition 9.17, and the discussion above about costs.

Chapter 10

Pointed Crumbling Machines

As we have already discussed on page 115, we have been calling the Crumble GLAMs "machines" even though they do not satisfy the usual requirements for abstract machines. One of these requirements is the presence of rules that guide evaluation to the next redex: in the Crumble GLAMs we left implicit the search for the next redex in the evaluation rules, because searching a redex simply corresponds to traversing the environment from right to left.

In order to implement the Crumble GLAMs while respecting the complexity analysis that we presented in sections 8.3 and 9.3, we introduce variants of the Crumble GLAMs called Pointed Crumble GLAMs, where the search for the next redex is decomposed into $O(1)$ transitions called src (for *search*). The other machine transitions are in a one-to-one correspondence with the ones of the Crumble GLAMs. Finally, we prove that the overall number of the search steps is bilinear in the number of β steps and the size of the initial term, establishing bilinearity of the Pointed Crumble GLAMs.

10.1 The Pointed Crumble GLAM

The key idea to turn the Crumble GLAM into the Pointed Crumble GLAM is to avoid plugging and unplugging in the definition of transition rules: we obtain so by letting transitions act on *pointed crumbles*, which are crumbles where the beginning of the evaluated coda is explicitly marked using a pointer. For instance, the crumble (b, ee_λ) could be represented as $(b, e \blacktriangleleft e_\lambda)$ where $\blacktriangleleft$ is the explicit separator that must be followed by λ-environments only.

A pointed crumble $(b, e[x{\leftarrow}b'] \blacktriangleleft e_\lambda)$ is the machine state that is attempting to reduce the crumble $C\langle b', e_\lambda\rangle$ with respect to the evaluation context $C = (b, e[x{\leftarrow}\langle\cdot\rangle])$. If (b', e_λ) is a Crumble GLAM r-redex (for some rule $r \in \{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$), the Pointed Crumble GLAM will transition according to the corresponding r-transition that also takes care of setting (in $O(1)$) the pointer to the rightmost unevaluated crumble. Otherwise, by harmony (proposition 8.5), b' must be a crumbled value v and therefore the pointer is moved (in $O(1)$) one step to the left, looking for the next redex: $(b, e[x{\leftarrow}v] \blacktriangleleft e_\lambda) \to_{\mathsf{src}} (b, e \blacktriangleleft [x{\leftarrow}v]e_\lambda)$.

Not all pointed crumble configurations are of the form $(b, [x{\leftarrow}b'] \blacktriangleleft e_\lambda)$: the configura-

tions $(b, \epsilon \blacktriangleleft e_\lambda)$ must be also taken into account and reduced if b is not a crumbled value. However, there is no simple way to describe machine transitions that act uniformly on both configurations $(b, \epsilon \blacktriangleleft e_\lambda)$ and $(b, e[x{\leftarrow}b'] \blacktriangleleft e_\lambda)$ without duplicating the rules or without reintroducing a notion of contextual closure. To solve the issue, we abandon pointed crumbles and adopt pointed environments instead.

A pointed environment $([x{\leftarrow}b]e \blacktriangleleft e_\lambda)$ is just a representation of a pointed crumble $(b, e \blacktriangleleft e_\lambda)$. The leftmost variable x in a pointed environment is a fresh variable that can be understood as the name given to the machine output. It plays a role similar to the outermost λ-abstraction introduced by CPS transformations, that binds the continuation that is fed with the output of the evaluation. In particular, a normal pointed environment $(\epsilon \blacktriangleleft [x{\leftarrow}v]e_\lambda)$ represents the normal crumble (v, e_λ).

Pointed environments and readback. Pointed environments are defined as:

Pointed environments

$$e_\blacktriangleleft ::= e \blacktriangleleft e'$$

where e and e' are non-pointed environments such that either e or e' is non-empty. The environment on the left of the cursor $\blacktriangleleft$ is the *unevaluated* part; the one on the right is the *evaluated* part.

The *encoding* $\iota(\bullet)$ embeds crumbles into pointed environments:

Encoding $\iota(\bullet)$

$$\iota(b, e) := [x{\leftarrow}b]e \blacktriangleleft \epsilon$$

where x is any variable name fresh in b and e.

The left inverse of $\iota(\bullet)$ is the readback function $(\bullet)_\Downarrow$ from pointed environments to crumbles, defined as follows:

Decoding $(\bullet)_\Downarrow$

$$(\epsilon \blacktriangleleft [x{\leftarrow}b]e)_\Downarrow := (b, e) \qquad ([x{\leftarrow}b]e \blacktriangleleft e')_\Downarrow := (b, ee').$$

Evaluation. The transitions of the Pointed Crumble GLAM are in fig. 10.1:

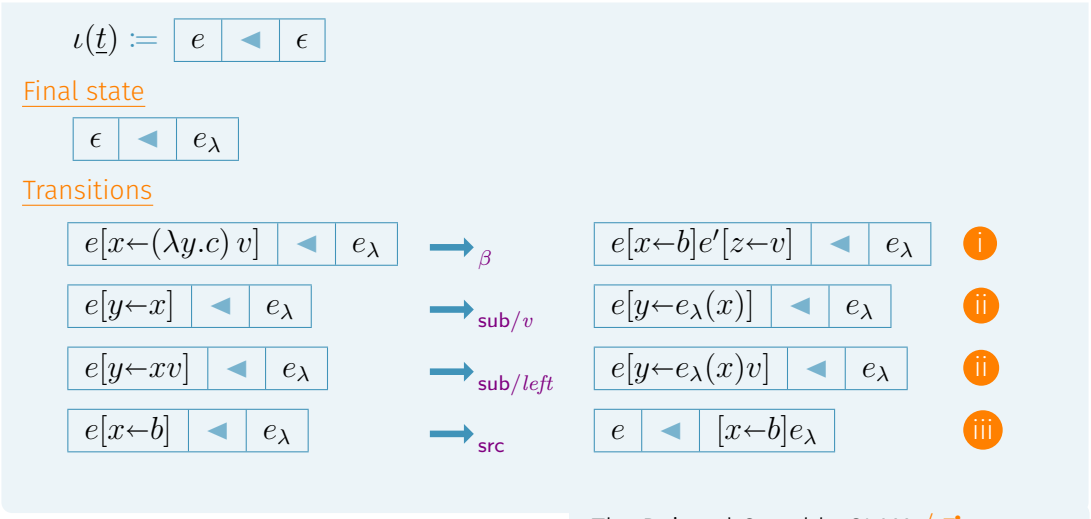
Syntax

$$\begin{array}{rrcl} \text{Environments} & e & ::= & \epsilon \mid e[x{\leftarrow}b] \\ & e_\lambda & ::= & \epsilon \mid [x{\leftarrow}v]e_\lambda \end{array}$$

States

e	$\blacktriangleleft$	e_λ

Initial state



The Pointed Crumble GLAM / **Figure 10.1**

where

i $\lambda z.(b, e') \coloneqq (\lambda y.c)^\alpha$ such that $(e[x\leftarrow b]e'[z\leftarrow v] \blacktriangleleft e_\lambda)$ is well-named.

ii if $x \in \mathsf{dom}(e_\lambda)$.

iii if none of the other rules is applicable, *i.e.* when b is an abstraction or when b is x or xv but x is not defined in e_λ.

Execution in the Pointed Crumble GLAM is then defined as:

$$\to \;\coloneqq\; \to_\beta \cup \to_{\mathsf{sub}/v} \cup \to_{\mathsf{sub}/left} \cup \to_{\mathsf{src}} .$$

Simulation. We prove that the Pointed Crumble GLAM simulates the Crumble GLAM, following the standard schema introduced in section 4.4 and that we have already employed for the Crumble GLAM in section 8.3. An important difference here is that principal transitions do not only consist of β transitions, but also of **sub** ones (because they are the ones that we want to simulate), while **src** transitions are overhead.

As usual, we call a pointed environment *reachable* if it is obtained through transitions starting from the encoding $\iota(c)$ of a well-named closed crumble c.

First of all, we prove some properties that are invariant under the execution of the Pointed Crumble GLAM:

Proposition 10.1 \ Invariants for the Pointed Crumble GLAM

Let $e_{\blacktriangleleft}$ be a reachable pointed environment in the Pointed Crumble GLAM:

1. *Freshness:* $e_{\blacktriangleleft}$ is well-named.
2. *Closure:* $e_{\blacktriangleleft}$ is closed.
3. *Rightmost:* $e_{\blacktriangleleft} = (e \blacktriangleleft e_\lambda)$ for some environment e and some λ-environment e_λ.

Proof. By induction on the length of the execution sequence leading to $e_{\blacktriangleleft}$. The base cases hold by the definition of reachability and by the definition of $\iota(\bullet)$. As for the inductive cases, we proceed by cases on the transitions:

1. For β the claim follows from the side condition. The rules in $\{\mathsf{sub}/v, \mathsf{sub}/\mathit{left}, \mathsf{src}\}$ do not change the domain of the pointed environment, hence the claim follows from the *i.h.*
2. Similar to the discussion in proposition 8.8.
3. The rules in $\{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$ do not change the evaluated part, hence the claim follows from the *i.h.*. As for the src rule, it suffices to prove that (b, e_λ) is an λ-crumble, knowing that the other rules in $\{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$ cannot be applied: this follows from proposition 8.3.

Directly from the "rightmost" property above it follows a notion of *harmony* for the Pointed Crumble GLAM: final states are exactly those where the unevaluated environment is empty.

Proposition 10.2 \ Harmony for the Pointed Crumble GLAM

A reachable pointed environment $e_{\blacktriangleleft}$ is normal iff it has the form $(\epsilon \blacktriangleleft e_\lambda)$ for some non-empty λ-environment e_λ.

Proof. The proof of the implication from right to left is trivial. Let us now prove the other direction. Let $e_{\blacktriangleleft}$ a reachable normal pointed environment. By proposition 10.1, $e_{\blacktriangleleft}$ has the form $(e \blacktriangleleft e_\lambda)$. e cannot be non-empty, because otherwise one of the transitions in $\{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}, \mathsf{src}\}$ could be applied, contradicting the hypothesis that $e_{\blacktriangleleft}$ is normal.

The following two are small technical propositions needed in the proof of the implementation theorem below:

Proposition 10.3

For every pointed environment $(e[x{\leftarrow}b] \blacktriangleleft e')$: $(e[x{\leftarrow}b] \blacktriangleleft e')_{\Downarrow} = C\langle b, e'\rangle$ where C is:

$$C := \begin{cases} \langle\cdot\rangle & \text{if } e = \epsilon \\ (b', e''[x{\leftarrow}\langle\cdot\rangle]) & \text{if } e := [y{\leftarrow}b']e'' \end{cases}.$$

Proof. Easy by the definition of $(\bullet)_{\Downarrow}$.

Proposition 10.4

If $[x{\leftarrow}b]e$ is a λ-environment, then (b, e) is a λ-crumble.

Proof. Obvious from the definition of λ-environment and λ-crumble. ∎

We have now proved all the requirements of the implementation theorem for the Pointed Crumble GLAM:

Theorem 10.5 \ Implementation

Let $e \blacktriangleleft$ be a pointed environment reachable by the Pointed Crumble GLAM.

1. *Initialization:* $(\iota(c))_{\Downarrow} = c$ for every crumble c.
2. *Principal Projection:* if $e \blacktriangleleft \to_r e' \blacktriangleleft$ then $(e \blacktriangleleft)_{\Downarrow} \to_r (e' \blacktriangleleft)_{\Downarrow}$ for any rule $r \in \{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$.
3. *Overhead Transparency:* if $e \blacktriangleleft \to_{\mathsf{src}} e' \blacktriangleleft$ then $(e \blacktriangleleft)_{\Downarrow} = (e' \blacktriangleleft)_{\Downarrow}$.
4. *Determinism:* evaluation $\to$ is deterministic.
5. *Progress:* if $e \blacktriangleleft$ is normal then $(e \blacktriangleleft)_{\Downarrow}$ is normal.
6. *Overhead Termination*: $\to_{\mathsf{src}}$ terminates.

Therefore, the Pointed Crumble GLAM, the Crumble GLAM evaluation $\to$, the injection $\iota(\bullet)$, and the readback $(\bullet)_{\Downarrow}$ form an implementation system.

Proof.

1. Let $c = (b, e)$: by the definitions, $(\iota(c))_{\Downarrow} = ([x{\leftarrow}b]e \blacktriangleleft \epsilon)_{\Downarrow} = (b, e\epsilon) = c$.
2. By cases on the transitions:
   - Case β: suppose $e[x{\leftarrow}(\lambda y.c)\, v] \blacktriangleleft e_\lambda \to_\beta e[x{\leftarrow}b]e'[z{\leftarrow}v] \blacktriangleleft e_\lambda$. By proposition 10.3, $(e[x{\leftarrow}(\lambda y.c)\, v] \blacktriangleleft e_\lambda)_{\Downarrow} = C\langle((\lambda y.c)\, v, e_\lambda)\rangle$ and $(e[x{\leftarrow}b]e'[z{\leftarrow}v] \blacktriangleleft e_\lambda)_{\Downarrow} = C\langle b, e'[z{\leftarrow}v]e_\lambda\rangle$ for some crumble context C. Clearly also $C\langle((\lambda y.c)\, v, e_\lambda)\rangle \to_\beta C\langle(b, e'[z{\leftarrow}v]e_\lambda)\rangle$.
   - Case sub/v: suppose $e[y{\leftarrow}x] \blacktriangleleft e_\lambda \to_{\mathsf{sub}/v} e[y{\leftarrow}e_\lambda(x)] \blacktriangleleft e_\lambda$. By proposition 10.3, $(e[y{\leftarrow}x] \blacktriangleleft e_\lambda)_{\Downarrow} = C\langle x, e_\lambda\rangle$ and $(e[y{\leftarrow}e_\lambda(x)] \blacktriangleleft e_\lambda)_{\Downarrow} = C\langle(e_\lambda(x), e_\lambda)\rangle$ for some crumble context C. Clearly also $C\langle(x, e_\lambda)\rangle \to_{\mathsf{sub}/v} C\langle(e_\lambda(x), e_\lambda)\rangle$.
   - Case $\mathsf{sub}/\mathit{left}$: suppose $e[y{\leftarrow}xv] \blacktriangleleft e_\lambda \to_{\mathsf{sub}/\mathit{left}} e[y{\leftarrow}e_\lambda(x)v] \blacktriangleleft e_\lambda$. By proposition 10.3, $(e[y{\leftarrow}xv] \blacktriangleleft e_\lambda)_{\Downarrow} = C\langle(xv, e_\lambda)\rangle$ and $(e[y{\leftarrow}e_\lambda(x)v] \blacktriangleleft e_\lambda)_{\Downarrow} = C\langle(e_\lambda(x)v, e_\lambda)\rangle$ for some crumble context C. Clearly also $C\langle(xv, e_\lambda)\rangle \to_{\mathsf{sub}/\mathit{left}} C\langle(e_\lambda(x)\, v, e_\lambda)\rangle$.
3. By inspection of the rule src. We need to prove that $(e[x{\leftarrow}b] \blacktriangleleft e_\lambda)_{\Downarrow} = (e \blacktriangleleft [x{\leftarrow}b]e_\lambda)_{\Downarrow}$. By cases on the structure of e: if $e = \epsilon$, then $(\epsilon[x{\leftarrow}b] \blacktriangleleft e_\lambda)_{\Downarrow} =$

$(b, e_\lambda) = (\epsilon \blacktriangleleft [x{\leftarrow}b]e_\lambda)_{\Downarrow}$. If instead $e = [y{\leftarrow}b']e'$, then $(e[x{\leftarrow}b] \blacktriangleleft e_\lambda)_{\Downarrow} = (b', e'[x{\leftarrow}b]e_\lambda) = (e \blacktriangleleft [x{\leftarrow}b]e_\lambda)_{\Downarrow}$.

4. The rule **src** can be applied by definition only when the other rules cannot be applied. The rules $\{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$ apply to a pointed environment of the form $(e[x{\leftarrow}b] \blacktriangleleft e_\lambda)$ for distinct shapes of b, *i.e.* when b is respectively the application of an abstraction to a crumbled value, a variable, and the application of a variable to a crumbled value. In order to show that no single rule in $\{\mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$ can transition to different pointed environments, it suffices to remember that the lookup of a variable in the environment is deterministic by the assumption that the environment is well-named (proposition 10.1).

5. By proposition 10.2, $e_\blacktriangleleft$ is normal iff it has the form $(\epsilon \blacktriangleleft e_\lambda)$ for some non-empty λ-environment e_λ. Then $(e_\blacktriangleleft)_{\Downarrow} = (\epsilon \blacktriangleleft e_\lambda)_{\Downarrow}$ which is an λ-crumble by proposition 10.4. By proposition 8.5, $(e_\blacktriangleleft)_{\Downarrow}$ is normal.

6. Immediate consequence of forthcoming corollary 10.8 (proved independently).

To conclude, apply theorem 4.12. ∎

Complexity We reuse the complexity measures introduced in section 8.3. The only difference here is that we need to bound also the number of **src** steps, which intuitively depends on the length of the pointed environment that is being evaluated. We define the new measure $|\bullet|_{\mathsf{len}}$ on environments and crumbles, that simply counts the number of entries:

$$|b, e|_{\mathsf{len}} := 1 + |e|_{\mathsf{len}} \qquad |\epsilon|_{\mathsf{len}} := 0 \qquad |e[x{\leftarrow}b]|_{\mathsf{len}} := 1 + |e|_{\mathsf{len}}.$$

Note that after each β step, the length of a pointed environment increases by the length of the body of the abstraction that is being reduced. For every environment e, we also define the constant $L(e)$ that bounds the length of the bodies of abstractions occurring anywhere in e:

$$L(e) := \sup\{|c|_{\mathsf{len}} : \lambda x.c \text{ in } e\}.$$

We extend the definitions above to pointed environments and crumbles in the expected way:

$$|e \blacktriangleleft e'|_{\mathsf{len}} := |ee'|_{\mathsf{len}} \qquad L(e \blacktriangleleft e') := L(ee') \qquad L(c) := L(\iota(c)).$$

Remark 10.6

For any crumble c and λ-term t: $|c|_{\mathsf{len}} = |\iota(c)|_{\mathsf{len}}$, $L(\underline{t}) \leq |t| + 1$ and $|\underline{t}|_{\mathsf{len}} \leq |t| + 1$.

We denote by ρ an execution of the Pointed Crumble GLAM (as usual, a sequence of transitions), and we use $|\rho|_\beta$, $|\rho|_{\mathsf{sub}/v}$, $|\rho|_{\mathsf{sub}/\mathit{left}}$, $|\rho|_{\mathsf{src}}$ to count the number of respectively β, sub/v, $\mathsf{sub}/\mathit{left}$, **src** transitions in ρ.

By theorem 10.5, β, sub/v and $\mathsf{sub}/\mathit{left}$ transitions are mapped one-to-one to corresponding steps in the Crumble GLAM. As for the number of **src** transitions:

Proposition 10.7 \ Number of **src** transitions

Let c be a well-named crumble, and let $\rho\colon \iota(c) \to^* e \blacktriangleleft = (e \blacktriangleleft e_\lambda)$ an execution of the Pointed Crumble GLAM. Then $|e|_{\mathsf{len}} \leq |c|_{\mathsf{len}} + |\rho|_\beta \cdot L(c) - |\rho|_{\mathsf{src}}$.

Proof. By induction on the length of ρ. In the base case $|e|_{\mathsf{len}} = |c|_{\mathsf{len}}$ and $|\rho|_\beta = |\rho|_{\mathsf{src}} = 0$. In the inductive case, use the *i.h.* and proceed by cases on the transitions. The transitions $\mathsf{sub}/v, \mathsf{sub}/\mathit{left}$ do not change the length of the environments. A **src** transition decreases by 1 the length of the unevaluated part. A β transition increases the length of the unevaluated part by a number bound by $L(c)$.

As an easy consequence of remark 10.6 and proposition 10.7, we obtain the following bound on **src** transitions:

Corollary 10.8

Let t be a λ-term. For a normalizing execution ρ of the Pointed Crumble GLAM starting from $\iota(\underline{t})$, we have $|\rho|_{\mathsf{src}} \leq (|\rho|_\beta + 1) \cdot (|t| + 1)$.

Cost of evaluation. We already discussed the cost of β, sub/v and $\mathsf{sub}/\mathit{left}$ transitions in section 8.3. The cost of a **src** transition is clearly $O(1)$.

Theorem 10.9 \ The Pointed Crumble GLAM is reasonable & efficient

For any λ-term t and any execution $\rho\colon \iota(\underline{t}) \to^* e \blacktriangleleft$ of the Pointed Crumble GLAM, the transitions in ρ cost all together $O((|\rho|_\beta + 1) \cdot |t|)$ on a RAM.

Proof. The cost $|\rho|$ of ρ is the the total cost of β, $\mathsf{sub}/v, \mathsf{sub}/\mathit{left}$ transitions (which was proved in section 8.3 to be bilinear in the number of β steps (plus one) and the size of the initial crumble when starting from λ-terms) plus the total cost of **src** transitions. The cost of **src** transitions in ρ is big-O of the number of **src** transitions, and therefore by corollary 10.8, again bilinear in the number of β steps and the size of the initial term (both plus one).

OCaml implementation. An implementation in OCaml of the Pointed Crumble GLAM can be found in chapter 11, together with the code that implements the crumbling translation. That chapter also discusses in detail a parsimonious choice of data structures for the implementation of pointed environments.

10.2 The Open Pointed Crumble GLAM

In this section we provide the *Open Pointed Crumble GLAM*, a machine for open terms that decomposes the search for the next redex in commutative $O(1)$ steps following the same pattern as the Pointed Crumble GLAM in section 10.1 for the closed case. In particular, the definitions of pointed environment, encoding $\iota(\bullet)$ (from crumbles to pointed environments) and decoding $(\bullet)_{\Downarrow}$ (from pointed environments to crumbles) are the same. The transitions of the Open Pointed Crumble GLAM differ from the ones of the Pointed Crumble GLAM exactly as the transitions of Open Crumble GLAM differ from the ones of the Crumble GLAM.

Evaluation. The transitions of the Open Pointed Crumble GLAM are in fig. 10.2:

Syntax

Environments $e ::= \epsilon \mid e[x \leftarrow b]$
$e_f ::= \epsilon \mid [x \leftarrow b]e_f$

States

$e \mid \blacktriangleleft \mid e_f$

Initial state

$\iota(\underline{t}) := e \mid \blacktriangleleft \mid \epsilon$

Final state

$\epsilon \mid \blacktriangleleft \mid e_f$

Transitions

$e[x \leftarrow (\lambda y.c)\, v] \mid \blacktriangleleft \mid e_f$	$\to_{\beta}$	$e[x \leftarrow b]e'[z \leftarrow v] \mid \blacktriangleleft \mid e_f$	i
$e[y \leftarrow x] \mid \blacktriangleleft \mid e_f$	$\to_{\text{sub}/v}$	$e[y \leftarrow e_f(x)] \mid \blacktriangleleft \mid e_f$	ii
$e[y \leftarrow xv] \mid \blacktriangleleft \mid e_f$	$\to_{\text{sub}/left}$	$e[y \leftarrow e_f(x)v] \mid \blacktriangleleft \mid e_f$	ii
$e[x \leftarrow b] \mid \blacktriangleleft \mid e_f$	$\to_{\text{src}}$	$e \mid \blacktriangleleft \mid [x \leftarrow b]e_f$	iii

The Open Pointed Crumble GLAM / **Figure 10.2**

where

i $\lambda z.(b, e') := (\lambda y.c)^{\alpha}$ such that $(e[x \leftarrow b]e'[z \leftarrow v] \blacktriangleleft e_f)$ is well-named.

ii if $x \in \mathsf{dom}(e_f)$ and $e_f(x)$ is an abstraction.

iii if none of the other rules is applicable, *i.e.* when b is an abstraction or when b is x or xv but x is not defined in e_f or $e_f(x)$ is not an abstraction.

As in the Open Crumble GLAM, then only difference with respect to the corresponding closed machine is in the condition ii, where we also require that $e_f(x)$ is an abstraction.

Again, we call a pointed environment *reachable* (in the Open Pointed Crumble GLAM) if it is obtained through transitions starting from the encoding $\iota(c)$ of a well-named crumble c.

First of all we prove the following invariants for the Open Pointed Crumble GLAM; the proof proceeds like the one of proposition 10.1, with the only difference that we drop the property of $e_{\blacktriangleleft}$ being closed.

Proposition 10.10 \ Invariants for the Open Pointed Crumble GLAM

Let $e_{\blacktriangleleft}$ be a reachable pointed environment:

1. *Freshness:* $e_{\blacktriangleleft}$ is well-named;
2. *Rightmost:* $e_{\blacktriangleleft} = (e \blacktriangleleft e_f)$ for some e and some f-environment e_f.

Proof. By induction on the length of the execution sequence leading to $e_{\blacktriangleleft}$. The base cases hold by the definition of reachability and by the definition of $\iota(\bullet)$. As for the inductive cases, we proceed by cases on the transitions:

1. For β the claim follows from the side condition. The rules in $\{\mathsf{sub}/v, \mathsf{sub}/\mathit{left}, \mathsf{src}\}$ do not change the domain of the pointed environment, hence the claim follows from the *i.h.*
2. The rules in $\{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$ do not change the evaluated part, hence the claim follows from the *i.h.*. As for the **src** rule, it suffices to prove that (b, e_f) is a f-crumble, knowing that the other rules in $\{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$ cannot be applied: this follows from proposition 9.4.

Also the Open Pointed Crumble GLAM enjoys a form of harmony:

Proposition 10.11 \ Harmony for the Open Pointed Crumble GLAM

Let $e_{\blacktriangleleft}$ a reachable pointed environment in the Open Pointed Crumble GLAM: $e_{\blacktriangleleft}$ is normal if and only if it has the form $(\epsilon \blacktriangleleft e_f)$ for some non-empty f-environment e_f.

Proof. The proof of the implication from right to left is trivial. Let us now prove the other direction. Let $e_{\blacktriangleleft}$ a reachable normal pointed environment. By proposition 10.10, $e_{\blacktriangleleft}$ has the form $(e \blacktriangleleft e_f)$. e cannot be non-empty, because otherwise one of the transitions in $\{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}, \mathsf{src}\}$ could be applied, contradicting the hypothesis that $e_{\blacktriangleleft}$ is normal.

It follows a technical proposition corresponding to proposition 10.4, which then leads to the implementation theorem right below.

Proposition 10.12

If $[x{\leftarrow}b]e$ is a f-environment, then (b, e) is a f-crumble.

Proof. Let $(t, e) = C\langle(b', e')\rangle$ for some C, b', e'. Then $[x{\leftarrow}b]e = E\langle C\langle b', e'\rangle\rangle$ for $E := [x{\leftarrow}\langle\cdot\rangle]$. The requirements for a f-crumble follow from the definition of f-environment for $[x{\leftarrow}b]e$.

Theorem 10.13 \ Implementation

Let $e_{\blacktriangleleft}$ a reachable pointed environment in the Open Pointed Crumble GLAM.

1. *Initialization:* $(\iota(c))_{\Downarrow} = c$.
2. *Principal Projection:* if $e_{\blacktriangleleft} \to_r e'_{\blacktriangleleft}$ then $(e_{\blacktriangleleft})_{\Downarrow} \to_r (e'_{\blacktriangleleft})_{\Downarrow}$ for any transition $r \in \{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$
3. *Overhead Transparency:* if $e_{\blacktriangleleft} \to_{\mathsf{src}} e'_{\blacktriangleleft}$ then $(e_{\blacktriangleleft})_{\Downarrow} = (e'_{\blacktriangleleft})_{\Downarrow}$.
4. *Determinism:* $\to$ is deterministic.
5. *Progress:* if $e_{\blacktriangleleft}$ is normal, then $(e_{\blacktriangleleft})_{\Downarrow}$ is normal.
6. *Overhead Termination:* $\to_{\mathsf{src}}$ terminates.

Therefore, the Open Pointed Crumble GLAM, the Open Crumble GLAM evaluation $\to$, the injection $\iota(\bullet)$, and the readback $(\bullet)_{\Downarrow}$ form an implementation system.

Proof.

1. Let $c = (b, e)$: by the definitions, $(\iota(c))_{\Downarrow} = ([x{\leftarrow}b]e \blacktriangleleft \epsilon)_{\Downarrow} = (b, e\epsilon) = c$.
2. By cases on the transitions:
   - Case β: suppose $e[x{\leftarrow}(\lambda y.c)v] \blacktriangleleft e_f \to_\beta e[x{\leftarrow}t]e'[z{\leftarrow}v] \blacktriangleleft e_f$. By proposition 10.3, $(e[x{\leftarrow}(\lambda y.c)v] \blacktriangleleft e_f)_{\Downarrow} = C\langle((\lambda y.c)v, e_f)\rangle$ and $(e[x{\leftarrow}b]e'[z{\leftarrow}v] \blacktriangleleft e_f)_{\Downarrow} = C\langle(b, e'[z{\leftarrow}v]e_f)\rangle$ for some crumble context C. Clearly also $C\langle((\lambda y.c)v, e_f)\rangle \to_\beta C\langle(b, e'[z{\leftarrow}v]e_f)\rangle$.
   - Case sub/v: suppose $e[y{\leftarrow}x] \blacktriangleleft e_f \to_{\mathsf{sub}/v} e[y{\leftarrow}e_f(x)] \blacktriangleleft e_f$. By proposition 10.3, $(e[y{\leftarrow}x] \blacktriangleleft e_f)_{\Downarrow} = C\langle x, e_f\rangle$ and $(e[y{\leftarrow}e_f(x)] \blacktriangleleft e_f)_{\Downarrow} = C\langle e_f(x), e_f\rangle$ for some crumble context C. Clearly also $C\langle(x, e_f)\rangle \to_{\mathsf{sub}/v} C\langle e_f(x), e_f\rangle$.
   - Case $\mathsf{sub}/\mathit{left}$: suppose $e[y{\leftarrow}xv] \blacktriangleleft e_f \to_{\mathsf{sub}/\mathit{left}} e[y{\leftarrow}e_f(x)\, v] \blacktriangleleft e_f$. By proposition 10.3, $(e[y{\leftarrow}xv] \blacktriangleleft e_f)_{\Downarrow} = C\langle(xv, e_f)\rangle$ and $(e[y{\leftarrow}e_f(x)\, v] \blacktriangleleft e_f)_{\Downarrow} = C\langle(e_f(x)\, v, e_f)\rangle$ for some crumble context C. Clearly also $C\langle(xv, e_f)\rangle \to_{\mathsf{sub}/\mathit{left}} C\langle(e_f(x)\, v, e_f)\rangle$.
3. By inspection of the rule src. We need to prove that $(e[x{\leftarrow}b] \blacktriangleleft e_f)_{\Downarrow} = (e \blacktriangleleft [x{\leftarrow}b]e_f)_{\Downarrow}$. By cases on the structure of e: if $e = \epsilon$, then $(\epsilon[x{\leftarrow}b] \blacktriangleleft e_f)_{\Downarrow} = (b, e_f) = (\epsilon \blacktriangleleft [x{\leftarrow}b]e_f)_{\Downarrow}$. If instead $e = [y{\leftarrow}b']e'$, then $(e[x{\leftarrow}b] \blacktriangleleft e_f)_{\Downarrow} = (b', e'[x{\leftarrow}b]e_f) = (e \blacktriangleleft [x{\leftarrow}b]e_f)_{\Downarrow}$.

4. The rule **src** can be applied by definition only when the other rules cannot be applied. The rules $\{\beta, \mathsf{sub}/v, \mathsf{sub}/\mathit{left}\}$ apply to a pointed environment of the form $(e[x \leftarrow b] \blacktriangleleft e_f)$ for distinct shapes of b, *i.e.* when b is respectively the application of an abstraction to a crumbled value, a variable, and the application of a variable to a crumbled value. In order to show that no substitution can transition in two different ways, it suffices to remember that the lookup of a variable in the environment is deterministic by the assumption that the environment is well-named (proposition 10.10).

5. By proposition 10.11, $e_{\blacktriangleleft}$ is normal iff it has the form $(\epsilon \blacktriangleleft e_f)$ for some non-empty f-environment e_f. Then $(e_{\blacktriangleleft})_{\Downarrow} = (\epsilon \blacktriangleleft e_f)_{\Downarrow}$ which is a f-crumble by proposition 10.12. By proposition 9.6, $(e_{\blacktriangleleft})_{\Downarrow}$ is normal.

6. Immediate consequence of forthcoming corollary 10.15 (proved independently).

To conclude, apply theorem 4.12. ∎

Complexity We reuse the measures $|\bullet|_{\mathsf{len}}$ and $L(c)$ introduced for the Pointed Crumble GLAM on page 146. We also reuse the usual notations $|\rho|_\beta$, $|\rho|_{\mathsf{sub}/v}$, $|\rho|_{\mathsf{sub}/\mathit{left}}$, $|\rho|_{\mathsf{src}}$ to count the number of respectively β, sub/v, $\mathsf{sub}/\mathit{left}$, **src** transitions in an execution ρ of the Open Crumble GLAM.

By theorem 10.13, β, sub/v and $\mathsf{sub}/\mathit{left}$ transitions are mapped one-to-one to corresponding reduction steps in the Open Crumble GLAM. As for the number of **src** transitions:

Proposition 10.14 \ Number of **src** transitions

Let $\rho \colon \iota(c) \to^* e_{\blacktriangleleft} = (e \blacktriangleleft e_f)$ an execution of the Open Pointed Crumble GLAM. Then $|e|_{\mathsf{len}} \leq |c|_{\mathsf{len}} + |\rho|_\beta \cdot L(c) - |\rho|_{\mathsf{src}}$.

Proof. By induction on the length of ρ. In the base case $|e|_{\mathsf{len}} = |c|_{\mathsf{len}}$ and $|\rho|_\beta = |\rho|_{\mathsf{src}} = 0$. In the inductive case, use the *i.h.* and proceed by cases on the transitions. The transitions $\mathsf{sub}/v, \mathsf{sub}/\mathit{left}$ do not change the length of the environments. A **src** transition decreases by 1 the length of the unevaluated part. A β transition increases the length of the unevaluated part by a number bound by $L(c)$. ∎

Corollary 10.15

For a normalizing execution ρ starting from $\iota(\underline{t})$, we have $|\rho|_{\mathsf{src}} \leq (|\rho|_\beta + 1) \cdot (|t| + 1)$.

Cost of evaluation. The cost of β, sub/v and $\mathsf{sub}/\mathit{left}$ transitions was already discussed in section 9.3. The cost of a **src** transition is clearly $O(1)$.

Theorem 10.16 \ The Open Pointed Crumble GLAM is reasonable & efficient

For any execution $\rho\colon \iota(\underline{t}) \to^* e_{\blacktriangleleft}$ in the Open Pointed Crumble GLAM, the transitions in ρ cost all together $O((|\rho|_\beta + 1) \cdot |t|)$ on a RAM.

Proof. The cost $|\rho|$ of ρ is the the total cost of β, sub/v, $\mathsf{sub}/\mathit{left}$ transitions (which was proved in section 9.3 to be bilinear in the number of β steps (plus one) and the size of the initial crumble when starting from λ-terms) plus the total cost of src transitions. The cost of src transitions in ρ is big-O of the number of src transitions, and therefore by corollary 10.15, again bilinear in the number of β steps and the size of the initial term (both plus one). ∎

OCaml implementation. An implementation in OCaml of the Open Pointed Crumble GLAM can be found in chapter 11. The OCaml code for the Open Pointed Crumble GLAM is identical to the one for the Pointed Crumble GLAM but for three lines: two lines implement the additional check for values in the substitution rule, and one considers also inert terms in the search rule.

Chapter 11

OCaml Implementation

The goal of this section is implementing in OCaml the Pointed Crumble GLAM and Open Pointed Crumble GLAM presented in sections 10.1 and 10.2. Before picking a concrete implementation, we are going to describe the abstract requirements of the data structures. The two machines share the same data structures and auxiliary functions, and only differ by three lines of code.

Data structures. The machine works on pointed environments of the form $(e \blacktriangleleft e_\lambda)$. The two environments are subject to different requirements:

- the unevaluated part e is extended only on the right during the β rule by appending an α-renamed unevaluated environment. Because α-renaming requires at least linear time, appending does not need to be faster than $O(n)$ where n is the length of the environment to be appended.

  Only the rightmost entry is inspected by every reduction rule. Finally, the **src** reduction step removes the rightmost entry, moving it to the evaluated part. Therefore the unevaluated environment must implement the stack interface, allowing to perform push, pop and topmost inspection in constant time.

- As for the evaluated part e_λ, reduction rules never exploit the sequential structure of e_λ. On the contrary, the **sub**$/v$ and **sub**$/left$ rules need to access the entry associated with a variable x in constant time. The only other operation required (by the **src** rule) is to add an entry to it in constant time.

To satisfy lookup in constant time for e_λ, we implement e_λ as a store, thus ignoring its list structure. In turn, this choice impacts on the data structure for terms, because it forces (occurrences of) variables to be implemented as pointers to memory locations. More explicitly, an entry $[x \leftarrow b]$ is represented as a node n which is a record containing a field `content` holding b together with additional fields soon to be described. If α is the address of n, occurrences of x are represented in OCaml as $\texttt{Shared}(n)$, which means a memory crumble tagged as `Shared` and pointing to α.

Occurrences of λ-bound variables are instead presented as `Var`(v) where v is a unique identifier for that variable. Thus the data structure for terms is:

```
...
and term =
 | Var of var
 | Lam of var * crumble
 | App of term * term
 | Shared of node
...
```

Because a variable can point both to a evaluated or unevaluated entry, the unevaluated environment e must consist of nodes as well. The simplest implementation of a stack is a linked list of nodes. Therefore we add to each node a field `prev` used to point to the previous entry in the environment, and we identify e with the pointer to its first node. An additional field `copying` and mutability of all fields are required to implement α-renaming in linear time; we explain their use later. Therefore a node is:

```
type node =
 { mutable content : term
 ; mutable copying : bool
 ; mutable prev : node option }
...
```

Unreachable nodes can be garbage-collected by the runtime of OCaml. Because the evaluator holds a pointer to the unevaluated environment, only evaluated nodes can be garbage-collected.

A crumble should be a term together with an unevaluated environment. For the same reason of section 10.1, we implement the crumble (b, e) as the unevaluated environment $\iota(b, e) = [x \leftarrow b]e$ where x is a fresh variable. Therefore the `crumble` datatype is just an environment, *i.e.* a pointer to a node:

```
...
and crumble = node
...
```

Finally, we implement the datatype of unique λ-bound variable identifiers `var` as the address of an OCaml record that holds no useful information. Thus comparing variables can be achieved using pointer equality `==`. Concrete implementations can add fields to the record, for example to associate the name of the variable as a string.

```
type var = { dummy: unit } (* no empty records in OCaml *)
```

A summary of the data structures can be found in fig. 11.1.

```
type var = { dummy : unit }
type node =
 { mutable content : term
```

```
 ; mutable copying : bool
 ; mutable prev : node option }
and crumble = node
and term =
 | Var of var
 | Lam of var * crumble
 | App of term * term
 | Shared of node

let mk_node content =
 { content ; copying = false ; prev = None }

let push n e =
 n.prev <- Some e
```

Data structures / **Figure 11.1**

Implementation of reduction rules. The code that implements reduction in the closed and open cases can be found in fig. 11.2. The evaluation functions `eval_c`/`eval_o` take in input an unevaluated environment, *i.e.* a node `n`.

The code that implements the rule β for $(\lambda y.e)b$ is the most complex because it must:

1. α-rename $\lambda y.e$ to $(\lambda y.e)^{\alpha} =: \lambda y'.e'$. Let $e' = [y'_1 \leftarrow b'_1] \dots [y'_k \leftarrow b'_k]$.
2. change the top of the unevaluated environment (stack) `n` to $[\mathsf{n} \leftarrow b'_1]$ and append to it $[y'_2 \leftarrow b'_2] \dots [y'_k \leftarrow b'_k]$
3. push $[y' \leftarrow b]$ on top of the unevaluated environment

To implement the previous steps efficiently, the code creates the node y' pointing to b and then calls a function `copy_env` $y\ y'$ `n` e that performs at once steps 1 and 2 in linear time, returning the new unevaluated environment e'. Finally `push` $y'\ e'$ pushes y' on top of e'.

The sub/*left* and sub/v rules just update the content of the top of the unevaluated environment in the required way.

The src rule is implemented by the function `pop` that pops the top of the unevaluated environment and calls evaluation on the new top, if present. Otherwise a normal form is reached, and evaluation returns the term that, pointing to the evaluated environment, forms the normal crumble. When a node is popped, its `prev` pointer is unset to facilitate garbage-collection of unreferenced nodes.

Let us remark that the implementation is tail-recursive[1]; since OCaml optimizes tail-recursion, the machine only consumes constant space on the process execution stack.

[1] when the `pop` function is inlined

As a minor optimization to the expected code, our implementation merges execution of rule sub/v with that of the src step which always follows the former. The merging is obtained calling pop in place of eval_c/eval_o.

```
let rec eval_c n =
 match n.content with
 | App(Lam(y, e), t) ->
   (* rule β *)
   let y' = mk_node t in
   let e' =
    copy_env y y' n e in
   push y' e';
   eval_c y'
 | App
    (Shared
     {content=t1}
    , t2) ->
   (* rule sub/left *)
   n.content <- App(t1, t2);
   eval_c n
 | Shared
    {content=l} ->
   (* rule sub/v *)
   n.content <- l;
   pop n

 | Lam _ ->
   (* rule src *)
   pop n
 | Var _ | App(Var _, _)
   -> failwith "Open"
 | App(_ ,App _)
 | App(App _,_)
   -> assert false
and pop n =
 match n.prev with
 | None -> n.content
 | Some p ->
    n.prev <- None;
    eval_c p
```

```
let rec eval_o n =
 match n.content with
 | App(Lam(y, e), t) ->
   (* rule β *)
   let y' = mk_node t in
   let e' =
    copy_env y y' n e in
   push y' e';
   eval_o y'
 | App
    (Shared
     {content=(Lam _ as t1)}
    , t2) ->
   (* rule sub/left *)
   n.content <- App(t1, t2);
   eval_o n
 | Shared
    {content=Lam _ as l} ->
    (* rule sub/v *)
    n.content <- l;
    pop n
 | Shared _
 | App(Shared _, _)
 | Var _ | App(Var _, _)
 | Lam _ ->
    (* rule src *)
    pop n

 | App(_ ,App _)
 | App(App _,_)
    -> assert false
and pop n =
 match n.prev with
 | None -> n.content
 | Some p ->
    n.prev <- None;
    eval_o p
```

Evaluation: closed (left) vs open (right) / **Figure 11.2**

The complexity of each case is $O(1)$, but for the multiplicative rule which requires a re-

naming of environment (`copy_env`). It remains to see how this operation can be implemented with linear complexity.

Implementation of α-renaming. We implement α-renaming of unevaluated environments by creating a copy of the environment. The representation in memory of the environment is a DAG because terms in the nodes of the environment contain occurrences of `Shared` nodes defined in the same environment. Therefore we need to implement a copy algorithm over DAGs that runs in linear time.

The algorithm consists in using the content field of nodes to perform the renaming. When a node is being copied, it is temporarily put in the `copying=true` status, and its `content` field is changed to point to the corresponding new node. Then, the rest of the environment is copied recursively. When an occurrence of a node that is being copied is found in the term being copied, it is replaced with the new node stored in the `content` field of the old one. Finally, when the copy is over, the `copying` status of every node is reset to `false` and the previous value of `content` is restored, yielding the original environment.

The auxiliary `copying_node y y' f` function, where `y` is the node to be copied to `y'`, implements the idea above by temporarily putting `y` in `copying=true` status, until `f` is executed.

```
let copying_node y y' f =
  let saved = y.content in
  y.content <- Shared y' ;
  y.copying <- true ;
  let res = f () in
  y.content <- saved ;
  y.copying <- false ;
  res
```

The `copy_env y y' n e` function (fig. 11.3) not only implements the algorithm above by copying `e`, but it also replaces occurrences of `Var y` (the bound variable in a $\rightarrow_\beta$ redex) with `Shared y'` (the new node pointing to the argument of the redex) and it reuses the node `n` as the last node in the new environment. *i.e.* let $e := [x_0 \leftarrow b_0] \dots [x_k \leftarrow b_k]$; then `copy_env y y' n e` $= [\mathtt{n} \leftarrow b'_1][y'_2 \leftarrow b'_2] \dots [y'_k \leftarrow b'_k]$, where $b'_i := b_i\{y \leftarrow y'\}\{x_{i+1} \leftarrow x'_{i+1}\} \dots \{x_k \leftarrow x'_k\}$.

```
let copy_env y y' n e =
  let rec copy_term = function
  | Var v when v == y -> Shared y'
  | Shared {content; copying} when copying -> content
  | Var _ | Shared _ as t -> t
  | App(t1,t2) -> App(copy_term t1,copy_term t2)
  | Lam(v,e) -> Lam(v,copy_env_aux e)
  and copy_env_aux ?n e =
    let rec map e =
      let t' = copy_term e.content in
```

```
      match e.prev with
      | None ->
        (match n with
         | None -> mk_node t'
         | Some n' -> n'.content <- t' ; n')
      | Some p ->
         let n' = mk_node t' in
         let e' = copying_node e n' (fun () -> map p) in
         push n' e' ; n'
    in map e
  in copy_env_aux ~n e
```

The `copy_env` function / **Figure 11.3**

The implementation of `copy_env` just calls `map`, that iterates over the environment copying it. Only the last node is handled specially, to reuse the node `n` if provided. `map` is recursive with `copy_term`, that iterates over terms and calls `copy_env` under abstractions without handling the last element in a special way. The auxiliary function `copy_env_aux` is a technical trick to use an optional argument `?n` to factorize the code between the two handlings of the last element of the environment (*i.e.* inside vs outside abstractions).

Crumbling. The code in fig. 11.4 takes an unevaluated environment and returns the corresponding crumbled unevaluated environment. It generalizes the function $\underline{\bullet}$ that translates λ-terms into crumbles.

It consists in three mutually recursive functions. The first two take in input `this` which is the new environment being computed.

1. `to_crumble this t` computes $\underline{t} = (b', e)$, appends e to `this` and returns b'.
2. `aux_crumble this t` computes the auxiliary translation $\overline{t} = (v, e)$.
3. `aux_env e` creates a copy of `e` in crumbled form in linear time reusing the same trick of the `copying` flag as in α-renaming.

```
let anf e =
   let dummy = Var { dummy = () } in
   let rec to_crumble this = function
   | Var _ as t -> t, this
   | App(t1, t2) ->
      let t1, this = aux_crumble this t1 in
      let t2, this = aux_crumble this t2 in
      App(t1, t2), this
   | Lam(v, e) -> let e = aux_env e in Lam(v, e), this
```

```
   | Shared n -> n.content, this
   and aux_crumble this = function
   | App _ as t ->
      let n = mk_node dummy in
      push n this;
      let t, this' = to_crumble n t in
      n.content <- t;
      Shared n, this'
   | Var _ | Lam _ | Shared _ as t -> to_crumble this t
   and aux_env e =
     let n = mk_node dummy in
     let t', last = to_crumble n e.content in
     n.content <- t' ;
     (match e.prev with
      | None -> ()
      | Some p ->
         let e = copying_node e n (fun () -> aux_env p) in
         push n e) ;
     last
   in aux_env e
;;
```

The `anf` function / **Figure 11.4**

Part III

Comparison

This part of the dissertation is about Comparison, *i.e.* how to compare efficiently λ-terms with sharing.

As discussed in chapter 4, in order to perform evaluation in an efficient way it is fundamental to *share* subterms along computation. The notion of *useful* evaluation mandates that the result of evaluation is a compact, shared representation of the normal form, which cannot be unfolded without causing an exponential-time blow up known as size explosion (section 4.1).

Succint representations. Being a compact encoding of the unfolded normal form, one may wonder if one can still perform on shared terms the usual operations (notably, compare them) without introducing an exponential overhead.

In fact, a common problem for succint representations of data structures is that the complexity of algorithms operating on them may increase dramatically (Papadimitriou, 1994): even algorithms with low computation complexity on λ-terms may become slow on their compact encoding, simply because their running time is now considered as a function of the length of the shared λ-terms, which can be exponentially smaller.

Shared λ-terms have been shown to be acceptable succint encodings of λ-terms by Accattoli and Lago, 2014: terms with ES can be compared for equality in time *quadratic* in the size of the shared terms, hence there is no blow up. However, in this dissertation we are focused on *linear*-time complexity: our motivation is that we require our abstract machines to run in *bilinear* time, *i.e.* linear in the size of the initial term to be evaluated and the number of β-steps. When combining evaluation through abstract machines with a *sharing equality* algorithm in order to obtain a bilinear Conversion algorithm, the complexity of sharing equality has to match the complexity of evaluation: any sharing equality algorithm whose running time is super-linear in the size of the evaluated terms spoils the resulting overall complexity, which becomes not bilinear anymore.

In this part of the dissertation we provide our linear-time algorithm for sharing equality, the first one with such low complexity in the literature. Our algorithm is based on the algorithm by Paterson and Wegman, 1978 for first-order unification, which also runs in linear-time. It is not at all evident whether an algorithm for first-order unification can be adapted to the case of λ-terms, which have a notion of scoping and binders that is not present in first-order terms. One of our main contributions is to show that this is in fact possible: one can split the problem of sharing equality in two subproblems, one which only accounts for the first-order structure of a shared λ-term, and the other also taking into account the binder structure.

λ-graphs. In this part we use the representation of shared λ-terms as λ-graphs that we have introduced in chapter 6. The reason for this is that Paterson-Wegman's algorithm operates on a graph data structure, and cannot be reformulated easily in an algebraic way. However, as discussed on page 62, we required the same underlying λ-graphs representation also for our abstract machines: in order to prove that abstract machines with global environments can be implemented with bilinear-time complexity on a RAM, variables bound in the environment must be implemented as pointers to the corresponding term in the environment entry, thus forcing a λ-graph-like representation in memory. This is why our abstract

machines basically output λ-graphs that can be compared for sharing equality by our upcoming algorithm.

Outline. This part is structured as follows:

- In chapter 12 we develop our theory of sharing equality. We call *sharing equivalences* the binary relations over a λ-graph that capture the sharing equality of related nodes: they are basically bisimulations plus an additional requirement for free variable nodes. We focus on how to define and construct the smallest sharing equivalences, by means of *propagating* a given input query over the structure of the λ-graph.
- In chapter 13, we showcase problems related to sharing equality, showing also their computational complexity.
- In chapter 14 we provide our linear-time algorithm for sharing equality, together with full proofs of correctness, completeness, termination and linearity. We also show that our sharing equality algorithm actually computes the smallest sharing equivalence over a λ-graph.

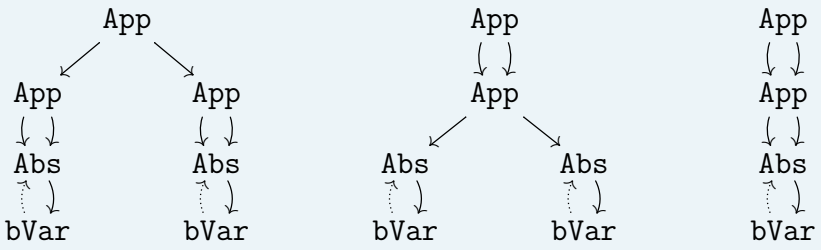
Conference paper. This part covers the results published in the article (Condoluci, Accattoli, and Sacerdoti Coen, 2019), whose full proofs can be found in the accompanying technical report (Condoluci, Accattoli, and Sacerdoti Coen, 2019).

Chapter 12

The Theory of Sharing Equality

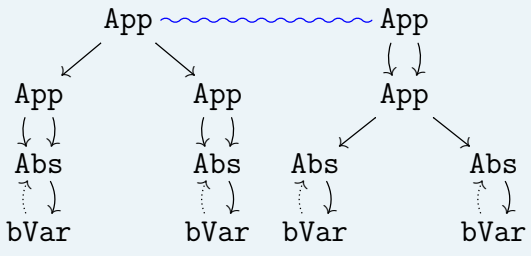
In this chapter we study the theory of sharing equality, *i.e.* in what sense different λ-graphs can represent the same λ-term.

A good first intuition is that two λ-graphs describe the same unfolded λ-term if they present the same structural paths, just collapsed in a different way. Consider for example the three equivalent λ-graphs below:



Sharing equivalent λ-graphs / **Figure 12.1**

The most natural way of checking sharing equality then seems to test the given graphs for bisimilarity: the expected outcome is that two nodes are bisimilar if and only if they have the same readback to λ-terms. For example, we may query whether the first two λ-graphs of fig. 12.1 are sharing equivalent by asking whether their roots nodes are bisimilar. We denote queries visually by a curly blue wave:



A query / **Figure 12.2**

The simplest case is when there are only two roots, say n and m, and the query contains only $\mathsf{n}\ \mathcal{Q}\ \mathsf{m}$. However, since λ-graphs may have multiple root nodes it is not necessary to restrict the query to only two roots. Hence we give a more general definition of *queries* that may relate any number of roots of not necessarily disjoint or even distinct λ-graphs:

Definition 12.1 \ Query $\mathcal{Q}$

Let G be a pre-λ-graph. We call *query* (over G) any binary relation $\mathcal{Q}$ over the root nodes[1] of G.

Traditionally two nodes are said to be *bisimilar* if there exists a bisimulation that relates them, but in our more general setting of a query $\mathcal{Q}$ the requirement is the existence of a bisimulation that contains $\mathcal{Q}$.

To define formally what a bisimulation is, we need two ingredients. First of all, bisimulations can only relate nodes that are *homogeneous*, *i.e.* nodes with the same label: this is justified from the perspective of λ-terms, since nodes with different labels clearly cannot have the same readback.

Definition 12.2 \ Homogeneous nodes (Paterson and Wegman, 1978)

Let n, m be nodes of a labeled graph G. We say that n and m are *homogeneous* (in G) if they have the same label.

In the rest of the chapter we will denote by $\mathcal{R}$ a generic binary relation over the nodes of a λ-graph G. We call $\mathcal{R}$ *homogeneous* if it only relates pairs of homogeneous nodes, *i.e.* $\mathsf{n}\ \mathcal{R}\ \mathsf{m}$ implies that n and m are homogeneous in G.

The other requirement for a bisimulation is to be closed under the expected structural rules over λ-graphs, which we provide in fig. 12.3.

$$\frac{}{\mathsf{n}\ \mathcal{R}\ \mathsf{n}}\ \bullet \qquad \frac{\mathsf{n}\ \mathcal{R}\ \mathsf{m}}{\mathsf{m}\ \mathcal{R}\ \mathsf{n}}\ \leftrightarrow \qquad \frac{\mathsf{n}\ \mathcal{R}\ \mathsf{m} \qquad \mathsf{m}\ \mathcal{R}\ \mathsf{p}}{\mathsf{n}\ \mathcal{R}\ \mathsf{p}}\ \twoheadrightarrow$$

$$\frac{\mathtt{App}(\mathsf{n}_1, \mathsf{n}_2)\ \mathcal{R}\ \mathtt{App}(\mathsf{m}_1, \mathsf{m}_2)}{\mathsf{n}_1\ \mathcal{R}\ \mathsf{m}_1}\ \swarrow \qquad \frac{\mathtt{App}(\mathsf{n}_1, \mathsf{n}_2)\ \mathcal{R}\ \mathtt{App}(\mathsf{m}_1, \mathsf{m}_2)}{\mathsf{n}_2\ \mathcal{R}\ \mathsf{m}_2}\ \searrow$$

$$\frac{\mathtt{Abs}(\mathsf{n})\ \mathcal{R}\ \mathtt{Abs}(\mathsf{m})}{\mathsf{n}\ \mathcal{R}\ \mathsf{m}}\ \downarrow \qquad \frac{\mathtt{bVar}(\mathsf{n})\ \mathcal{R}\ \mathtt{bVar}(\mathsf{m})}{\mathsf{n}\ \mathcal{R}\ \mathsf{m}}\ \circlearrowleft$$

Sharing equivalence rules / **Figure 12.3**

We group the rules in fig. 12.3 in the following way:

- *Equivalence rules*: rules $\bullet$ $\leftrightarrow$ $\twoheadrightarrow$ are the usual rules that characterize equivalence relations, respectively reflexivity, symmetry, and transitivity.

[1] One may wonder why restricting queries to root nodes only: see the discussion on page 176.

- *Bisimulation rules*:
  - *Propagation rules:* rules ↙ ↓ ↘ are downward propagation rules on the λ-graph. The rules ↙ and ↘ state that if two application nodes are related, then also their corresponding left and right children should be related. The rule ↓ states that if two abstraction nodes are related, then also their bodies should be related.
  - *Scoping rule:* the rule ↻ states that if two bound variable nodes are related, then also their binders should be related.

We are now ready to define bisimulations formally:

Definition 12.3 \ Bisimulation

Let G be a λ-graph, and $\mathcal{R}$ be a binary relation over the nodes of G. $\mathcal{R}$ is a *bisimulation* (over G) if and only if it is homogeneous and closed under the rules ↙ ↓ ↘ ↻ of fig. 12.3.

However, the existence of a bisimulation is not sufficient to capture sharing equality: free variable nodes must be handled separately, because we must take into account their attached name. We thus introduce the new concept of *open* relations, relations that do not relate free variable nodes with different names:

Definition 12.4 \ Open

Let $\mathcal{R}$ be a binary relation over the nodes of a λ-graph G. We call $\mathcal{R}$ *open* when $\texttt{fVar}(x)\ \mathcal{R}\ \texttt{fVar}(y)$ implies $x =_\mathcal{V} y$.

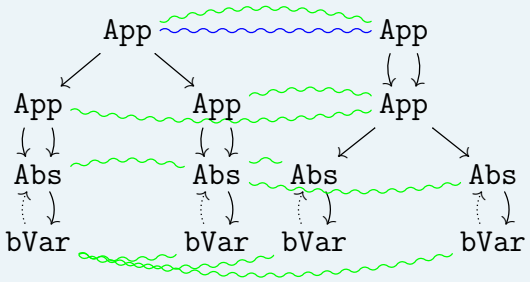
As we are going to show, open bisimulations characterize exactly sharing equality. However our sharing equality algorithm in chapter 14 computes what we call sharing equivalences, which are open bisimulations and also equivalence relations.

Definition 12.5 \ Sharing equivalence

Let $\cong$ be an equivalence relation over the nodes of a λ-graph G. We say that $\cong$ is a *sharing equivalence* if and only if it is an open bisimulation.

Let us come back to the query $\mathcal{Q}$ from fig. 12.2: we provide below a sharing equivalence containing $\mathcal{Q}$. Sharing equivalences are equivalence relations, but in the figure we use an economical representation with green (horizontal) waves, connecting nodes in the same equivalence class by a green path, and omitting reflexive/transitive waves.

Sharing equivalence (cf. fig. 12.2)



Let us outline the development of the rest of this chapter. We delay until section 12.3 the proof that sharing equivalences correctly characterize the equality of the read back λ-terms. We first focus on characterizing the smallest sharing equivalence containing a given query, since our algorithm in chapter 14 computes exactly that smallest sharing equivalence. In order to do so, we define two different closures of queries over a λ-graph: *propagations* in section 12.1, and *spreadings* in section 12.2. For both closures we prove *universality*, *i.e.* the fact that they produce the desired relations whenever possible: the propagation of a query is the universal (open) bisimulation containing that query, and similarly the spreading of a query is the universal sharing equivalence containing that query. We conclude the chapter with the *sharing equality theorem* (theorem 12.29).

12.1 Propagation $\Downarrow$

In this section we study the problem of finding the smallest open bisimulation containing a given query $\mathcal{Q}$. To this aim, we first define the propagation $\mathcal{Q}^{\Downarrow}$ of $\mathcal{Q}$, which amounts to the closure of $\mathcal{Q}$ under the structural rules of the underlying pre-λ-graph. Since propagation does not distinguish bound variable nodes (*i.e.* the backward $\circlearrowleft$ edges) it does not look like a good candidate for a bisimulation, but this is not actually the case. In fact, the main and surprising result of this section is the universality of $\mathcal{Q}^{\Downarrow}$, theorem 12.16: $\mathcal{Q}^{\Downarrow}$ is actually the smallest (open) bisimulation containing $\mathcal{Q}$, whenever any (open) bisimulation containing $\mathcal{Q}$ exists at all.

We first define *propagated relations*, that are relations closed under the structural rules of pre-λ-graphs:

Definition 12.6 \ Propagated relation

Let $\mathcal{R}$ be a binary relation over the nodes of a pre-λ-graph G. We say that $\mathcal{R}$ is *propagated* (over G) if and only if it is closed under the rules $\swarrow$, $\downarrow$, $\searrow$ of fig. 12.3.

We also define a weaker form of bisimulations, which we call pre-bisimulations. They are to pre-λ-graphs what bisimulations are to λ-graphs:

Definition 12.7 \ Pre-bisimulation

Let G be a pre-λ-graph, and $\mathcal{B}$ be a binary relation over the nodes of G. We say that $\mathcal{B}$ is a *pre-bisimulation* if and only if it is homogeneous and propagated over G.

In the rest of this dissertation we denote by $\mathcal{B}$ a generic (pre-)bisimulation.

We start with two auxiliary propositions that connect propagated relations and paths of λ-graphs. The first one shows that propagated relations are closed also under the iteration of propagation rules, *i.e.* under traces:

Proposition 12.8 \ Trace propagation

Let $\mathcal{R}$ be a propagated relation over a pre-λ-graph G. Let n, m be nodes of G such that $\mathsf{n}\ \mathcal{R}\ \mathsf{m}$, and let τ be a trace such that $\tau\colon \mathsf{n} \rightsquigarrow \mathsf{n}'$ and $\tau\colon \mathsf{m} \rightsquigarrow \mathsf{m}'$. Then $\mathsf{n}'\ \mathcal{R}\ \mathsf{m}'$.

Proof. We prove that $\mathsf{n}'\ \mathcal{R}\ \mathsf{m}'$ by structural induction on the trace τ:

- Case ϵ. Clearly $\mathsf{n}' = \mathsf{n}$ and $\mathsf{m}' = \mathsf{m}$, and we use the hypothesis $\mathsf{n}\ \mathcal{R}\ \mathsf{m}$.
- Case $(\tau \cdot \swarrow)$. Assume $(\tau \cdot \swarrow)\colon \mathsf{n} \rightsquigarrow \mathsf{n}'$ and $(\tau \cdot \swarrow)\colon \mathsf{m} \rightsquigarrow \mathsf{m}'$. By inversion, there exist $\mathsf{n}'', \mathsf{m}''$ such that $\tau\colon \mathsf{n} \rightsquigarrow \mathtt{App}(\mathsf{n}', \mathsf{n}'')$ and $\tau\colon \mathsf{m} \rightsquigarrow \mathtt{App}(\mathsf{m}', \mathsf{m}'')$. By *i.h.* $\mathtt{App}(\mathsf{n}', \mathsf{n}'')\ \mathcal{R}\ \mathtt{App}(\mathsf{m}', \mathsf{m}'')$, which implies $\mathsf{n}'\ \mathcal{R}\ \mathsf{m}'$ by ↙.
- Case $(\tau \cdot \searrow)$. Symmetric to the case above.
- *Case* $(\tau \cdot \downarrow)$. Assume $(\tau \cdot \downarrow)\colon \mathsf{n} \rightsquigarrow \mathsf{n}'$ and $(\tau \cdot \downarrow)\colon \mathsf{m} \rightsquigarrow \mathsf{m}'$. By inversion, $\tau\colon \mathsf{n} \rightsquigarrow \mathtt{Abs}(\mathsf{n}')$ and $\tau\colon \mathsf{m} \rightsquigarrow \mathtt{Abs}(\mathsf{m}')$. By *i.h.* $\mathtt{Abs}(\mathsf{n}')\ \mathcal{R}\ \mathtt{Abs}(\mathsf{m}')$, which implies $\mathsf{n}'\ \mathcal{R}\ \mathsf{m}'$ by ↓. ∎

The second auxiliary proposition shows that whenever two nodes are related by a pre-bisimulation, then they are start nodes of paths with the same traces:

Proposition 12.9 \ Trace equivalence

Let $\mathcal{R}$ be a pre-bisimulation over a pre-λ-graph G. Let n, m be nodes of G such that $\mathsf{n}\ \mathcal{R}\ \mathsf{m}$. Then for every trace τ, $\tau\colon \mathsf{n}$ if and only if $\tau\colon \mathsf{m}$.

Proof. Assume that $\mathsf{n}\ \mathcal{R}\ \mathsf{m}$ holds. We proceed by structural induction on τ:

- Case ϵ. Clearly $\epsilon\colon \mathsf{n}$ and $\epsilon\colon \mathsf{m}$.
- Case $(\tau \cdot d)$. We assume without loss of generality that $(\tau \cdot d)\colon \mathsf{n}$, and prove that $(\tau \cdot d)\colon \mathsf{m}$.

  By inversion, $\tau\colon \mathsf{n} \rightsquigarrow \mathsf{n}'$ for some n', and by *i.h.* $\tau\colon \mathsf{m} \rightsquigarrow \mathsf{m}'$ for some m'. From proposition 12.8 we obtain $\mathsf{n}'\ \mathcal{R}\ \mathsf{m}'$, where n' and m' are homogenous because $\mathcal{R}$ is a pre-bisimulation by hypothesis. We proceed by cases on d:

  - Case $d = \downarrow$. Then $\mathsf{m}' = \mathtt{Abs}(\mathsf{m}'')$ for some m''. Clearly $(\tau \cdot \downarrow)\colon \mathsf{m} \rightsquigarrow \mathsf{m}''$.

- Cases $d = \swarrow$ or $d = \searrow$. Then $\mathsf{m}' = \mathtt{App}(\mathsf{m}'_1, \mathsf{m}'_2,)$ for some $\mathsf{m}'_1, \mathsf{m}'_2$. Clearly $(\tau \cdot \swarrow) \colon \mathsf{m} \rightsquigarrow \mathsf{m}'_1$ and $(\tau \cdot \searrow) \colon \mathsf{m} \rightsquigarrow \mathsf{m}'_2$.

We now turn to the definition of propagation. We define the propagation $\mathcal{R}^{\Downarrow}$ of a relation $\mathcal{R}$ as the smallest propagated relation containing $\mathcal{R}$:

Definition 12.10 \ Propagation $\mathcal{R}^{\Downarrow}$

Let $\mathcal{R}$ be a binary relation over the nodes of a pre-λ-graph G. The propagation $\mathcal{R}^{\Downarrow}$ of $\mathcal{R}$ is obtained by closing $\mathcal{R}$ under the rules ↙ ↓ ↘ of fig. 12.3.

The following remark states that $\mathcal{R}^{\Downarrow}$ is the smallest propagated relation containing $\mathcal{R}$:

Remark 12.11 \ Minimality of $\mathcal{R}^{\Downarrow}$

If $\mathcal{R} \subseteq \mathcal{R}'$ and $\mathcal{R}'$ is propagated, then $\mathcal{R}^{\Downarrow} \subseteq \mathcal{R}'$.

We can also rephrase the minimality of $\mathcal{R}^{\Downarrow}$ in terms of paths:

Proposition 12.12 \ Inversion

Let $\mathcal{R}$ be a binary relation over the nodes of a pre-λ-graph G, and n, m be nodes of G. $\mathsf{n}\ \mathcal{R}^{\Downarrow}\ \mathsf{m}$ if and only there exists a trace τ and nodes n', m' such that $\mathsf{n}'\ \mathcal{R}\ \mathsf{m}'$, $\tau \colon \mathsf{n}' \rightsquigarrow \mathsf{n}$, and $\tau \colon \mathsf{m}' \rightsquigarrow \mathsf{m}$.

Proof. We prove separately the two directions of the "if and only if":

(⇒) Assume that $\mathsf{n}\ \mathcal{R}^{\Downarrow}\ \mathsf{m}$. We proceed by induction on the inductive definition of the derivation of "$\mathsf{n}\ \mathcal{R}^{\Downarrow}\ \mathsf{m}$":

- Base case: $\mathsf{n}\ \mathcal{R}^{\Downarrow}\ \mathsf{m}$ because $\mathsf{n}\ \mathcal{R}\ \mathsf{m}$. Conclude by considering $\mathsf{n}' = \mathsf{n}, \mathsf{m} = \mathsf{m}'$, and $\tau' = \epsilon$.
- Case ↙: assume that $\mathsf{n}\ \mathcal{R}^{\Downarrow}\ \mathsf{m}$ because $\mathtt{App}(\mathsf{n}, \mathsf{n}'')\ \mathcal{R}^{\Downarrow}\ \mathtt{App}(\mathsf{m}, \mathsf{m}'')$ for some $\mathsf{n}'', \mathsf{m}''$. By *i.h.* there exists τ such that $\tau \colon \mathsf{n}' \rightsquigarrow \mathtt{App}(\mathsf{n}, \mathsf{n}'')$ and $\tau \colon \mathsf{m}' \rightsquigarrow \mathtt{App}(\mathsf{m}, \mathsf{m}'')$. Clearly $(\tau \cdot \swarrow) \colon \mathsf{n}' \rightsquigarrow \mathsf{n}$ and $(\tau \cdot \swarrow) \colon \mathsf{m}' \rightsquigarrow \mathsf{m}$.
- Case ↘. Symmetric to the case above.
- Case ↓: assume that $\mathsf{n}\ \mathcal{R}^{\Downarrow}\ \mathsf{m}$ because $\mathtt{Abs}(\mathsf{n})\ \mathcal{R}^{\Downarrow}\ \mathtt{Abs}(\mathsf{m})$. By *i.h.* there exists τ such that $\tau \colon \mathsf{n}' \rightsquigarrow \mathtt{Abs}(\mathsf{n})$ and $\tau \colon \mathsf{m}' \rightsquigarrow \mathtt{Abs}(\mathsf{m})$. Clearly $(\tau \cdot \downarrow) \colon \mathsf{n}' \rightsquigarrow \mathsf{n}$, and $(\tau \cdot \downarrow) \colon \mathsf{m}' \rightsquigarrow \mathsf{m}$.

(⇐) Note that $\mathsf{n}\ \mathcal{R}\ \mathsf{m}$ implies $\mathsf{n}\ \mathcal{R}^{\Downarrow}\ \mathsf{m}$, and conclude by proposition 12.8.

The following proposition is a fundamental property of bisimulations, required to prove many upcoming results: it states that (pre-)bisimulations cannot relate both a node and

their children. This is a weaker form of the fact that if we quotient a pre-λ-graph over a pre-bisimulation, the result is still a pre-λ-graph, and in particular it implies that there cannot be cycles in the quotient graph.

Proposition 12.13 \ No triangles

Let G be a finite and acyclic pre-λ-graph, and let $\mathcal{B}$ be a pre-bisimulation over G. Let $\mathsf{n}, \mathsf{m}, \mathsf{m}'$ be nodes of G, and τ a non-empty trace. If $\mathsf{n}\ \mathcal{B}\ \mathsf{m}$ and $\tau\colon \mathsf{m} \rightsquigarrow \mathsf{m}'$, then $\mathsf{n}\ \mathcal{B}\ \mathsf{m}'$ can not hold.

Proof. Assume that $\mathsf{n}\ \mathcal{B}\ \mathsf{m}'$, and obtain a contradiction. Intuitively, we are going to show that the trace τ can be iterated arbitrarily many times starting from m, contradicting the hypothesis that the input pre-λ-graph is finite and acyclic.
The contradiction will follow by iterating the following construction:

- From the hypotheses that $\mathsf{n}\ \mathcal{B}\ \mathsf{m}$ and $\tau\colon \mathsf{m} \rightsquigarrow \mathsf{m}'$ it follows from proposition 12.9 that $\tau\colon \mathsf{n} \rightsquigarrow \mathsf{n}'$ for some n'. From the hypothesis that $\mathsf{n}\ \mathcal{B}\ \mathsf{m}$ together with $\tau\colon \mathsf{n} \rightsquigarrow \mathsf{n}'$ and $\tau\colon \mathsf{m} \rightsquigarrow \mathsf{m}'$, it follows that $\mathsf{n}'\ \mathcal{B}\ \mathsf{m}'$ by proposition 12.8.
- From the hypothesis that $\mathsf{n}\ \mathcal{B}\ \mathsf{m}'$ and $\tau\colon \mathsf{n} \rightsquigarrow \mathsf{n}'$ it follows from proposition 12.9 that $\tau\colon \mathsf{m}' \rightsquigarrow \mathsf{m}''$ for some m''. From the hypothesis that $\mathsf{n}\ \mathcal{B}\ \mathsf{m}'$ together with $\tau\colon \mathsf{n} \rightsquigarrow \mathsf{n}'$ and $\tau\colon \mathsf{m}' \rightsquigarrow \mathsf{m}''$, it follows that $\mathsf{n}'\ \mathcal{B}\ \mathsf{m}''$ by proposition 12.8.

The construction above can be repeated by using the new hypotheses $\mathsf{n}'\ \mathcal{B}\ \mathsf{m}'$, $\tau\colon \mathsf{m}' \rightsquigarrow \mathsf{m}''$, and $\mathsf{n}'\ \mathcal{B}\ \mathsf{m}''$. Iterating the construction arbitrarily many times yields a sequence of nodes $\mathsf{m}, \mathsf{m}', \mathsf{m}'', \ldots$ such that $\tau\colon \mathsf{m} \rightsquigarrow \mathsf{m}'$, $\tau\colon \mathsf{m}' \rightsquigarrow \mathsf{m}''$, …
This contradicts the hypothesis that G is finite and acyclic. ∎

We now turn to proving the universality of $\mathcal{Q}^{\Downarrow}$ (upcoming theorem 12.16), but we first need two auxiliary propositions. The first one is a simple remark about cutting a subpath in correspondence of a crossed node.

Remark 12.14 \ Cutting a path

Let $\mathsf{n}, \mathsf{m}, \mathsf{p}$ be nodes of a pre-λ-graph G, and τ a trace such that $\tau\colon \mathsf{n} \rightsquigarrow \mathsf{m}$ crosses p. Then there exists a trace τ' such that $\tau'\colon \mathsf{p} \rightsquigarrow \mathsf{m}$.

The following proposition is the fundamental property to prove universality. It is the inductive variant needed in the proof of theorem 12.16, showing how it is not necessary to consider binding edges (↻) when looking for the smallest bisimulation. It basically states the following: take a bisimulation $\mathcal{B}$ containing a relation $\mathcal{R}$, and two root nodes r, r' related by these relations. By walking down the graph, starting from the two nodes and using the same trace, one may cross various abstraction nodes. The proposition states that $\mathcal{B}$ and $\mathcal{R}^{\Downarrow}$ relate exactly the same pairs of abstraction nodes encountered.

Proposition 12.15 \ Propagation agrees with bisimulations

Let G be a finite and acyclic pre-λ-graph, let $\mathcal{B}$ be a pre-bisimulation over G, and let $\mathcal{R}$ a relation such that $\mathcal{R} \subseteq \mathcal{B}$. Let n, m be nodes of G such that $\mathsf{n}\ \mathcal{R}\ \mathsf{m}$, and τ a trace such that $\tau : \mathsf{n}$ crosses l and $\tau : \mathsf{m}$ crosses l'. Then $\mathsf{l}\ \mathcal{B}\ \mathsf{l}'$ implies $\mathsf{l}\ \mathcal{R}^{\Downarrow}\ \mathsf{l}'$.

Proof. Assume $\mathsf{l}\ \mathcal{B}\ \mathsf{l}'$. We proceed by structural induction over the derivations of "$\tau : \mathsf{n}$ crosses l" and "$\tau : \mathsf{m}$ crosses l'":

- Case $\tau : \mathsf{n} \rightsquigarrow \mathsf{l}$ and $\tau : \mathsf{m} \rightsquigarrow \mathsf{l}'$. By proposition 12.12 it follows that $\mathsf{l}\ \mathcal{R}^{\Downarrow}\ \mathsf{l}'$, and we can conclude.
- Case $\tau : \mathsf{n} \rightsquigarrow \mathsf{l}$ and $\tau : \mathsf{m} \rightsquigarrow \mathsf{m}' \neq \mathsf{l}'$: this case is not possible, *i.e.* we derive a contradiction. Since $\tau : \mathsf{m} \rightsquigarrow \mathsf{m}'$ crosses l', we obtain by remark 12.14 a trace τ' such that $\tau' : \mathsf{l}' \rightsquigarrow \mathsf{m}'$, that is non-empty because $\mathsf{l}' \neq \mathsf{m}'$. Note that by proposition 12.12 it holds that $\mathsf{l}\ \mathcal{R}^{\Downarrow}\ \mathsf{m}'$, that implies $\mathsf{l}\ \mathcal{B}\ \mathsf{m}'$ by remark 12.11. Obtain a contradiction by proposition 12.13, using $\mathsf{l}\ \mathcal{B}\ \mathsf{l}'$, $\tau' : \mathsf{l}' \rightsquigarrow \mathsf{m}'$, and $\mathsf{l}\ \mathcal{B}\ \mathsf{m}'$.
- Case $\tau : \mathsf{n} \rightsquigarrow \mathsf{n}' \neq \mathsf{l}$ and $\tau : \mathsf{m} \rightsquigarrow \mathsf{l}'$: symmetric to the case above.
- Case $\tau = (\tau' \cdot d), (\tau' \cdot d) : \mathsf{n} \rightsquigarrow \mathsf{n}' \neq \mathsf{l}$ and $(\tau' \cdot d) : \mathsf{m} \rightsquigarrow \mathsf{m}' \neq \mathsf{l}'$. It holds that $\tau' : \mathsf{n}$ crosses l and $\tau' : \mathsf{m}$ crosses l', and we just use the *i.h.* to conclude. ∎

We can finally prove the universality of $\mathcal{Q}^{\Downarrow}$:

Theorem 12.16 \ Universality of $\mathcal{Q}^{\Downarrow}$

Let $\mathcal{Q}$ be a query over a λ-graph G. $\mathcal{Q}^{\Downarrow}$ is an open bisimulation if and only if there exists an open bisimulation containing $\mathcal{Q}$.

Proof. Clearly if $\mathcal{Q}^{\Downarrow}$ is an open bisimulation then $\mathcal{Q}^{\Downarrow}$ is the required open bisimulation containing $\mathcal{Q}$. As for the other direction, let $\mathcal{B}$ be an open bisimulation containing $\mathcal{Q}$. By definition, $\mathcal{Q}^{\Downarrow}$ is closed under the rules ↙, ↓, ↘ of fig. 12.3. In order to prove that $\mathcal{Q}^{\Downarrow}$ is an open bisimulation, it suffices to show that $\mathcal{Q}^{\Downarrow}$ is also open, homogeneous and closed under ↻.

- *Open:* since $\mathcal{Q}^{\Downarrow}$ is contained in $\mathcal{B}$ by remark 12.11, $\mathtt{fVar}(x)\ \mathcal{Q}^{\Downarrow}\ \mathtt{fVar}(x')$ implies $\mathtt{fVar}(x)\ \mathcal{B}\ \mathtt{fVar}(x')$, and since $\mathcal{B}$ is open we obtain $x = x'$.
- *Homogeneous:* $\mathcal{Q}^{\Downarrow}$ is homogeneous because $\mathcal{Q}^{\Downarrow}$ is contained in $\mathcal{B}$ (by remark 12.11), and $\mathcal{B}$ is homogeneous.
- *Closed under* ↻: we need to show that $\mathsf{l}\ \mathcal{Q}^{\Downarrow}\ \mathsf{l}'$ whenever $\mathtt{bVar}(\mathsf{l})\ \mathcal{Q}^{\Downarrow}\ \mathtt{bVar}(\mathsf{l}')$. Since $\mathcal{Q}^{\Downarrow}$ is contained in $\mathcal{B}$ by remark 12.11, $\mathtt{bVar}(\mathsf{l})\ \mathcal{Q}^{\Downarrow}\ \mathtt{bVar}(\mathsf{l}')$ implies $\mathtt{bVar}(\mathsf{l})\ \mathcal{B}\ \mathtt{bVar}(\mathsf{l}')$, and since $\mathcal{B}$ is closed under ↻ we obtain $\mathsf{l}\ \mathcal{B}\ \mathsf{l}'$.

  From $\mathtt{bVar}(\mathsf{l})\ \mathcal{Q}^{\Downarrow}\ \mathtt{bVar}(\mathsf{l}')$ and proposition 12.12, there exists a trace τ and nodes $\mathsf{n}\ \mathcal{Q}\ \mathsf{m}$ such that $\tau : \mathsf{n} \rightsquigarrow \mathtt{bVar}(\mathsf{l})$ and $\tau : \mathsf{m} \rightsquigarrow \mathtt{bVar}(\mathsf{l}')$. Since $\mathcal{Q}$ is a query,

$\tau : \mathsf{n} \rightsquigarrow \mathtt{bVar}(\mathsf{l})$ crosses l and $\tau : \mathsf{m} \rightsquigarrow \mathtt{bVar}(\mathsf{l}')$ crosses l' by domination. We conclude with $\mathsf{l}\ \mathcal{Q}^{\Downarrow}\ \mathsf{l}'$ by proposition 12.15. ∎

12.2 Spreading

In this section, we lift the results of the previous section to relations that are also equivalences. We define the spreading $\mathcal{Q}^{\#}$ of a query $\mathcal{Q}$, which is similar to the propagation $\mathcal{Q}^{\Downarrow}$ but also an equivalence relation. The main result is the universality of $\mathcal{Q}^{\#}$, theorem 12.26: $\mathcal{Q}^{\#}$ is the smallest sharing equivalence containing $\mathcal{Q}$, whenever any sharing equivalence containing $\mathcal{Q}$ exists at all.

Equivalence relations. First of all we introduce the *equivalence closure* of a relation:[2]

Definition 12.17 \ Equivalence closure

Let $\mathcal{R}$ be a binary relation over the nodes of a λ-graph G. We denote with $\mathcal{R}^*$ the reflexitive, symmetric, and transitive (equivalence) closure of $\mathcal{R}$, *i.e.* the relation obtained by closing $\mathcal{R}$ under the rules ● ↔ ↠ of fig. 12.3.

We can now lift the notions introduced in the previous section to the equivalence closure. The equivalence closure preserves homogeneous relations:

Proposition 12.18 \ Equivalence preserves homogeneity

Let $\mathcal{R}$ be a binary relation over the nodes of a λ-graph G. $\mathcal{R}$ is homogeneous if and only if $\mathcal{R}^*$ is homogeneous.

Proof. If $\mathcal{R}^*$ is homogeneous, then also $\mathcal{R}$ is homogeneous because $\mathcal{R} \subseteq \mathcal{R}^*$. For the other direction, it suffices to note that the equivalence rules ● ↔ ↠ preserve homogeneity. ∎

And similarly open relations:

Proposition 12.19 \ Equivalence preserves openness

Let $\mathcal{R}$ be an homogeneous relation over the nodes of a λ-graph. $\mathcal{R}$ is open if and only if $\mathcal{R}^*$ is open.

Proof. If $\mathcal{R}^*$ is open, then also $\mathcal{R}$ is open because $\mathcal{R} \subseteq \mathcal{R}^*$. As for the other implication, let us assume that $\mathcal{R}$ is open, and that $\mathsf{n}\ \mathcal{R}^*\ \mathsf{m}$ for some free variable nodes n, m. We need to show that n and m carry the same name. We proceed by induction on the

[2] Note that in this part we denote the equivalence closure with $\bullet^*$; the same symbol is used in other places to denote the Kleene closure, which is not necessarily symmetric.

derivation of $\mathsf{n}\ \mathcal{R}^*\ \mathsf{m}$. The base case and the cases ● ↔ are easy; let us discuss the case ↠. Assume that $\mathsf{n}\ \mathcal{R}^*\ \mathsf{m}$ because $\mathsf{n}\ \mathcal{R}^*\ \mathsf{p}$ and $\mathsf{p}\ \mathcal{R}^*\ \mathsf{m}$ for some node p. Since $\mathcal{R}$ is homogeneous, by proposition 12.18 $\mathcal{R}^*$ is homogeneous too, and therefore p is a free variable node as well. By *i.h.* we obtain $\mathsf{n} = \mathsf{p}$ and $\mathsf{p} = \mathsf{m}$, and we conclude with $\mathsf{n} = \mathsf{m}$. ∎

Preservation under equivalence rules of closure under structural rules requires instead homogeneity:

Proposition 12.20 \ Closure preserved by equivalence closure

Let $\mathcal{R}$ be an homogeneous relation, and $r \in \{↙, ↓, ↘, ↻\}$. If $\mathcal{R}$ is closed under r, then $\mathcal{R}^*$ is closed under r.

Proof. We only consider the case when r is ↻; the proofs for the other rules are similar. Assume that $\mathcal{R}$ is closed under r, and let $\mathtt{bVar}(\mathsf{l})\ \mathcal{R}^*\ \mathtt{bVar}(\mathsf{l}')$: we need to show that $\mathsf{l}\ \mathcal{R}^*\ \mathsf{l}'$. We proceed by induction on the inductive definition of $\mathcal{R}^*$:

- Base case: $\mathtt{bVar}(\mathsf{l})\ \mathcal{R}^*\ \mathtt{bVar}(\mathsf{l}')$ because $\mathtt{bVar}(\mathsf{l})\ \mathcal{R}\ \mathtt{bVar}(\mathsf{l}')$. By the hypothesis that $\mathcal{R}$ is closed under ↻ it follows that $\mathsf{l}\ \mathcal{R}\ \mathsf{l}'$, which implies $\mathsf{l}\ \mathcal{R}^*\ \mathsf{l}'$.
- Case ●: $\mathtt{bVar}(\mathsf{l})\ \mathcal{R}^*\ \mathtt{bVar}(\mathsf{l}')$ because $\mathtt{bVar}(\mathsf{l}) = \mathtt{bVar}(\mathsf{l}')$. Then $\mathsf{l} = \mathsf{l}'$, and we conclude with $\mathsf{l}\ \mathcal{R}^*\ \mathsf{l}'$ by the rule ●.
- Case ↔: $\mathtt{bVar}(\mathsf{l})\ \mathcal{R}^*\ \mathtt{bVar}(\mathsf{l}')$ because $\mathtt{bVar}(\mathsf{l}')\ \mathcal{R}^*\ \mathtt{bVar}(\mathsf{l})$. By *i.h.* $\mathsf{l}'\ \mathcal{R}^*\ \mathsf{l}$, and we obtain $\mathsf{l}\ \mathcal{R}^*\ \mathsf{l}'$ by the rule ↔.
- Case ↠: $\mathtt{bVar}(\mathsf{l})\ \mathcal{R}^*\ \mathtt{bVar}(\mathsf{l}')$ because $\mathtt{bVar}(\mathsf{l})\ \mathcal{R}^*\ \mathsf{n}$ and $\mathsf{n}\ \mathcal{R}^*\ \mathtt{bVar}(\mathsf{l}')$ for some node n. Since $\mathcal{R}$ is homogeneous, by proposition 12.18 $\mathcal{R}^*$ is homogeneous too, and therefore $\mathsf{n} = \mathtt{bVar}(\mathsf{l}'')$ for some l''. By *i.h.* we obtain $\mathsf{l}\ \mathcal{R}^*\ \mathsf{l}''$ and $\mathsf{l}''\ \mathcal{R}^*\ \mathsf{l}'$, and we conclude with $\mathsf{l}\ \mathcal{R}^*\ \mathsf{l}'$ by the rule ↠. ∎

Spreading. We now turn to introduce the spreading $\mathcal{R}^{\#}$ of a relation $\mathcal{R}$:

Definition 12.21 \ Spreading $\mathcal{R}^{\#}$

Let $\mathcal{R}$ be a binary relation over the nodes of a λ-graph. $\mathcal{R}^{\#}$ is obtained by closing $\mathcal{R}$ under the rules ● ↔ ↠ ↙ ↓ ↘ of fig. 12.3.

Clearly $(\mathcal{R}^{\Downarrow})^* \subseteq (\mathcal{R}^{\#})$ for every relation $\mathcal{R}$. The converse may in principle not hold, as the equivalence rules may not commute with the rules of the spreading: for example, the equivalence closure of a spreaded relation may not be spreaded in general. But surprisingly

this is not the case, under the hypothesis that $\mathcal{R}^{\Downarrow}$ is homogeneous:

Proposition 12.22 \ Spreading *vs* propagation

Let $\mathcal{R}$ be a binary relation over the nodes of a λ-graph G. If $\mathcal{R}^{\Downarrow}$ is homogeneous, then $\mathcal{R}^{\#} = (\mathcal{R}^{\Downarrow})^*$.

Proof. Note that $(\mathcal{R}^{\Downarrow})^* \subseteq (\mathcal{R}^{\#})$, and that $(\mathcal{R}^{\Downarrow})^*$ is closed under the rules ● ↔ ↠ by the definition of $\bullet^*$. It remains to show that $(\mathcal{R}^{\Downarrow})^*$ is also closed under the rules ↙ ↓ ↘ which follows from proposition 12.20. □

We lift propositions 12.18 and 12.19 to $\mathcal{R}^{\#}$:

Proposition 12.23 \ $\mathcal{R}^{\Downarrow}$ preserves homogeneity

Let $\mathcal{R}$ be a binary relation over the nodes of a λ-graph. $\mathcal{R}^{\#}$ is homogeneous if and only if $\mathcal{R}^{\Downarrow}$ is homogeneous.

Proof. Clearly if $\mathcal{R}^{\#}$ is homogeneous, then also $\mathcal{R}^{\Downarrow}$ is homogeneous because $(\mathcal{R}^{\Downarrow}) \subseteq (\mathcal{R}^{\#})$. For the other direction, assume that $\mathcal{R}^{\Downarrow}$ is homogeneous: by proposition 12.18 it follows that $(\mathcal{R}^{\Downarrow})^*$ is homogeneous, and we can conclude by proposition 12.22. □

Proposition 12.24 \ $\mathcal{R}^{\Downarrow}$ preserves openness

Let $\mathcal{Q}$ be a query such that $\mathcal{Q}^{\Downarrow}$ is homogeneous. $\mathcal{Q}^{\Downarrow}$ is open if and only if $\mathcal{Q}^{\#}$ is open.

Proof. Clearly if $\mathcal{R}^{\#}$ is open, then also $\mathcal{R}^{\Downarrow}$ is open because $(\mathcal{R}^{\Downarrow}) \subseteq (\mathcal{R}^{\#})$. For the other direction, assume that $\mathcal{R}^{\Downarrow}$ is open and homogeneous: by proposition 12.18 it follows that $(\mathcal{R}^{\Downarrow})^*$ is homogeneous, and we can conclude by proposition 12.19. □

The following theorem shows that $\mathcal{Q}^{\Downarrow}$ and $\mathcal{Q}^{\#}$ are somewhat equivalent:

Theorem 12.25 \ $\mathcal{Q}^{\Downarrow}$ *vs* $\mathcal{Q}^{\#}$

Let $\mathcal{Q}$ be a query over a λ-graph G. $\mathcal{Q}^{\Downarrow}$ is an open bisimulation if and only if $\mathcal{Q}^{\#}$ is a sharing equivalence.

Proof. Let $\mathcal{Q}$ be a query over G. We prove separately the two directions of the double implication:

(⇒) Assume that $\mathcal{Q}^{\Downarrow}$ is an open bisimulation; in order to prove that $\mathcal{Q}^{\#}$ is a sharing equivalence, it suffices to show that $\mathcal{Q}^{\#}$ is open, homogeneous and closed under ↻. By proposition 12.22 we can discuss $(\mathcal{R}^{\Downarrow})^*$ in place of $\mathcal{R}^{\#}$. Then the requirements follow from proposition 12.20, proposition 12.23, and proposition 12.24.

(⇐) Assume that $\mathcal{Q}^{\#}$ is a sharing equivalence. Then it is also an open bisimulation, and

thus $\mathcal{Q}^{\Downarrow}$ is an open bisimulation by the universality of $\mathcal{Q}^{\Downarrow}$ (theorem 12.16). ∎

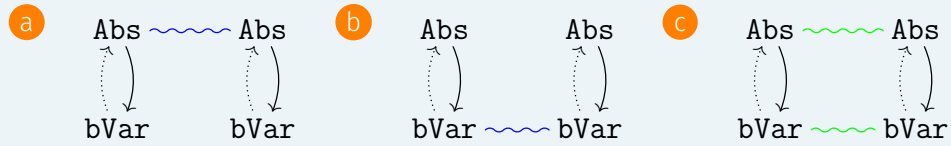
And finally the universality of $\mathcal{Q}^{\#}$ easily follows:

Theorem 12.26 \ Universality of $\mathcal{Q}^{\#}$

Let $\mathcal{Q}$ be a query over a λ-graph G. $\mathcal{Q}^{\#}$ is a sharing equivalence if and only if there exists a sharing equivalence containing $\mathcal{Q}$.

Proof. The implication from left to right is trivial. The other implication follows from theorem 12.16 and theorem 12.25. ∎

Queries and universality. The universality of $\mathcal{Q}^{\Downarrow}$ and $\mathcal{Q}^{\#}$ is a neat result, but the hypotheses may seem too restrictive: why requiring the relation $\mathcal{Q}$ to be a query, *i.e.* to only relate root nodes? It turns out that universality does not hold for the closure of generic relations. Consider for example the three relations shown in fig. 12.4 over the same λ-graph:



Various relations over a λ-graph. / **Figure 12.4**

Note that only the relation in fig. 12.4a is a query, but in all three cases there exists a sharing equivalence containing that relation; such a sharing equivalence is depicted in fig. 12.4c. However the spreading of fig. 12.4b is (the reflexive closure of) fig. 12.4b itself, which is different from fig. 12.4c and not a sharing equivalence.

12.3 Correctness

In this section we investigate the relationship between sharing equivalences over a λ-graph and the equality of the underlying λ-terms. The main result is the sharing equality theorem (theorem 12.29), showing that the existence of a sharing equivalence containing a given query correctly characterizes the equality of the λ-terms read back from that query.

First of all we prove the following auxiliary but fundamental proposition, which corresponds directly to proposition 12.15 for relations. This proposition states that a bisimulation can relate two abstraction nodes along a given trace if and only if their computed de Bruijn indices coincide.

Proposition 12.27 \ de Bruijn indices agree with bisimulations

Let $\mathcal{B}$ be a pre-bisimulation over a λ-graph G. Let n, m nodes of G such that $\mathsf{n}\ \mathcal{B}\ \mathsf{m}$, and τ a trace such that $\tau\colon \mathsf{n}$ crosses l and $\tau\colon \mathsf{m}$ crosses l'. Then $\mathsf{l}\ \mathcal{B}\ \mathsf{l}'$ if and only if $\mathtt{index}(\mathsf{l} \mid \tau\colon \mathsf{n}) = \mathtt{index}(\mathsf{l}' \mid \tau\colon \mathsf{m})$.

Proof. Let $\tau\colon \mathsf{n} \rightsquigarrow \mathsf{n}'$ and $\tau\colon \mathsf{m} \rightsquigarrow \mathsf{m}'$. Note that by proposition 12.8 it follows that $\mathsf{n}'\ \mathcal{B}\ \mathsf{m}'$, which implies that n' and m' have the same label since $\mathcal{B}$ is homogeneous. We proceed by structural induction on the trace τ, and following the cases of the definition of $\mathtt{index}(\mathsf{l} \mid \tau\colon \mathsf{n})$ and $\mathtt{index}(\mathsf{l}' \mid \tau\colon \mathsf{m})$.

1. Case $\mathtt{index}(\mathsf{l} \mid \tau\colon \mathsf{n} \rightsquigarrow \mathsf{l}) = 0 = \mathtt{index}(\mathsf{l}' \mid \tau\colon \mathsf{m} \rightsquigarrow \mathsf{l}')$. From proposition 12.8 we obtaine $\mathsf{l}\ \mathcal{B}^{\Downarrow}\ \mathsf{l}'$, and we can conclude.
2. Case $\mathtt{index}(\mathsf{l} \mid (\tau \cdot d)\colon \mathsf{n} \rightsquigarrow \mathsf{l}) = 0 \neq 1 + \mathtt{index}(\mathsf{l}' \mid \tau\colon \mathsf{m}) = \mathtt{index}(\mathsf{l}' \mid (\tau \cdot d)\colon \mathsf{m})$ with $\mathsf{l}' \neq \mathsf{m}'$. It holds that $\mathsf{l}\ \mathcal{B}^{\Downarrow}\ \mathsf{m}$, and from the fact that $\tau\colon \mathsf{m} \rightsquigarrow \mathsf{m}'$ crosses l' we obtain by remark 12.14 a trace τ' such that $(\tau' \cdot d)\colon \mathsf{l}' \rightsquigarrow \mathsf{m}'$. From proposition 12.13 (using $\mathsf{l}\ \mathcal{B}\ \mathsf{m}'$ and $(\tau' \cdot d)\colon \mathsf{l}' \rightsquigarrow \mathsf{m}'$) we obtain $\mathsf{l}\ \not{\mathcal{B}}\ \mathsf{l}'$ and conclude.
3. Case $\mathtt{index}(\mathsf{l} \mid (\tau \cdot d)\colon \mathsf{n}) = 1 + \mathtt{index}(\mathsf{l} \mid \tau\colon \mathsf{n}) \neq 0 = \mathtt{index}(\mathsf{l} \mid \tau\colon \mathsf{m} \rightsquigarrow \mathsf{l}')$ with $\mathsf{l} \neq \mathsf{n}'$. Symmetric to the case above.
4. Case $\mathtt{index}(\mathsf{l} \mid (\tau \cdot d)\colon \mathsf{n}) = 1 + \mathtt{index}(\mathsf{l}' \mid \tau\colon \mathsf{n})$ and $\mathtt{index}(\mathsf{l}' \mid (\tau \cdot d)\colon \mathsf{m}) = 1 + \mathtt{index}(\mathsf{l}' \mid \tau\colon \mathsf{m})$ with $\mathsf{l} \neq \mathsf{n}'$ and $\mathsf{l}' \neq \mathsf{m}'$. Use the *i.h.* to conclude.
5. Case $\mathtt{index}(\mathsf{l} \mid (\tau \cdot d)\colon \mathsf{n}) = \mathtt{index}(\mathsf{l} \mid \tau\colon \mathsf{n})$ and $\mathtt{index}(\mathsf{l}' \mid (\tau \cdot d)\colon \mathsf{m}) = \mathtt{index}(\mathsf{l}' \mid \tau\colon \mathsf{m})$: use the *i.h.* to conclude. ∎

To prove the sharing equality theorem, we first consider the propagation $\mathcal{Q}^{\Downarrow}$: in this way, we can temporarily set aside the equivalence rules and only consider the propagation rules of an open bisimulation, which correspond pretty much to the strctural rules of equality on (locally nameless) λ-terms. Later, we lift this result to any sharing equivalence by means of universality.

Proposition 12.28 \ Correctness of $\mathcal{Q}^{\Downarrow}$

Let $\mathcal{Q}$ be a query over a λ-graph G. $\mathcal{Q}^{\Downarrow}$ is an open bisimulation if and only if $[\![\mathsf{n}]\!] = [\![\mathsf{m}]\!]$ for every nodes n, m such that $\mathsf{n}\ \mathcal{Q}\ \mathsf{m}$.

Proof. First note that $\mathcal{Q}^{\Downarrow}$ is an open bisimulation if and only if $\mathcal{Q}^{\Downarrow}$ is open, homogeneous, and closed under  . We now prove separately the two implications of the statement of the theorem:

($\Leftarrow$) Assume $[\![\mathsf{n}]\!] = [\![\mathsf{m}]\!]$ for every $\mathsf{n}\ \mathcal{Q}\ \mathsf{m}$; we prove that $\mathcal{Q}^{\Downarrow}$ is open, homogeneous, and closed under  :

- *Open.* Let $\mathtt{fVar}(x)\ \mathcal{Q}^{\Downarrow}\ \mathtt{fVar}(x')$. By proposition 12.12, $\tau\colon \mathsf{r} \rightsquigarrow \mathtt{fVar}(x)$ and $\tau\colon \mathsf{r}' \rightsquigarrow \mathtt{fVar}(x)$ for some τ and $\mathsf{r}\ \mathcal{Q}\ \mathsf{r}'$. By proposition 6.11 $[\![\tau\colon \mathsf{r} \rightsquigarrow]\!] = [\![\tau\colon \mathsf{r}' \rightsquigarrow]\!]$, and by the definition of readback to λ-terms $x = x'$ and thus $x = x'$.
- *Homogeneous.* Let $\mathsf{n}\ \mathcal{Q}^{\Downarrow}\ \mathsf{m}$. By proposition 12.12, $\tau\colon \mathsf{n}' \rightsquigarrow \mathsf{n}$ and $\tau\colon \mathsf{m}' \rightsquigarrow \mathsf{m}$ for some τ and $\mathsf{n}'\ \mathcal{Q}\ \mathsf{m}'$. By proposition 6.11 $[\![\tau\colon \mathsf{n}' \rightsquigarrow]\!] = [\![\tau\colon \mathsf{m}' \rightsquigarrow]\!]$. Conclude by the definition of readback to λ-terms.
- *Closed under* ↻. Let $\mathtt{bVar}(\mathsf{l})\ \mathcal{Q}^{\Downarrow}\ \mathtt{bVar}(\mathsf{l}')$: we need to prove that $\mathsf{l}\ \mathcal{Q}^{\Downarrow}\ \mathsf{l}'$. By proposition 12.12, $\tau\colon \mathsf{n} \rightsquigarrow \mathtt{bVar}(\mathsf{l})$ and $\tau\colon \mathsf{m} \rightsquigarrow \mathtt{bVar}(\mathsf{l}')$ for some τ and $\mathsf{n}\ \mathcal{Q}\ \mathsf{m}$, and by proposition 6.11 $[\![\tau\colon \mathsf{n} \rightsquigarrow \mathtt{bVar}(\mathsf{l})]\!] = [\![\tau\colon \mathsf{m} \rightsquigarrow \mathtt{bVar}(\mathsf{l}')]\!]$. By the definition of the readback to λ-terms, $\mathtt{index}(\mathsf{l} \mid \tau\colon \mathsf{n} \rightsquigarrow) = \mathtt{index}(\mathsf{l}' \mid \tau\colon \mathsf{m} \rightsquigarrow)$. Conclude by proposition 12.27.

($\Rightarrow$) Assume that $\mathcal{Q}^{\Downarrow}$ is open, homogeneous, and closed under ↻, and let $\mathsf{n}\ \mathcal{Q}\ \mathsf{m}$. In order to prove that $[\![\mathsf{n}]\!] = [\![\mathsf{m}]\!]$, by proposition 6.11 it suffices to prove that for every trace τ the conditions *Trace Equivalence* and *Trace Propagation* hold:

- *Trace Equivalence.* Note that $\mathsf{n}\ \mathcal{Q}\ \mathsf{m}$ implies $\mathsf{n}\ \mathcal{Q}^{\Downarrow}\ \mathsf{m}$, and conclude by proposition 12.9.
- *Trace Propagation.* Assume $\tau\colon \mathsf{n} \rightsquigarrow \mathsf{n}'$ and $\tau\colon \mathsf{m} \rightsquigarrow \mathsf{m}'$. By proposition 12.12, $\mathsf{n}'\ \mathcal{Q}^{\Downarrow}\ \mathsf{m}'$. Note that n' and m' have the same label because $\mathcal{Q}^{\Downarrow}$ is homogeneous by hypothesis. By the property that the λ-graph is finite and acyclic, there exists a bound on the length of traces in the λ-graph, say B. The proof that $[\![\mathsf{n}']\!] = [\![\mathsf{m}']\!]$ proceeds by (course of value) induction on $B - |\tau|$, and by cases on the labels of the nodes (that must be identical because $\mathcal{Q}^{\Downarrow}$ is homogeneous):
  * Case $\mathtt{fVar}(x)\ \mathcal{Q}^{\Downarrow}\ \mathtt{fVar}(x')$. From the hypothesis that $\mathcal{Q}^{\Downarrow}$ is open it follows that $x = x'$, and we conclude.
  * Case $\mathtt{bVar}(\mathsf{l})\ \mathcal{Q}^{\Downarrow}\ \mathtt{bVar}(\mathsf{l}')$. From the hypothesis that $\mathcal{Q}^{\Downarrow}$ is closed under ↻ it follows that $\mathsf{l}\ \mathcal{Q}^{\Downarrow}\ \mathsf{l}'$. We need to show that $[\![\tau\colon \mathsf{n} \rightsquigarrow \mathtt{bVar}(\mathsf{l})]\!] = [\![\tau\colon \mathsf{m} \rightsquigarrow \mathtt{bVar}(\mathsf{l}')]\!]$, *i.e.* $\mathtt{index}(\mathsf{l} \mid \tau\colon \mathsf{n} \rightsquigarrow \mathtt{bVar}(\mathsf{l})) = \mathtt{index}(\mathsf{l}' \mid \tau\colon \mathsf{m} \rightsquigarrow \mathtt{bVar}(\mathsf{l}'))$. Since $\mathcal{Q}$ is a query, $\tau\colon \mathsf{n} \rightsquigarrow$ crosses l and $\tau\colon \mathsf{m} \rightsquigarrow$ crosses l'. The thesis $\mathtt{index}(\mathsf{l} \mid \tau\colon \mathsf{n} \rightsquigarrow \mathtt{bVar}(\mathsf{l})) = \mathtt{index}(\mathsf{l}' \mid \tau\colon \mathsf{m} \rightsquigarrow \mathtt{bVar}(\mathsf{l}'))$ follows from proposition 12.27.
  * Case **Abs**. By *i.h.* $[\![(\tau \cdot \downarrow)\colon \mathsf{n} \rightsquigarrow]\!] = [\![(\tau \cdot \downarrow)\colon \mathsf{m} \rightsquigarrow]\!]$. We conclude by the definition of readback, since $[\![\tau\colon \mathsf{n} \rightsquigarrow]\!] = \lambda[\![(\tau \cdot \downarrow)\colon \mathsf{n} \rightsquigarrow]\!]$ and $[\![(\tau \cdot \downarrow)\colon \mathsf{m} \rightsquigarrow]\!] = \lambda[\![(\tau \cdot \downarrow)\colon \mathsf{m} \rightsquigarrow]\!]$.
  * Case **App**. By *i.h.* $[\![(\tau \cdot \swarrow)\colon \mathsf{n} \rightsquigarrow]\!] = [\![(\tau \cdot \swarrow)\colon \mathsf{m} \rightsquigarrow]\!]$ and $[\![(\tau \cdot \searrow)\colon \mathsf{n} \rightsquigarrow]\!] = [\![(\tau \cdot \searrow)\colon \mathsf{m} \rightsquigarrow]\!]$. We conclude by the definition

of readback, since $[\![\tau : \mathsf{n} \rightsquigarrow]\!] = [\![(\tau \cdot \swarrow) : \mathsf{n} \rightsquigarrow]\!]\,[\![(\tau \cdot \searrow) : \mathsf{n} \rightsquigarrow]\!]$ and $[\![\tau : \mathsf{m} \rightsquigarrow]\!] = [\![(\tau \cdot \swarrow) : \mathsf{m} \rightsquigarrow]\!]\,[\![(\tau \cdot \searrow) : \mathsf{m} \rightsquigarrow]\!]$.

We conclude this section with the sharing equality theorem: there exists a sharing equivalence containing a given query $\mathcal{Q}$ if and only if the readbacks to locally nameless λ-terms of the nodes related by $\mathcal{Q}$ are equal.

Theorem 12.29 \ Sharing equality

Let $\mathcal{Q}$ be a query over a λ-graph G. There exists a sharing equivalence containing $\mathcal{Q}$ if and only if $[\![\mathsf{n}]\!] = [\![\mathsf{m}]\!]$ for every nodes n, m such that $\mathsf{n}\ \mathcal{Q}\ \mathsf{m}$.

Proof. By theorem 12.25, theorem 12.26, and proposition 12.28.

Corollary 12.30 \ Correctness of $\mathcal{Q}^{\#}$

Let $\mathcal{Q}$ be a query over a λ-graph G. $\mathcal{Q}^{\#}$ is a sharing equality if and only if $[\![\mathsf{n}]\!] = [\![\mathsf{m}]\!]$ for every nodes n, m such that $\mathsf{n}\ \mathcal{Q}\ \mathsf{m}$.

12.4 Up To Relations

Before concluding this chapter, we introduce the notion of relations that are propagated "up to" another relation. This new concept will be necessary when formulating the invariants of our sharing equality algorithm in chapter 14: since the algorithm constructs the required relations progressively, the invariants need to capture properties of relations that are temporarily incomplete.

Definition 12.31 \ Propagated upto

Let $\mathcal{R}, \mathcal{R}'$ be binary relations over the nodes of a λ-graph G. We say that $\mathcal{R}$ is propagated upto $\mathcal{R}'$ when $\mathcal{R}$ is closed under the following rules (which are variants of the rules ↙ ↘ ↓ in fig. 12.3):

1. $$\frac{\mathtt{App}(\mathsf{n}_1, \mathsf{n}_2)\ \mathcal{R}\ \mathtt{App}(\mathsf{m}_1, \mathsf{m}_2)}{\mathsf{n}_1\ \mathcal{R}'\ \mathsf{m}_1}$$
2. $$\frac{\mathtt{App}(\mathsf{n}_1, \mathsf{n}_2)\ \mathcal{R}\ \mathtt{App}(\mathsf{m}_1, \mathsf{m}_2)}{\mathsf{n}_2\ \mathcal{R}'\ \mathsf{m}_2}$$
3. $$\frac{\mathtt{Abs}(\mathsf{n})\ \mathcal{R}\ \mathtt{Abs}(\mathsf{m})}{\mathsf{n}\ \mathcal{R}'\ \mathsf{m}}$$

(Note the relation $\mathcal{R}'$ in the conclusion of the rules.)

When the relations $\mathcal{R}$ and $\mathcal{R}'$ are the same, the concept of propagated upto coincides with the usual concept of propagated:

Fact 12.32 \ Upto itself

If a relation $\mathcal{R}$ is propagated upto $\mathcal{R}$, then $\mathcal{R}$ is simply propagated.

Two properties of upto relations follow. The first proposition shows that the property of being propagated upto $\mathcal{R}'$ is monotonous on $\mathcal{R}'$.

Proposition 12.33 \ Monotonicity of upto

Let $\mathcal{R}, \mathcal{R}', \mathcal{R}''$ be binary relations over the nodes of a λ-graph G. If $\mathcal{R}$ is propagated upto $\mathcal{R}'$, and $\mathcal{R}' \subseteq \mathcal{R}''$, then $\mathcal{R}$ is propagated upto $\mathcal{R}''$.

Proof. Let $\mathcal{R}$ be propagated upto $\mathcal{R}'$, and let $\mathcal{R}''$ be a relation such that $\mathcal{R}' \subseteq \mathcal{R}''$. In order to show that $\mathcal{R}$ is propagated upto $\mathcal{R}''$, it suffices to check that it is closed under the rules of definition 12.31. We show that it is closed under the first rule, the proof for the other rules is similar. Let $\mathtt{App}(\mathsf{n}_1, \mathsf{n}_2)\ \mathcal{R}\ \mathtt{App}(\mathsf{m}_1, \mathsf{m}_2)$; since $\mathcal{R}$ is propagated upto $\mathcal{R}'$, it follows that $\mathsf{n}_1\ \mathcal{R}'\ \mathsf{m}_1$. Since $\mathcal{R}' \subseteq \mathcal{R}''$, we conclude with $\mathsf{n}_1\ \mathcal{R}''\ \mathsf{m}_1$.

The second proposition shows that the property of being propagated upto commutes with the equivalence closure:

Proposition 12.34 \ Upto *vs* equivalences

Let $\mathcal{R}$ be a binary relation over the nodes of a λ-graph G, and assume $\mathcal{R}$ to be homogeneous. If $\mathcal{R}$ is propagated upto $\mathcal{R}^*$, then $\mathcal{R}^*$ is propagated.

Proof. Let $\mathcal{R}$ be propagated upto $\mathcal{R}^*$. In order to prove that $\mathcal{R}^*$ is propagated, by fact 12.32 it suffices to prove that $\mathcal{R}^*$ is propagated upto $\mathcal{R}^*$. The rest of the proof is similar to the proof of proposition 12.20.

Lastly, we extend the notion of upto to pre-bisimulations:

Definition 12.35 \ Pre-bisimulation upto

Let $\mathcal{R}, \mathcal{R}'$ be binary relations over the nodes of a λ-graph G. We say that $\mathcal{R}$ is a *pre-bisimulation upto* $\mathcal{R}'$ when $\mathcal{R}$ is homogeneous and propagated upto $\mathcal{R}'$.

Chapter 13

Computing Sharing Equality

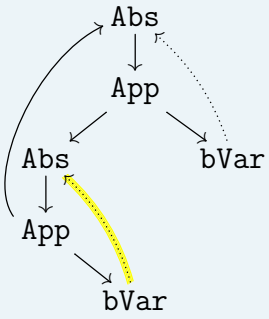
From now on we focus on the algorithmic side of sharing equality, *i.e.* the problem of effectively deciding and computing it.

In chapter 14 we will provide a sharing equality algorithm (algorithm 4) based on the Paterson–Wegman algorithm for first-order unification (algorithm 1). In this chapter instead we discuss the literature on sharing equality, and other existing problems that are related to it.

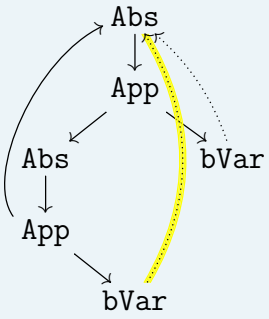
We are aware of very few works that consider the problem of sharing equality in presence of binders:

- One is by Accattoli and Lago, 2012, where the authors present an algorithm with quadratic running time based on dynamic programming. That algorithm proceeds by filling an unfolding matrix with previous results of intermediate readbacks, so that it avoids repeating the same equality check many times leading to an exponential blowup. As a matter of fact, the algorithm does not read back intermediate subterms completely—as this would cause an exponential blowup too—but it uses "relative unfoldings" that take into account the reused structure without constructing the complete readback.

- Another paper is by Grabmayer and Rochel, 2014, where the λ-calculus is extended with `letrec`-expressions, *i.e.* recursive `let` definitions introduced in section 6.2. In the setting of cyclic λ-graphs, the problem of sharing equality is much harder because terms are to be compared not just up to α-equivalence, but up to α-equivalence of their infinite unfolding. This difference may seem harmless, but it creates much complexity due to the interaction between cycles and scopes: as it turns out, bisimilarity is not sufficient to capture sharing equality up to infinite unfoldings. Consider the two graphs in fig. 13.1:



$t' \coloneqq \texttt{letrec}\ f = \lambda x.(\lambda y.\ f\ y)\ x\ \texttt{in}\ f$



$t'' \coloneqq \texttt{letrec}\ f = \lambda x.(\lambda y.\ f\ x)\ x\ \texttt{in}\ f$

Bisimilar $\lambda_{\texttt{letrec}}$-graphs / **Figure 13.1**

The two graphs in fig. 6.7 are bisimilar, however the corresponding $\lambda_{\texttt{letrec}}$-terms t' and t'' are not equivalent: t' unfolds to $u = \lambda x.(\lambda y.\ (\cdots)\ y)\ x$ defined in fig. 6.7 at fig. 6.7, while t'' unfolds to $\lambda x.(\lambda y.\ (\cdots)\ x)\ x$, clearly not α-equivalent.

To overcome this problem, the algorithm proposed by Grabmayer and Rochel, 2014 performs a complex encoding of the terms in first-order graphs where additional nodes are introduced to record information about scopes. Finally they employ a variant of the algorithm by Hopcroft and Karp, 1971 to test equivalence of *deterministic finite automata* (DFA, see also below). The latter algorithm has quasi-linear complexity, i.e. it runs in time $O(n \times \alpha(n))$ where $\alpha(\cdot)$ is the inverse of the Ackermann function[1] and n is the size of the graphs in input. However, when the reduction to first-order graphs is considered, the final complexity of Grabmayer and Rochel's algorithm is $O(n^2 \times \alpha(n))$.

13.1 Related Problems

There are various problems that are closely related to sharing equality, and that are also treated with bisimilarity-based algorithms. In the following subsections we are going to list similarities and differences with respect to other problems (like *DFA equivalence* and *unification*) and other techniques (like *hash-consing*).

Equivalence of DFA

A deterministic finite automaton is a finite-state machine that accepts or rejects strings of symbols. Formally, it is a finite directed graph where nodes represent states, and edges represent state transitions, labelled by symbols from a given alphabet; a start state and possibly various accepting states are provided.

[1] The inverse Ackermann function is a function growing as extremely slowly as the Ackermann function grows fast. It is used in computer science to provide time bounds of various algorithms, in this case the operations of the union-find data structure.

Automata do not have binders like λ-graphs, and yet they are structurally more general since they allow arbitrary directed cycles, not even dominated (cf. definition 6.8). As already pointed out above, however, the best DFA equivalence algorithm (Hopcroft and Karp, 1971) is only quasi-linear.

One may obtain from Hopcroft-Karp's algorithm a quasi-linear time algorithm for sharing equality by proceeding in a similar way as we are going to do in chapter 14: in fact, one can see the nodes of a λ-graph as states of a DFA, and each directed edge labelled with direction d as a transition labelled by d. Hopcroft-Karp's algorithm builds an equivalence relation over the nodes of the DFA by using a union-find data structure to record the equivalence classes, and it mainly consists of a loop over a `todo` queue that records the pairs of nodes that must be checked and propagated. It is possible to prove that (in case of success) the algorithm constructs the spreading $\mathcal{Q}^{\#}$ of the input query $\mathcal{Q}$; one can then exploit the sharing equality theorem (theorem 12.29) and perform the check for sharing equality in two phases, just as we do in algorithm 3. The complexity of the resulting algorithm is then pseudo-linear, because the operations on the union-find data structure can be performed in only *almost* constant time (Tarjan, 1975). Such a quasi-linear algorithm based on Hopcroft-Karp may in practice run faster than the linear-time algorithm that we provide in chapter 14, but as already mentioned we are interested in linearity for theoretical reasons.

Alpha-equivalence

Clearly the closest problem to sharing equality is plain α-equivalence. Schmidt-Schauß, Rau, and Sabel, 2013 discuss algorithms for α-equivalence extended with further principles (*e.g.* permutations of `letrec` expressions), but not up to unfolding.

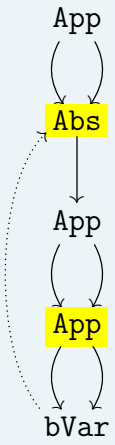
Hash-consing

As usual, we may circumvent the problem of α-equivalence by switching to nameless λ-terms. In actual programming languages, instead of using λ-graphs to represent sharing, one can switch to plain terms with De Bruijn indices and exploit the implicit sharing in memory of terms by reusing pointers to the same memory locations. A well-known technique is *hash-consing* (Ershov, 1958; Goto, 1974), which allows to share purely functional data and is traditionally realised through a hash table (Allen, 1978). With hash-consing, to check that two terms are sharing equivalent it suffices to compute their hash[2], but first the terms have to be recursively hash-consed (*i.e.* maximally shared), which requires *quasi*linear running time. Another option is an eager approach to conversion of λ-terms, in which one mantains all terms hash-consed (*i.e.* maximally shared) at each reduction step, keeping trace in a huge table of all the pairs of convertible terms previously encountered; however, computing a hash and accessing a table at each reduction step may be quite inefficient.

[2]The hash of an object is computed by means of an hash function, mapping each object to a fixed-side value that acts as key in a table.

Higher-order unification

Unification problems are usually represented as systems of equations between terms containing metavariables. A λ-graph itself can be encoded in linear time as a unification problem of λ-terms by using metavariables to represent common subgraphs. Take for example the λ-graph in fig. 13.2, that represents the unshared λ-term $(\lambda x.(xx)(xx))(\lambda x.(xx)(xx))$:



Example of λ-graph / **Figure 13.2**

The most straightforward way to translate sharing to a system of unification equations is by using a new unification metavariable for each (non-variable) node of the λ-graph that has more than one parent. As for the λ-graph in fig. 13.2, we introduce two metavariables X and Y corresponding respectively to the abstraction node and to the application node two levels below it (highlighted). Then the root node of the λ-graph is represented by the term XX together with the system of equations $\{X \doteq (\lambda x.YY), Y \doteq (xx)\}$ where x is allowed to occur in the substitution for Y. If one desires to check whether that λ-graph is sharing equivalent to another λ-graph represented by the term t, it suffices to add to the system the equation $XX \doteq t$ and check if the resulting unification problem is *solvable*: here solvable means that there exists a substitution (a mapping from metavariables to terms) which makes the left-hand and the right-hand sides of each equation in the system equal.

The most natural notion of unification in this setting seems higher-order unification, however this is clearly an overkill. Higher-order unification is in general undecidable (Huet, 1973), and it unifies terms up to full $\alpha\beta$-conversion, while we strictly require α-conversion only. In the following sections, we consider decidable fragments of higher-order unification that are closer to the problem of sharing equality.

Nominal unification. Nominal unification is unification up to α-equivalence of λ-calculi extended with *name swappings*, in the nominal tradition. It was first studied by Urban, Pitts, and Gabbay, 2003 and efficient algorithms are due to two groups, Calvès & Fernández and Levy & Villaret, adapting Paterson–Wegman and Martelli–Montanari first-order unification al-

gorithms. Nominal unification is very close to sharing equality (which is a special case where name swappings do not occur) but the known best algorithms (Calvès and Fernández, 2010b; Levy and Villaret, 2010) are only quadratic. Calvès, 2013 presents an abstract algorithm to solve nominal unification problems, based on Paterson–Wegman and subsuming the two best known algorithms. The key idea is to enrich Paterson–Wegman with an additional data structure, called *modality*, that records freshness constraints and identification/swapping of variables. Managing the modality has a cost: the two best algorithms do that in different ways, but always with a linear overhead per operation, yielding the final quadratic complexity. Our contribution of chapter 14 can be seen as showing that no explicit modality data structure is required at all — and thus no overhead at all — to check sharing equality: the information about identification of bound variables is already encoded by the equivalence class of their binders.

Nominal matching. This is a special case of nominal unification. Calvès and Fernández, 2010a present an algorithm for nominal matching that is linear, but *only* on unshared input terms.

Pattern unification. Miller's *pattern unification* can also be stripped down to test sharing equality. Pattern unification is a decidable fragment of higher-order unification where the terms to be unified are so-called *higher-order patterns*, terms where metavariables must be applied to distinct bound variables. For example, the λ-graph of fig. 13.2 can again be represented by the term XX together with the system of equations

$$\{X \doteq (\lambda x.\,(Y\,x)\,(Y\,x)), Y\,z \doteq (z\,z)\}.$$

Qian, 1993 presents a PW-inspired algorithm for pattern unification — claiming linear complexity — that actually seem to work only on unshared terms. We have not investigated the relation between the two linear-time algorithms, but we note that Qian's algorithm is very involved, requires a lot more code and more data structures and the overall algorithm would surely have higher computational constants than ours. Moreover, according to Levy and Villaret, 2010, "it is really difficult to obtain a practical algorithm from the proof described by Qian, 1993".

First-order unification. The most elementary kind of unification is first-order unification, where terms consist either of metavariables, constants, or functions applied to arguments. At first sight, first-order terms do not seem to accommodate the higher-order nature of λ-terms: while globally free variables in λ-terms can be safely mapped to first-order constants, bound variables cannot be mapped to constants as they must be handled up to α-equivalence (first-order unification fails when comparing the first-order terms $\lambda x.\lambda y.x$ and $\lambda y.\lambda x.y$ when x and y are treated as constants). In addition, variables cannot be mapped to unification metavariables because they cannot be instantiated with other terms during unification: unification succeeds when comparing the terms $\lambda x.\lambda y.x$ and $\lambda y.\lambda x.x$ when x and y are treated as metavariables, but the two terms are not α-equivalent. In this case even resorting to the

(locally) nameless representations of λ-terms to obtain a first-order structure does not solve the issue, because then the sharing in λ-graphs cannot be translated as easily to a unification problem: shared subgraphs cannot be mapped to the same unification metavariable, since the de Bruijn indices of the variables that occur in terms depend on the number of abstraction that are above them. For example, consider the λ-term $\lambda x.\,(x\,x)(\lambda y.\,x\,x)$ where we want to share all the occurrences of the subterm $x\,x$: without names the λ-term becomes $\lambda.\#0\,\#0\,(\lambda.\,\#1\,\#1)$, where the desired subterms have become different and cannot be shared.

Even though in some way a higher-order problem, we can actually reduce sharing equality to first-order unification (plus an additional test for variables). This follows from the theory developed in chapter 12, where we reduced sharing equality to "first-order" bisimulations over the λ-graph, *i.e.* the propagation and the spreading of a given query $\mathcal{Q}$. By universality and the sharing equality theorem (corollary 12.30) it suffices to first compute one of these first-order relations, say $\mathcal{Q}^{\#}$, and then simply check separately the conditions for free and bound variables. Well-known first-order unification algorithms that use equivalences classes compute exactly $\mathcal{Q}^{\#}$ over a term graph.

There are basically two linear-time algorithm for first-order unification, Paterson and Wegman, 1978 (PW) and Martelli and Montanari, 1982 (MM). Both rely on sharing to run in linear time. PW even takes terms with sharing as inputs, while MM deals with sharing in a less direct way, except in its less known variant (Martelli and Montanari, 1977) that takes in input terms shared using the *Boyer-Moore* technique (Boyer and Moore, 1972).

13.2 The Paterson–Wegman Algorithm

The Paterson-Wegman (PW) algorithm, we said, is the first-order algorithm that we will adapt in the next chapter to obtain a sharing equality algorithm. PW was introduced by Paterson and Wegman, 1978, and then refined by Champeaux, 1986.

Just for reference, we provide in algorithm 1 the original PW algorithm. We delay the discussion about the algorithm to the next chapter, where we will also explain the required data structures, and the differences between PW and our sharing equality algorithm.

Algorithm 1 \ Paterson–Wegman Unification Algorithm

Data: a term graph G

```
1  Procedure Main(n, m)
       // test n and m for unifiability
2      create undirected edge n ∼ m
3      foreach function node n do Finish(n)
4      foreach meta-variable node n do Finish(n)
       // unified
5  Procedure Finish(n)
6      if canonic(n) undefined then canonic(n) := n
7      else FAIL
8      s := new stack
9      push n on s
10     while s is non-empty do
11         m := pop s
12         if n,m have different function symbols then
13             FAIL
14         while m has some parent p do
15             Finish(p)
16         while there is an undirected edge m ∼ p do
17             if canonic(p) undefined then canonic(p) := n
18             else if canonic(p) ≠ n then FAIL
19             delete undirected edge m ∼ p
20             push p on s
21         if n ≠ m then
22             if m is a function node with outdegree q > 0 then
23                 create undirected edges {jth child(n) ∼ jth child(m) | 1 ≤ j ≤ q}
24                 delete m and directed arcs out of m
25     delete n and directed arcs out of n
```

Chapter 14

Sharing Equality is Linear

In this chapter we provide our linear-time algorithm for sharing equality. By the universality of spreaded queries (theorem 12.26), checking the satisfability of a query $\mathcal{Q}$ over a λ-graph G boils down to compute its spreading $\mathcal{Q}^{\#}$ over G, and then checking that it is a sharing equality: the fact that the requirements on variables are modular to the propagation requirements is one of our main contributions. Indeed it is possible to check sharing equality in two phases:

1. Blind Check : building $\mathcal{Q}^{\#}$ and at the same time checking that it is a pre-bisimulation, *i.e.* that it is an homogeneous relation;
2. Variables Check : verifing that $\mathcal{Q}^{\#}$ is a sharing equivalence by checking the conditions for free and bound variable nodes.

Of course, the difficulty is doing it in linear time, and it essentially lies in the Blind Check. The fact that our algorithm first uses PW ignoring variable names, and only after checks that the computed relation is a sharing equality is also a direct application of the "first-order nominal link" of Calvès and Fernández, 2010b.

The rest of this chapter presents two algorithms, the *Blind Check* (algorithm 2) and the *Variables Check* (algorithm 3), with proofs of correctness and completeness, and complexity analyses. The second algorithm is actually straightforward. Be careful, however: the Variables check algorithm is trivial just because the subtleties of this part have been isolated in chapter 12.

14.1 The Blind Check

In this section we introduce the basic concepts for the Blind Check, plus the algorithm itself. Our algorithm is an adaptation of Paterson and Wegman's, and it relies on the same key ideas in order to be linear. With respect to PW's original algorithm, our reformulation does not rely on their notions of *dead/alive nodes* used to keep track of the nodes already processed; in addition it is not destructive, *i.e.* it does not remove edges and nodes from the graph, hence

it is more suitable for use in computer tools where the λ-terms to be checked for equality need not be destroyed. Another contribution of this part is a formal proof of correctness and completeness, obtained via the isolation of properties of program runs.

Intuitions for the Blind Check. Paterson and Wegman's algorithm is based on a tricky, linear time visit of the λ-graph. It addresses two main efficiency issues:

1. *The spreaded query is quadratic*: the number of pairs in the spreaded query $\mathcal{Q}^{\#}$ can be quadratic in the size of the λ-graph. An equivalence class of cardinality n has indeed $\Omega(n^2)$ pairs for the relation—this is true for every equivalence relation. This point is addressed by rather computing a linear relation $=_{\mathsf{c}}$ (same canonic, definition 14.17) *generating* $\mathcal{Q}^{\#}$, based on keeping a *canonical element* for every sharing equivalence class.

2. *Merging equivalence classes*: merging equivalence classes is an operation that, for as efficient as it may be, it is not a costant time operation. The trickiness of the visit of the λ-graph is indeed meant to guarantee that, if the query is satisfiable, one never needs to merge two equivalence classes, but only to add single elements to classes.

More specifically, the ideas behind algorithm 2 are:

- *Top-down recursive exploration*: the algorithm can start on any node, not necessarily a root. However, when processing a node n the algorithm first makes a recursive call on the parents of n that have not been visited yet. This is done to avoid the risk of reprocessing n later because of some new equality requests on n coming from a parent processed after n.

- *Query edges*: the query is represented through additional *undirected query edges* between nodes, and it is propagated on child nodes by adding further query edges. The query is propagated carefully, on-demand. The fully propagated query is never computed, because, as explained, in general its size is quadratic in the number of nodes.

- *Canonic edges*: when a node is visited, it is assigned a canonic node that is a representative of its sharing equivalence class. This is represented via a *directed canonic edge*, which is implemented as a pointer.

- *Building flag*: each node has a boolean "building" flag that is used only by canonic representatives and notes the state of construction of their equivalence class. When undefined, it means that that node is not currently designated as canonic; when true, it means that the equivalence class of that canonic is still being computed; when false, it signals that the equivalence class has been completely computed.

- *Failures and cycles*: the algorithm fails in three cases. First, when it finds two nodes in the same class that are not homogeneous (line 26), because then the approximation of $\mathcal{Q}^{\#}$ that it is computing cannot be a pre-bisimulation. In the two other cases (on line 13 and line 17) the algorithm uses the fact that a canonic edge is already present to infer that it found a cycle up to $\mathcal{Q}^{\#}$, and so, again $\mathcal{Q}^{\#}$ cannot be a pre-bisimulation.

Algorithm 2 \ The Blind Check Algorithm

Data: an initial state
Result: $\mathcal{F}ail$ or a final state

```
1  Procedure BlindCheck()
2      for every node n do
3          if canonic(n) undefined then BuildClass(n)
4  Procedure BuildClass(c)
5      canonic(c) := c
6      building(c) := true
7      queue(c) := {c}
8      while queue(c) is non-empty do
9          n := queue(c).pop()
10         for every parent m of n do
11             case canonic(m) of
12                 undefined ⇒ BuildClass(m)
13                 c' ⇒ if building(c') then FAIL
14         for every ∼neighbour m of n do
15             case canonic(m) of
16                 undefined ⇒ EnqueueAndPropagate(m, c)
17                 c' ⇒ if c' ≠ c then FAIL
18     building(c) := false
19 Procedure EnqueueAndPropagate(m, c)
20     case m, c of
21         Abs(m'), Abs(c') ⇒ create edge m' ∼ c'
22         App(m1, m2), App(c1, c2) ⇒
23             create edges m1 ∼ c1 and m2 ∼ c2
24         bVar(_), bVar(_) ⇒ ()
25         fVar(_), fVar(_) ⇒ ()
26         _, _ ⇒ FAIL
27     canonic(m) := c
28     queue(c).push(m)
```

- *Building a class*: calling BuildClass(n) boils down to
  1. collect without duplicates all the nodes in the intended sharing equivalence class of n, that is, the nodes related to n by a sequence of query edges. This is done by the *while* loop at line 14, that first collects the nodes queried with n and then iterates on the nodes queried with them. These nodes are inserted in a queue;
  2. set n as the canonical element of its class, by setting the canonical edge of every node in the class (including n) to n;
  3. propagate the query on the children (in case n is a `Abs` or a `App` node), by adding query edges between the corresponding children of every node in the class and their canonic.
  4. Pushing a node in the queue, setting its canonic, and propagating the query on its children is done by the procedure EnqueueAndPropagate.
- *Linearity*: let us now come back to the two efficiency issues we mentioned before:
  - *Merging classes*: the recursive calls are done in order to guarantee that when a node is processed all the query edges for its sharing class are already available, so that the class shall not be extended nor merged with other classes later on during the visit of the λ-graph.
  - *Propagating the query*: the query is propagated only *after* having set the canonics of the current sharing equivalence class. To explain, consider a class of k nodes, which in general can be defined by $\Omega(k^2)$ query edges. Note that after canonization, the class is represented using only $k-1$ canonic edges, and thus the algorithm propagates only $O(k)$ query edges—this is why the number of query edges is kept linear in the number of the nodes (assuming that the original query itself was linear). If instead one would propagate query edges *before* canonizing the class, then the number of query edges may grow quadratically.

States. As explained, the algorithm needs to enrich λ-graphs with a few additional concepts, namely *canonic edges*, *query edges*, *building flags*, *queues*, and *execution stack*, all grouped under the notion of *program state*.

A *state* $\mathcal{S}$ of the algorithm is either $\mathcal{F}ail$ or a tuple

$$(G, \texttt{undir}, \texttt{canonic}, \texttt{building}, \texttt{queue}, \texttt{active})$$

where G is a λ-graph, and the remaining data structures have the following properties:

- *Undirected query edges ($\sim$)*: `undir` is a *multiset* of undirected query edges, pairing nodes that are expected to be placed by the algorithm in the same sharing equivalence class. Undirected loops are admitted and there may be multiple occurrences of an undirected edge between two nodes. More precisely, for every undirected edge between n and m with multiplicty k in the state, both (n, m) and (m, n) belong with multiplicity k to `undir`. We denote by $\sim$ the binary relation over G such that $\mathsf{n} \sim \mathsf{m}$ iff the edge (n, m) belongs to `undir`.

- *Canonic edges (*c*)*: nodes may have one additional `canonic` directed edge pointing to the computed canonical representative of that node. The partial function mapping each node to its canonical representative (if defined) is noted $\mathsf{c}(\bullet)$. We then write $\mathsf{c}(\mathsf{n}) = \mathsf{m}$ if the canonical of n is m, and $\mathsf{c}(\mathsf{n}) =$ undefined otherwise. By abuse of notation, we also consider $\mathsf{c}(\bullet)$ a binary relation on G, where $\mathsf{n}\ \mathsf{c}\ \mathsf{m}$ iff $\mathsf{c}(\mathsf{n}) = \mathsf{m}$.

- *Building flags (*b*)*: nodes may have an additional boolean flag `building` that signals whether an equivalence class has or has not been constructed yet. The partial function mapping each node to its building flag (if defined) is noted b. We then write $\mathsf{b}(\mathsf{n}) = \mathsf{true} \mid \mathsf{false}$ if the building flag of n is defined, and $\mathsf{b}(\mathsf{n}) =$ undefined otherwise.

- *Queues (*q*)*: nodes have a queue data structure that is used only on canonic representatives, and contains the nodes of the class that are going to be processed next. The partial function mapping each node to its queue (if defined) is noted q.

- *Active Calls (*`active`*)*: a program state contains information on the execution stack of the algorithm, including the active procedures, local variables, and current execution line. We leave the concept of execution stack informal; we only define more formally `active`, which records the order of visit of equivalence classes that are under construction, and that is essential in the proof of completeness of the algorithm. `active` is an abstraction of the implicit execution stack of active calls to the procedure BuildClass where only (part of) the activation frames for BuildClass(c) are represented. Formally, `active` is simply a sequence of nodes of the λ-graph, and $\texttt{active} = [c_1, \ldots, c_K]$ if and only if:

  - *Active:* BuildClass(c_1), …, BuildClass(c_K) are exactly the calls to BuildClass that are currently active, *i.e.* have been called before $\mathcal{S}$, but have not yet returned;
  - *Call Order:* for every $0 < i < j \leq K$, BuildClass(c_i) was called before BuildClass(c_j).

Moreover we introduce the following concepts:

- *Program transition*: the change of state caused by the execution of a piece of code. For the sake of readability, we avoid a technical definition of transitions; roughly, a transition is the execution of a line of code, as they appear numbered in the algorithm itself. When the line is a `while` loop, a transition is an iteration of the body, or the exit from the loop; when the line is a `if-then-else`, a transition is entering one of the branches according to the condition; and so on.

- *Fail state*: $\mathcal{F}ail$ is the state reached after executing a `FAIL` instruction; it has no attributes and no transitions.

- *Initial state* $\mathcal{S}_0$: a non-$\mathcal{F}ail$ state with the following attributes:

  - *Initial* $\sim$*edges:* simply the initial query, *i.e.* $\sim \coloneqq \mathcal{Q}$.
  - *Initial assignments:* $\mathsf{c}(\mathsf{n})$, $\mathsf{b}(\mathsf{n})$, and $\mathsf{q}(\mathsf{n})$ are undefined for every node n of G.

  - *Initial transition:* the first transition is a call to BlindCheck().

- *Program run*: a sequence of program states starting from $\mathcal{S}_0$ obtained by consecutive transitions.
- *Reachable*: a state which is the last state of a program run.
- *Failing*: a reachable state that transitions to $\mathcal{F}ail$.
- *Final*: a non-$\mathcal{F}ail$ reachable state that has no further transitions.

Details about how the additional structures of enriched λ-graphs are implemented are given in section 14.1.4, where the complexity of the algorithm is analysed.

In the following sections, we first show general properties of algorithm 2 (section 14.1.1); then we prove that the algorithm is sound (section 14.1.2), complete (section 14.1.3), and that it runs in linear time (section 14.1.4).

14.1.1 General Properties

In this section we prove general properties of program runs, grouped according to the concepts that they analyse: canonic assignment, BuildClass, EnqueueAndPropagate, enqueuing, dequeuing, parents, $\sim$neighbours (i—vii).

i Canonic assignment. The algorithm assigns a canonic to each node n: intuitively, $\mathsf{c}(\mathsf{n})$ is the canonic representative of the equivalence class to which n belongs.

The following proposition is a key property required by many of the next results: it shows that nodes cannot change class during a program run, *i.e.* once a node is assigned to an equivalence class, it is never re-assigned.

Proposition 14.1 \ Canonic assignment is definitive

Let n be a node. In every program run:

- $\mathsf{c}(\mathsf{n})$ is never assigned to undefined;
- $\mathsf{c}(\mathsf{n})$ is assigned at most once.

Proof. No line of the algorithm assigns a canonic to undefined, *i.e.* after $\mathsf{c}(\mathsf{n})$ is defined, it remains defined throughout the program run.
In order to show that $\mathsf{c}(\mathsf{n})$ is never assigned twice, it suffices to show that, whenever a canonic is assigned, it was previously undefined. A canonic is assigned only during the execution of two lines of the algorithm:

- Line 5, during the execution of BuildClass(c) that assigns $\mathsf{c}(c) := c$. Note that only lines 3 and 12 can call BuildClass, and both lines check beforehand whether the canonic of c is undefined.
- Line 27, during the execution of EnqueueAndPropagate(m, c) that assigns $\mathsf{c}(\mathsf{m})$ to

c. Only line 16 can call EnqueueAndPropagate, and that line checks beforehand whether the canonic of m is undefined.

The following proposition shows that the mechanism of canonic representatives is correct, *i.e.* if c is assigned as canonic of an equivalence class, then c is itself a representative of that class:

Proposition 14.2 \ $\mathsf{c}(\bullet)$ is idempotent

Let c, n be nodes, and $\mathcal{S}$ be a reachable state such that $\mathsf{c}(\mathsf{n}) = c$. Then $\mathsf{c}(c)$ is defined and $\mathsf{c}(c) = c$.

Proof. Let n be a node. It suffices to show that whenever $\mathsf{c}(\mathsf{n})$ is assigned to c, it holds that $\mathsf{c}(c) = c$; once true, this assertion cannot be later falsified because the canonic assignment is definitive (proposition 14.1).
A canonic is assigned only during the execution of two lines of the algorithm:

- Line 5, during the execution of BuildClass(c). In this case $\mathsf{n} = c$ and we conclude.
- Line 27, during the execution of EnqueueAndPropagate(m, c) that assigns $\mathsf{c}(\mathsf{m})$ to c. Only line 16 of BuildClass(c) can call EnqueueAndPropagate(m, c), and that line comes after line 5 which sets $\mathsf{c}(c)$ to c. Therefore $\mathsf{c}(c) = c$ in $\mathcal{S}$ as well, because the canonic assignment is definitive (proposition 14.1).

During a program run, nodes are assigned a canonic *i.e.* temporary equivalence classes are extended with new nodes. If the algorithm terminates successfully, the equivalence classes cover the whole set of nodes, as the following proposition shows:

Proposition 14.3 \ Canonic assignment completed

In a final state $\mathcal{S}_f$, every node has a canonic assigned.

Proof. Let n be a node, and let us show that $\mathsf{c}(\mathsf{n})$ is defined in $\mathcal{S}_f$. Since the algorithm terminated, execution exited the main loop on line 2. That loop iterates on every node n of the λ-graph, therefore for every n there must exist a state $\mathcal{S}$ prior to $\mathcal{S}_f$ such that the next transition is the execution of line 3 with local variable n. Line 3 first checks whether $\mathsf{c}(\mathsf{n})$ is defined. If it is, it does nothing, and we can conclude because $\mathsf{c}(\mathsf{n})$ must be defined in $\mathcal{S}_f$ too since the canonic assignment is definitive (proposition 14.1). If instead $\mathsf{c}(\mathsf{n})$ is not defined in $\mathcal{S}$, then BuildClass(n) is called; when BuildClass(n) is executed, n is assigned a canonic (line 5), and again the canonic is still assigned in $\mathcal{S}_f$ since the canonic assignment is definitive (proposition 14.1).

ii BuildClass. The procedure BuildClass(c) has the effect of constructing the equivalence class of a node c, where c is chosen as canonic representative of that class. When Build-

Class(c) returns, the equivalence class of c is finalised.

Proposition 14.4 \ No multiple calls to BuildClass(n)

In every program run, BuildClass(n) is called at most once for every node n.

Proof. Only line 3 and line 12 can call BuildClass(n), and only if the canonic of n is undefined. Right after BuildClass(n) is called, line 5 is executed, and a canonic is assigned to n. Therefore another call to BuildClass(n) is not possible in the future, because $\mathsf{c}(\mathsf{n})$ cannot be ever assigned again to undefined (proposition 14.1). ∎

A node c can be assigned as a canonic representative only if BuildClass(c) has been called:

Proposition 14.5 \ Only BuildClass(c) designates canonics

Let c, n be nodes, and $\mathcal{S}$ be a reachable state such that $\mathsf{c}(\mathsf{n}) = c$. Then BuildClass($c$) has been called before $\mathcal{S}$.

Proof. c is assigned as a canonic only during the execution of two lines of the algorithm:

- Line 5, during the execution of BuildClass(c).
- Line 27, during the execution of EnqueueAndPropagate(m, c) that sets $\mathsf{c}(\mathsf{m})$ to c. Only line 16 of BuildClass(c) can call EnqueueAndPropagate(m, c). ∎

iii EnqueueAndPropagate. The function EnqueueAndPropagate(m, c) adds the node m to the equivalence class represented by c, and delays the processing of m by pushing it into $\mathsf{q}(c)$.

Proposition 14.6 \ No multiple calls to EnqueueAndPropagate(n, —)

In every program run, EnqueueAndPropagate(n, —) is called at most once for every node n.

Proof. Only line 16 can call EnqueueAndPropagate(m, c), and only if the canonic of m is undefined. After EnqueueAndPropagate(m, c) is called, line 27 is executed, and a canonic is assigned to m. Therefore another call to EnqueueAndPropagate(m, —) is not possible in the future, because $\mathsf{c}(\mathsf{m})$ cannot be ever assigned again to undefined (proposition 14.1). ∎

iv Enqueuing. Two lines of the algorithm push a node to a queue: line 7, when $\mathsf{q}(c)$ is created and c pushed to it; line 28, when m is pushed to $\mathsf{q}(c)$. Intuitively, pushing a node to $\mathsf{q}(c)$ results in delaying its processing by the algorithm; the node will be fully processed only later, after being popped from the queue (from line 9 to line 17).

Proposition 14.7 \ Enqueue once

In a program run, every node n is enqueued at most once.

Proof. The algorithm enqueues a node only shortly after setting its canonic:

- on line 7, after assigning a canonic on line 5;
- on line 28, right after assigning a canonic on line 27.

Since the canonic of a node can be assigned at most once (proposition 14.1), it follows that a node can be enqueued at most once. ∎

The following lemma shows that each queue $\mathsf{q}(c)$ is a subset of the equivalence class with canonic representative c:

Proposition 14.8 \ Queue $\subseteq$ Equivalence class

Let $\mathcal{S}$ be a reachable state, and n a node. If $\mathsf{n} \in \mathsf{q}(c)$, then $\mathsf{c}(\mathsf{n})$ is defined and $\mathsf{c}(\mathsf{n}) = c$.

Proof. Let n be a node. Since the canonic assignment is definitive (proposition 14.1), it suffices to check that $\mathsf{c}(\mathsf{n}) = c$ whenever n is enqueued to $\mathsf{q}(c)$. A node is enqueued only during the execution of two lines of the algorithm:

- Line 7 during the execution of BuildClass(c). In this case $\mathsf{n} = c$, and the canonic of c was just assigned to c itself on line 5.
- Line 28 during the execution of EnqueueAndPropagate(m, c). Line 28 is executed right after line 27, which sets $\mathsf{c}(\mathsf{m}) := c$. ∎

v Dequeuing. Nodes are popped from queues only on line 9. Once a node is dequeued, the algorithm proceeds by first visiting its parents (line 10) and only then its $\sim$neighbours (line 14).

Proposition 14.9 \ Dequeued nodes have correct canonic

Let $\mathcal{S}$ be a state reached after the execution of line 9 with locals c, n. Then $\mathsf{c}(\mathsf{n})$ is defined and $\mathsf{c}(\mathsf{n}) = c$.

Proof. Let n be a node. $\mathsf{n} \in \mathsf{q}(c)$ in the state $\mathcal{S}'$ right before the execution of line 9 with locals c, n. Therefore $\mathsf{c}(\mathsf{n}) = c$ in $\mathcal{S}'$ by proposition 14.8. Conclude with $\mathsf{c}(\mathsf{n}) = c$ in $\mathcal{S}$ because the canonic assignment is definitive (proposition 14.1). ∎

We introduce the following notion of "processed node", meaning that the node has been processed by the loop in the body of BuildClass.

Definition 14.10 \ Processed node

In a reachable state $\mathcal{S}$, we say that a node n has already been *processed* if Lines 9–17 with locals c, n have already been executed (for some c).
In other words, a node n is processed after it is dequeued and its parents and $\sim$neighbours visited.

After BuildClass(c) returns, all the nodes in the equivalence class of c are processed:

Proposition 14.11 \ Equivalence class processed

Let c, n be nodes, and $\mathcal{S}$ a reachable state such that BuildClass(c) has already returned. If $\mathsf{c}(\mathsf{n}) = c$, then n is processed.

Proof. $\mathsf{c}(\mathsf{n})$ is assigned to c only during the execution of BuildClass(c) (proposition 14.5). First, note that shortly after $\mathsf{c}(\mathsf{n})$ is assigned to c (line 5 or line 27), n is enqueued to $\mathsf{q}(c)$ (resp. line 7 and line 28).
In both cases, n is enqueued to $\mathsf{q}(c)$ when the execution of the while loop on line 8 has not terminated yet: on line 7 before the beginning of the loop, on line 28 during the execution of the loop, since EnqueueAndPropagate is called from line 16 which is inside the while loop.
Note also that a node n is dequeued from $\mathsf{q}(c)$ only on line 9 during the execution of BuildClass(c), and that BuildClass(c) can be called at most once (proposition 14.4).
In conclusion, n was enqueued to $\mathsf{q}(c)$ before the completion of the while loop on $\mathsf{q}(c)$ on line 8, and since BuildClass(c) has already returned and the while loop terminated, at some point n was dequeued from $\mathsf{q}(c)$ on line 9, and the body of the loop executed with locals c, n. ∎

Proposition 14.12 \ Dequeue once

In each program run, line 9 is executed with local n at most once for every node n.

Proof. Executing line 9 with local n dequeues n, which cannot be enqueued twice by proposition 14.7. ∎

vi Parents. The loop on line 10 iterates on all the parents of a node, recursively building their equivalence classes:

Proposition 14.13 \ Parent classes built

Let c, n be nodes, and let $\mathcal{S}$ be a state reachable after the execution of the loop on line 10 of BuildClass(c) with local variables c, n. Then for every parent m of n:

- $\mathsf{c}(\mathsf{m})$ is defined, say $\mathsf{c}(\mathsf{m}) = c'$;

- BuildClass(c') has been called and has already returned.

Proof. The loop on line 10 iterates on all parents of **n**. For every parent **m**:

- If **m** has no canonic assigned, BuildClass(**m**) is called (line 12). Since $\mathcal{S}$ is reached after the execution of the loop, the call to BuildClass(**m**) has already returned. Note that **m** was assigned itself as a canonic on line 5 of BuildClass(**m**), and $\mathbf{c}(\mathbf{m}) = \mathbf{m}$ still in $\mathcal{S}$ (proposition 14.1).
- If **m** has some c' assigned as a canonic node, then line 13 enforces that $\texttt{building}(c') = \textbf{false}$. This means that line 18 of BuildClass(c') has already been executed, and therefore BuildClass(c') has already returned. ∎

vii ∼neighbours. The loop on line 14 iterates on all the ∼neighbours of a node. Note that the loop does not remove ∼edges after iterating on them: in fact, the algorithm simply ignores ∼edges that have already been encountered, as it calls EnqueueAndPropagate only on ∼neighbours that have not been enqueued yet (line 16).

Proposition 14.14 \ ∼ Grows

During a program run, ∼ monotonically grows.

Proof. Just note that no line of the algorithm removes ∼edges. ∎

After the parent classes of a node are built, the set of its ∼neighbours is not going to change during the program run:

Proposition 14.15 \ Finalization of ∼neighbours

Let c, **n** be nodes. No ∼edge with endnode **n** can be created in any state $\mathcal{S}$ reached after the execution of the loop on line 10 with local variables c, **n**.

Proof. Assume by contradiction that $\mathcal{S}$ is a state reached after the execution of the loop on line 10 with locals c, **n**, and that the next transition creates a new ∼edge with endnode **n**. New ∼edges may be created only on line 21 and line 23 during the execution of EnqueueAndPropagate(**m**, c') for some nodes **m**, c'. Note that a ∼edge with endpoint **n** may be created by these lines only if **n** has either **m** or c' as a parent. We show that these cases are both not possible:

- **m** cannot be a parent of **n** since $\mathbf{c}(\mathbf{m})$ is undefined because of the check on line 16 (the only line that may call EnqueueAndPropagate), while by proposition 14.13 all parents of **n** have a canonic assigned.
- In order to show that c' cannot be a parent of **n**, note first that BuildClass(c') has not returned yet, since EnqueueAndPropagate(**m**, c') is called only on line 16 of

BuildClass(c'). Note also that $\mathsf{c}(c') = c'$ because line 5 of BuildClass(c') has already been executed. Therefore c' cannot be a parent of n by proposition 14.13. ∎

The following lemma proves that, after the $\sim$neighbours of a node n are handled by the loop on line 14, the canonic assignment subsumes the relation $\sim$ on n:

Proposition 14.16 \ All $\sim$neighbours visited

Let $\mathcal{S}$ be a state reachable after the execution of the loop on line 14 with local variables c, n. Then for every $\sim$edge with endnodes n and m, $\mathsf{n}\ \mathsf{c}^*\ \mathsf{m}$.

Proof. Let $\mathcal{S}'$ be the state prior to $\mathcal{S}$ in which execution is just entering the loop on line 14. Note that both $\mathcal{S}'$ and $\mathcal{S}$ are reached only after the execution of the loop on line 10 (with locals c, n), and therefore in $\mathcal{S}'$ and $\mathcal{S}$ are present the same $\sim$edges with endnode n (by propositions 14.14 and 14.15). The loop on line 14 iterates on all such $\sim$edges with endnodes n and m, and for each m it either:

- Line 16: calls EnqueueAndPropagate(m, c), which sets $\mathsf{c}(\mathsf{m}) := c$ before returning (Line 27);
- Line 17: explicitly enforces that $\mathsf{c}(\mathsf{m}) = c$.

In both cases, we obtain that $\mathsf{c}(\mathsf{m}) = c$ holds also in $\mathcal{S}$ (because $\mathcal{S}$ is reached after the execution of the loop on line 14, and the canonic assignment is immutable by proposition 14.1). Note also that $\mathsf{c}(\mathsf{n}) = c$ in $\mathcal{S}$ by proposition 14.9, since $\mathcal{S}$ is reached after the execution of line 9 which dequeues n from $\mathsf{q}(c)$. We thus obtain $\mathsf{c}(\mathsf{n}) = \mathsf{c}(\mathsf{m}) = c$ in $\mathcal{S}$, *i.e.* $\mathsf{n}\ \mathsf{c}^*\ \mathsf{m}$. ∎

14.1.2 Correctness

The main result of this section is theorem 14.26, proving that whenever algorithm 2 terminates successfully with final state $\mathcal{S}_f$ from an initial query $\mathcal{Q}$, then $\mathcal{Q}^\#$ is homogeneous, and therefore a pre-bisimulation. Additionally, the theorem shows that the canonic assignment is a succint representation of $\mathcal{Q}^\#$, *i.e.* that for all nodes n, m: $\mathsf{n}\ \mathcal{Q}^\#\ \mathsf{m}$ if and only if n and m have the same canonic assigned in $\mathcal{S}_f$.

Let us first introduce $=_\mathsf{c}$, the equivalence relation that equates two nodes whenever their canonic representatives are both defined and coincide:

Definition 14.17 \ Same canonic $=_\mathsf{c}$

In every reachable state $\mathcal{S}$ we define $=_\mathsf{c}$, a binary relation on the nodes of the λ-graph such that, for all nodes n,m: $\mathsf{n} =_\mathsf{c} \mathsf{m}$ iff $\mathsf{c}(\mathsf{n})$ and $\mathsf{c}(\mathsf{m})$ are both defined and $\mathsf{c}(\mathsf{n}) = \mathsf{c}(\mathsf{m})$.

The most important properties to prove correctness are the following:

- *Pre-bisimulation upto:* in all reachable states, c is a pre-bisimulation upto $(\sim \cup =)$.
- *Undirected query edges approximate the spreaded query:* in every reachable state, $\mathcal{Q} \subseteq \sim \subseteq \mathcal{Q}^{\#}$.
- *The canonic assignment respects the $\sim$ constraints:* in all reachable states, $\mathsf{c} \subseteq \sim^*$
- *Eventually, all query edges are visited:* in all final states, all query edges are subsumed by the canonic assignment, *i.e.* $\sim \subseteq \mathsf{c}^*$.

The lemmas above basically state that during the execution of the algorithm, $=_{\mathsf{c}}$ is a pre-bisimulation up to the query edges, and that $\sim$ can indeed be seen as an approximation of the spreaded query $\mathcal{Q}^{\#}$. At the end of the algorithm, $\sim$ and c actually represent the same[1] relation $=_{\mathsf{c}}$, which is then exactly the spreaded query. We now turn to prove the results that we have just outlined.

First of all, let us show that $=_{\mathsf{c}}$ is identical to c^* in all final states: this allows to simplify the following proofs, using the more familiar relation c instead of the new $=_{\mathsf{c}}$.

Proposition 14.18 $\setminus$ $=_{\mathsf{c}}$ *vs* c

Let $\mathcal{S}_f$ be a final state. Then for all nodes n, m: $\mathsf{n} =_{\mathsf{c}} \mathsf{m}$ if and only if $\mathsf{n}\ \mathsf{c}^*\ \mathsf{m}$.

Proof. First note that in $\mathcal{S}_f$ all nodes have a canonic assigned by proposition 14.3.

$(\Rightarrow)$ Let $\mathsf{n} =_{\mathsf{c}} \mathsf{m}$, and let us prove that $\mathsf{n}\ \mathsf{c}^*\ \mathsf{m}$. By the definition of $=_{\mathsf{c}}$, there exists c such that $\mathsf{c}(\mathsf{n}) = \mathsf{c}(\mathsf{m}) = c$. Since $\mathsf{n}\ \mathsf{c}\ c$ and $\mathsf{m}\ \mathsf{c}\ c$, then clearly $\mathsf{n}\ \mathsf{c}^*\ \mathsf{m}$ because c is functional.

$(\Leftarrow)$ Let $\mathsf{n}\ \mathsf{c}^*\ \mathsf{m}$, and let us prove that $\mathsf{n} =_{\mathsf{c}} \mathsf{m}$. We proceed by induction on the definition of c^*:

- Base case. Assume that $\mathsf{n}\ \mathsf{c}^*\ \mathsf{m}$ because $\mathsf{n}\ \mathsf{c}\ \mathsf{m}$, *i.e.* because $\mathsf{c}(\mathsf{n}) = \mathsf{m}$. By proposition 14.2, $\mathsf{c}(\mathsf{m}) = \mathsf{m}$, and therefore $\mathsf{n} =_{\mathsf{c}} \mathsf{m}$.
- Rule ●. Assume that $\mathsf{n}\ \mathsf{c}^*\ \mathsf{m}$ and $\mathsf{n} = \mathsf{m}$. From the hypothesis it follows that n has a canonic assigned, and therefore clearly $\mathsf{n} =_{\mathsf{c}} \mathsf{n}$.
- Rule ↔. Assume that $\mathsf{n}\ \mathsf{c}^*\ \mathsf{m}$ because $\mathsf{m}\ \mathsf{c}^*\ \mathsf{n}$. By *i.h.* $\mathsf{m} =_{\mathsf{c}} \mathsf{n}$, and we conclude because $=_{\mathsf{c}}$ is symmetric.
- Rule ↠. Assume that $\mathsf{n}\ \mathsf{c}^*\ \mathsf{m}$ because $\mathsf{n}\ \mathsf{c}^*\ \mathsf{p}$ and $\mathsf{p}\ \mathsf{c}^*\ \mathsf{m}$ for some node p. By *i.h.* $\mathsf{n} =_{\mathsf{c}} \mathsf{p}$ and $\mathsf{p} =_{\mathsf{c}} \mathsf{m}$, and we conclude because $=_{\mathsf{c}}$ is transitive. ∎

In order to prove that $=_{\mathsf{c}}$ equals $\mathcal{Q}^{\#}$ in $\mathcal{S}_f$ (proposition 14.25), we are going to prove that c^* is homogeneous and propagated (proposition 14.24). The following proposition is a relaxed variant of that statement which holds for all reachable states: it collapses to the desired statement in the final state because $(\sim \cup =) = (\mathsf{c}^*)$.

[1] Up to equivalence closure.

Proposition 14.19 \ c is upto

Let $\mathcal{S}$ be a reachable state. Then c is a pre-bisimulation upto $(\sim \cup =)$.

Proof. We prove the statement by induction on the length of the program run leading to $\mathcal{S}$:

- In the initial state $\mathsf{c} = \emptyset$, and therefore the statement holds trivially.
- As for the inductive step, we only need to discuss the program transitions that alter $\sim$ and c. But first of all, note that $\sim$ can only grow (proposition 14.14), and that creating new $\sim$edges (while keeping c unchanged) does not falsify the statement if it was true before the addition. Therefore we actually need to discuss only the transitions that alter the canonical assignment, *i.e.* line 5 and line 27:
  - line 5: by *i.h.*, c was homogeneous and propagated upto $(\sim \cup =)$ before the execution of the assignment $\mathsf{c}(c) = c$. After the execution of that assignment, c differs only for the new entry (c, c). Clearly the new entry satisfies the homogeneous condition, and it satisfies the property of being propagated upto $=$.
  - line 27, during the execution of EnqueueAndPropagate(m, c): by *i.h.*, c was homogeneous and propagated upto $(\sim \cup =)$ before the execution of the assignment $\mathsf{c}(\mathsf{m}) := c$. After the execution of that assignment, c differs only for the new entry (m, c). The new entry satisfies the homogeneous condition and the property of being propagated upto $\sim$ because of the code executed in Lines 20–26, which checked the labels of the nodes and created $\sim$edges on the corresponding children of m, c if any.

The following proposition 14.20 and proposition 14.21 state properties that connect the relations c, $\mathcal{Q}$, and $\sim$. In a final state c^* is exactly $\sim^*$, but in intermediate states the following weaker property holds:

Proposition 14.20 \ c is sound

In every reachable state $\mathcal{S}$: $\mathsf{c} \subseteq \sim^*$.

Proof. We prove the statement by induction on the length of the program run leading to $\mathcal{S}$:

- In the initial state, $\mathsf{c} = \emptyset$ and therefore the statement holds trivially.
- As for the inductive step, we only need to discuss the program transitions that alter $\sim$ and c. But first of all, note that during the execution of the algorithm $\sim$ can only grow (proposition 14.14), and that creating new $\sim$edges (while keeping c unchanged) does not falsify the statement if it was true before the addition. Therefore we actually need to discuss only the transitions that alter the canonical assignment, *i.e.*

line 5 and line 27:

- line 5: by *i.h.*, $\mathsf{c} \subseteq \sim^*$ before the execution of the assignment $\mathsf{c}(c) = c$. After the execution of that assignment, c differs only for the new entry (c, c). Clearly $c \sim^* c$ because $\sim^*$ is reflexive by definition.
- line 27, during the execution of EnqueueAndPropagate(m, c): by *i.h.*, $\mathsf{c} \subseteq \sim^*$ before the execution of the assignment $\mathsf{c}(\mathsf{m}) := c$. After the execution of that assignment, c differs only for the new entry (m, c). Note that EnqueueAndPropagate(m, c) is called from line 16 during the execution of BuildClass(c). Because of the loop on line 14, m was a $\sim$neighbour of n for some node n, and it still is because $\sim$ can only grow (proposition 14.14). We are now going to prove that $\mathsf{n} \sim^* c$, which together with the fact that m is a $\sim$neighbour of n, will allow us to conclude with $\mathsf{m} \sim^* c$.

  line 16 is executed after line 9, therefore $\mathsf{c}(\mathsf{n}) = c$ (*i.e.* $\mathsf{n}\ \mathsf{c}\ c$) by proposition 14.9. By *i.h.* $\mathsf{n}\ \mathsf{c}\ c$ implies $\mathsf{n} \sim^* c$, and we are done. ∎

During a program run, the relation $\sim$ progressively approximates the relation $\mathcal{Q}^\#$:

Proposition 14.21 \ $\sim$ approximates $\mathcal{Q}^\#$

In every reachable state $\mathcal{S}$: $\mathcal{Q} \subseteq \sim \subseteq \mathcal{Q}^\#$.

Proof. We prove the statement by induction on the length of the execution trace leading to $\mathcal{S}$:

- Base case. In the initial state $(\sim) = \mathcal{Q}$ and the statement clearly holds.
- Inductive step. First note that during the execution of the algorithm $\sim$ can only grow (proposition 14.14), therefore the requirement $\mathcal{Q} \subseteq \sim$ follows from the *i.h.* In order to prove that $\sim \subseteq \mathcal{Q}^\#$, we only need to discuss the program transitions that alter $\sim$, *i.e.* those occurring on line 21 and line 23:
  - line 21 during the execution of EnqueueAndPropagate(m, c). Because of the check on the same line, $\mathsf{m} = \mathtt{Abs}(\mathsf{m}')$ and $c = \mathtt{Abs}(c')$ for some m', c'. Executing line 21 creates the new edge $\mathsf{m}' \sim c'$. Since $\mathcal{Q}^\#$ is propagated, in order to show the new requirement $\mathsf{m}'\ \mathcal{Q}^\#\ c'$ it suffices to show that $\mathsf{m}\ \mathcal{Q}^\#\ c$. We are going to show that $\mathsf{m} \sim^* c$ in the state prior to $\mathcal{S}$: then by using the *i.h.* we obtain $\mathsf{m}\ \mathcal{Q}^\#\ c$, and we can conclude.

    Note that EnqueueAndPropagate(m, c) is called from line 16 during the execution of BuildClass(c). Because of the loop on line 14, m was a $\sim$neighbour of n for some node n, and it still is because $\sim$ can only grow (proposition 14.14). In order to prove $\mathsf{m} \sim^* c$, it thus suffices to prove $\mathsf{n} \sim^* c$. The latter follows from proposition 14.20, by using the fact that line 16 is executed after line 9, therefore $\mathsf{c}(\mathsf{n}) = c$

– line 23 is similar to the previous case. ∎

Corollary 14.22

In every reachable state, $\mathsf{c}^* \subseteq \mathcal{Q}^\#$.

Proposition 14.23 \ All $\sim$edges visited

In every final state, $(\sim) \subseteq (\mathsf{c}^*)$.

Proof. Let $\mathcal{S}_f$ be a final state. By proposition 14.3, in $\mathcal{S}_f$ every node has a canonic assigned. Let n be a node, and $c \coloneqq \mathsf{c}(\mathsf{n})$; we are going to prove that $\mathsf{n}\ \mathsf{c}^*\ \mathsf{m}$ for every $\sim$neighbour m of n.
By proposition 14.5, BuildClass(c) was called before $\mathcal{S}_f$. Since $\mathcal{S}_f$ is final, BuildClass(c) must have returned, and therefore n has already been processed (proposition 14.11). After the loop on line 10, the $\sim$neighbours of n are finalized (proposition 14.15), and after the loop on line 14, the $\sim$neighbours of n are all visited (proposition 14.16). ∎

The following two propositions lead directly to theorem 14.26, proving correctness.

Proposition 14.24

Let $\mathcal{S}_f$ a final state. Then c^* is a pre-bisimulation.

Proof. By proposition 14.19, c is homogeneous and propagated up to $(\sim \cup =)$. We need to show that c^* is homogeneous and propagated.

- *Homogeneous.* It follows from proposition 12.18.
- *Propagated.* By proposition 14.23 and proposition 14.18, $(\sim) \subseteq (\mathsf{c}^*)$, which implies that $(\sim \cup =) \subseteq (\mathsf{c}^*)$ because c^* is reflexive. Therefore c is propagated up to c^* by proposition 12.33, and c^* is propagated by proposition 12.34. ∎

Proposition 14.25

Let $\mathcal{S}_f$ a final state. Then $(\mathsf{c}^*) = (\mathcal{Q}^\#)$.

Proof. By corollary 14.22, $(\mathsf{c}^*) \subseteq (\mathcal{Q}^\#)$. In order to prove $(\mathcal{Q}^\#) \subseteq (\mathsf{c}^*)$, it suffices to note that $\mathcal{Q} \subseteq (\mathsf{c}^*)$ (because $\mathcal{Q} \subseteq (\sim)$ by proposition 14.21, and $(\sim) \subseteq (\mathsf{c}^*)$ by proposition 14.23), that c^* is an equivalence relation (by definition), and that c^* is propagated (proposition 14.24). ∎

Theorem 14.26 \ Correctness

In every final state $\mathcal{S}_f$ of algorithm 2:

- *Succint representation:* $(=_c) = (\mathcal{Q}^\#)$,
- *Blind check:* $\mathcal{Q}^\#$ is homogeneous, and therefore a pre-bisimulation.

Proof. By proposition 14.24 and proposition 14.25. ∎

14.1.3 Completeness

In this section we prove completeness (theorem 14.31), *i.e.* that whenever algorithm 2 fails, $\mathcal{Q}^\#$ is not a pre-bisimulation. Recall that the algorithm can fail only while executing the following three lines of code:

- On line 13 during BuildClass, when a node n is being processed and it has a parent whose equivalence class is still being built;
- On line 17 during BuildClass, when a node n is being processed and it has a $\sim$neighbour belonging to a difference equivalence class;
- On line 26 during EnqueueAndPropagate(m, c), when the algorithm is trying two relate the nodes m and c which are not homogeneous.

While in the latter case (line 26) the failure of the homogeneous condition is more evident, in the first two cases it is more subtle. In fact, when the homogeneous condition fails on line 13 and line 17 it is not because we explicitly found two related nodes that are not homogeneous, but because $\mathcal{Q}^\#$ does not satisfy an indirect property that is necessary for $\mathcal{Q}^\#$ to be homogenous (see below). In these cases the algorithm fails early, even though it has not visited yet the actual pair of nodes that are not homogeneous.

To justify the early failures we use `active`, which basically records the equivalence classes that the algorithm is building in a given state, sorted according to the order of calls to BuildClass. Nodes in `active` respect a certain strict order: if `active` $= [c_1, \dots, c_K]$, then $c_1 \prec c_2 \prec \dots \prec c_K$ (proposition 14.28), where $\mathsf{n} \prec \mathsf{m}$ implies that the equivalence class of n is a child of the one of m in the quotient graph. The algorithm fails on line 13 and line 17 because it found a cyclic chain of nodes related by $\prec$, basically finding a cycle in the quotient graph. By proposition 12.13, there cannot exist any pre-bisimulation containing the initial query in this case.

First of all, let us define the staircase order $\prec$ mentioned above:

Definition 14.27 \ Staircase order $\prec$

Let $\mathcal{S}$ be a reachable state, and n, m be nodes of the λ-graph G. We say that $\mathsf{n} \prec \mathsf{m}$ iff there exist a direction d and a node p such that $d\colon \mathsf{m} \rightsquigarrow \mathsf{p}$ (riser) and $\mathsf{p}\ \mathsf{c}\ \mathsf{n}$ (tread).

In every reachable state, the canonic representatives in `active` respect the staircase order $\prec$:

Proposition 14.28 \ Order of active classes

Let $\mathcal{S}$ be a reachable state such that $\texttt{active} = [c_1, \dots, c_K]$. Then $c_i \prec c_{i+1}$ for every $0 < i < K$.

Proof. We proceed by induction on the length of the program run. In the base case, *i.e.* in the initial state, the callstack is empty and therefore the property holds trivially.
As for the inductive step, we only need to discuss the program transitions that actually alter the callstack, *i.e.* when BuildClass is called or when it returns:

- BuildClass returns. In this case, the last entry of the callstack is removed, and the statement follows from the *i.h.*
- BuildClass(n) is called (line 3). Right before the call to BuildClass(n) the callstack is empty, and right after the call $\texttt{active} = [\mathsf{n}]$, *i.e.* there is nothing to prove.
- BuildClass(m) is called (line 12). In this case m is a parent of n (*i.e.* there exists a direction d such that $d\colon \mathsf{m} \rightsquigarrow \mathsf{n}$), and $c = c_K$ before the call to BuildClass(m). Note that line 12 is executed after line 9 with locals c, n, and therefore $\mathsf{c}(\mathsf{n}) = c$ (proposition 14.9).

  After the call to BuildClass(m), $\texttt{active} = [c_1, \dots, c_K, \mathsf{m}]$. The statement for $i < K$ follows from the *i.h.*, and for the case $i = K$ we just proved that $d\colon \mathsf{m} \rightsquigarrow \mathsf{n}$ and $\mathsf{n}\ \mathsf{c}\ c_K$.

A useful intuition is that in a program run, recursive calls to BuildClass climb a stair of nodes in the λ-graph. The algorithm fails when it encounters a step that was already climbed: this is because the staircase order is strict when $\mathcal{Q}^\#$ is homogeneous and propagated.

Proposition 14.29 \ $\mathcal{Q}^\#$ respects staircase order

Let $\mathcal{S}$ be a reachable state, and let $c_1, \dots, c_K$ be nodes such that $c_1 \prec \dots \prec c_K$. If $\mathcal{Q}^\#$ is homogeneous, then $c_i\ \mathcal{Q}^\#\ c_j$ if and only if $i = j$.

Proof. If $i = j$, then clearly $c_i\ \mathcal{Q}^\#\ c_j$ because $\mathcal{Q}^\#$ is reflexive.
For the other direction, assume by contradiction and without loss of generality that $i < j$ (since $\mathcal{Q}^\#$ is symmetric). We are going to prove that there exist a non-empty trace τ and a node m such that $\tau\colon c_j \rightsquigarrow \mathsf{m}$ and $c_i\ \mathcal{Q}^\#\ \mathsf{m}$: this, together with $c_i\ \mathcal{Q}^\#\ c_j$, will yield a contradiction by proposition 12.13.
We proceed by induction on $j - i - 1$:

- Case $i + 1 = j$. Obtain from proposition 14.28 a node n and a direction d such that $c_{i+1} = d\colon c_j \rightsquigarrow \mathsf{n}$ and $\mathsf{n}\ \mathsf{c}\ c_i$, and conclude because $c_i\ \mathcal{Q}^\#\ \mathsf{m}$ by corollary 14.22.
- Case $i + 1 < j$. By *i.h.* there exist a non-empty trace τ and a node n such that

$\tau\colon c_j \rightsquigarrow \mathsf{n}$ and $c_{i+1}\ \mathcal{Q}^{\#}\ \mathsf{n}$. By proposition 14.28, there exist a direction d and a node m such that $d\colon c_{i+1} \rightsquigarrow \mathsf{m}$ and $\mathsf{m}\ \mathsf{c}\ c_i$ (and therefore $c_i\ \mathcal{Q}^{\#}\ \mathsf{m}$ by corollary 14.22). From $c_{i+1}\ \mathcal{Q}^{\#}\ \mathsf{n}$ and $d\colon c_{i+1} \rightsquigarrow \mathsf{m}$, by proposition 12.9 there exists a node m' such that $d\colon \mathsf{n} \rightsquigarrow \mathsf{m}'$ and $\mathsf{m}\ \mathcal{Q}^{\#}\ \mathsf{m}'$. We can conclude, since $\tau \cdot d\colon c_j \rightsquigarrow \mathsf{m}'$ and $c_i\ \mathcal{Q}^{\#}\ \mathsf{m}'$.

We are now ready to prove that the algorithm always fails correctly:

Proposition 14.30 \ Failure is correct

If the algorithm fails, then $\mathcal{Q}^{\#}$ is not homogeneous.

Proof. Let $\mathcal{S}$ be a failing state, *i.e.* a reachable state that transitions to $\mathcal{F}ail$. Let $\texttt{active} = [c_1, \dots, c_K]$ in $\mathcal{S}$. There are three lines of the algorithm in which failure may occur: line 13, line 17, and line 26. We discuss these cases separately; in all cases, in order to prove that $\mathcal{Q}^{\#}$ is not homogeneous, we assume $\mathcal{Q}^{\#}$ to be homogeneous, and derive a contradiction.

- line 13. Let m be a parent of n such that $\mathsf{c}(\mathsf{m}) = c'$ and $\texttt{building}(c') = \mathsf{true}$. Since $\texttt{building}(c') = \mathsf{true}$, the call to BuildClass(c') has not returned before $\mathcal{S}$, and therefore $c' = c_i$ for some $0 < i \leq K$. By proposition 14.28 $c_1 \prec \dots \prec c_K$, and therefore c' was already encountered in the stair of active classes. Note that $\mathsf{c}(\mathsf{n}) = c_K = c$ (proposition 14.9), and therefore $c_K \prec \mathsf{m}$. Also, from $\mathsf{c}(\mathsf{m}) = c_i$ it follows that $\mathsf{m}\ \mathcal{Q}^{\#}\ c_i$ (corollary 14.22). The contradiction is obtained by noting that by proposition 14.29 it cannot be the case that $\mathsf{m}\ \mathcal{Q}^{\#}\ c_i$, since $c_1 \prec \dots \prec c_K \prec \mathsf{m}$.
- line 17. Let m be a $\sim$neighbour of n such that $\mathsf{c}(\mathsf{m}) = c' \neq c$. By proposition 14.9, $\mathsf{c}(\mathsf{n}) = c$. Since $\mathsf{c}(\mathsf{m}) = c'$, BuildClass($c'$) must have been called (proposition 14.5), and there are two options:
  - BuildClass(c') has already returned. We show that this is not possible. In fact, if BuildClass(c') has returned before $\mathcal{S}$, then m has already been processed before $\mathcal{S}$ (proposition 14.11). Therefore $\mathsf{n} =_c \mathsf{m}$ (proposition 14.16), contradicting the hypothesis that $c \neq c'$.
  - BuildClass(c') has not yet returned. Therefore $c' = c_i$ for some $0 < i < K$. From $\mathsf{c}(\mathsf{n}) = c = c_K$ and $\mathsf{c}(\mathsf{m}) = c_i$ obtain $\mathsf{n} \sim^* c_K$ and $\mathsf{m} \sim^* c_i$ (proposition 14.20). Since m is a $\sim$neighbour of n, $c_i \sim^* c_K$, and therefore $c_i\ \mathcal{Q}^{\#}\ c_K$ by proposition 14.21. This yields a contradiction by proposition 14.28 and proposition 14.29.
- line 26. Assume that c and m do not respect the homogenity condition. We are going to prove that $c\ \mathcal{Q}^{\#}\ \mathsf{m}$, which contradicts the hypothesis that $\mathcal{Q}^{\#}$ is homogeneous. EnqueueAndPropagate(m, c) is called from line 16, and therefore m is a $\sim$neighbour of n. line 16 is executed after line 9 with locals c, n, and therefore

$\mathsf{c}(\mathsf{n}) = c$ by proposition 14.9. By proposition 14.20, $c \sim^* \mathsf{n}$, and therefore $c \sim^* \mathsf{m}$. By proposition 14.21, we conclude with $c \; \mathcal{Q}^{\#} \; \mathsf{m}$.

As an easy consequence, algorithm 2 is complete:

Theorem 14.31 \ Completeness

If algorithm 2 fails, then $\mathcal{Q}^{\#}$ is not a pre-bisimulation, and therefore by the universality of $\mathcal{Q}^{\#}$, there does not exist any pre-bisimulation nor sharing equivalence containing the initial query $\mathcal{Q}$.

Proof. By theorem 12.16 and proposition 14.30.

14.1.4 Linearity

In this section we show that algorithm 2 always terminates, and it does so in time linear in the size of the λ-graph and the initial query.

Low-level assumptions. In order to analyse the complexity of the algorithm we have to spell out some details about an hypothetical implementation on a RAM of the data structures used by the Blind check.

- *Nodes*: since line 2 of the algorithm needs to iterate on all nodes of the λ-graph, we assume an array of pointers to all the nodes of the graph.
- *Directed edges*: these edges—despite being directed—have to be traversed in both directions, for instance to recurse over the parents of a node. The λ-graphs used for efficient reduction as defined in chapter 6 miss these backward pointers. We then assume that every node has an additional array of pointers to its parents, which can anyway be constructed in linear time (Champeaux, 1986).
- *Undirected query edges*: query edges are undirected and are dynamically created; in addition, the algorithm needs to iterate on all $\sim$neighbours of a node. In order to obtain the right complexity, every node simply maintains a linked list of its $\sim$neighbours, in such a way that when a new undirected edge (n, m) is created, then n is pushed on the list of $\sim$neighbours of m, and m on the one of n.
- *Canonical assignment* is obtained by a pointer to a node (possibly undefined) in the data structure for nodes.
- *Building flags* are just implemented via a boolean on each node.
- *Queues* do not need to be recorded in the data structure for nodes, as they can be equivalently coded as local variables.

Let us call *atomic* the following operations performed by the check: finding the first node, finding the next node given the previous node, finding the first parent of a node, finding the next parent of a node given the previous parent, checking and setting canonics, checking and setting building flags, getting the next query edge on a given node, traversing a query edge, adding a query edge between two nodes, pushing to a queue, and popping an element off of a queue.

Proposition 14.32 \ Atomic operations are constant

The atomic operations of the Blind check are all implementable in constant time on a RAM.

We prove termination and linearity of algorithm 2 via a global estimation of the number of transitions in a program run. The difficult part is to estimate the number of transitions executing lines of BuildClass, since it contains multiple nested loops. First of all, we note that in every program run BuildClass is called at most once for each node. Also the body of the while loop on line 8 is executed in total at most once for each node, because in every program run each node is enqueued at most once. The loop on line 10 is not problematic because the parents of a node do not change during a program run. Estimating the number of iterations of the loop on line 14, which iterates over the $\sim$neighbours of a node n, seems much more involved because the code inside the loop may create new query edges; however as already discussed the $\sim$neighbours of n do not change after the parents of n are visited on line 10. The last insight for linearity is to recall that the algorithm parsimoniously propagates query edges only between a node and its canonic: as a consequence, in every reachable state, $|\mathtt{undir}|$ is linear in the size of $\mathcal{Q}$ and the number of nodes in the λ-graph:

Proposition 14.33 \ Bound on `undir`

Let $\mathcal{Q}$ be a query over a λ-graph G as in input to algorithm 2. In every reachable state $\mathcal{S}$, $|\mathtt{undir}| \leq 2 \times |\mathcal{Q}| + 4 \times |N|$ (where $|N|$ is the number of nodes of G).

Proof. Proved by induction on the length of the program run leading to $\mathcal{S}$. In the initial state $\mathcal{S}_0$, $|\mathtt{undir}| = 2 \times |\mathcal{Q}|$ because `undir` contains the same edges as $\mathcal{Q}$ (but `undir` is also symmetric and a multirelation, hence each $\sim$edge counts twice).
During a program run, new $\sim$edges are created only by the EnqueueAndPropagate procedure, which may create at most two new $\sim$edges per call (that count as four new entries of the symmetric multirelation `undir`). By proposition 14.6, EnqueueAndPropagate is called at most $|N|$ times in each program run, hence we conclude with the bound $|\mathtt{undir}| \leq 2 \times |\mathcal{Q}| + 4 \times |N|$. ∎

Let $|G|$ denote the size of a λ-graph G, *i.e.* the number of nodes $|N|$ plus the number of (directed) edges $|E|$ of G. Let also $\#edges(\sim, \mathsf{n})$ be the number of $\sim$edges with endnode n[2],and $\#parents(\mathsf{n})$ be the number of directed edges in E with target n. Then:

[2]Recall that $\sim$ is a multirelation.

Proposition 14.34 \ Linear-time termination

Let $\mathcal{S}_0$ be an initial state of the algorithm, with a query $\mathcal{Q}$ over a λ-graph G. Then the Blind check terminates in a number of transitions linear in $|G|$ and $|\mathcal{Q}|$.

Proof. Let us assume a program run with end state $\mathcal{S}$ (not necessarily a final state). We are going to discuss the number of transitions in the program run in a global way. We group the transitions in the program run according to which procedure they are executing a line of. The procedure EnqueueAndPropagate contains no loops and no function calls; therefore, there is a constant bound on the number of transitions that are executed by this procedure each time it is called. We consider this procedure as if it were inlined in BuildClass; we instead discuss separately the procedures BlindCheck and BuildClass.

1. Transitions executing lines of BlindCheck. The procedure BlindCheck is called exactly once, in the initial state. The loop on line 2 simply iterates over all the nodes of the λ-graph: hence the loop bound is just $|N|$. Line 3 may call the procedure BuildClass, but here we do not consider the transitions caused by that function, that will be discussed separately below. Therefore, the total number of transitions that execute lines of BlindCheck is simply $O(|N|)$.

2. Transitions executing lines of BuildClass. The discussion of the complexity of BuildClass is more involved, because it contains multiple nested loops.

   As a first approximation, the total number of transitions executing lines of BuildClass is big-O of the number of calls to BuildClass plus the numer of transitions executing lines of the loop on line 8 (*i.e.* Lines 8–17). The number of calls to BuildClass is $\leq |N|$ (proposition 14.4). As for the loop on line 8, in every program run the body of the loop is executed at most once for every node n (proposition 14.12).

   The body of the while loop contains two additional loops: one on line 10, and the other on line 14:

   - The loop on line 10 (with locals c, n) simply iterates on all the parents of a node, which is a fixed amount: therefore each time it is executed with local variable n, it is responsible for $O(\#parents(\mathsf{n}))$ transitions.
   - The discussion about the loop on line 14 (with locals c, n) is more involved: it iterates on all $\sim$neighbours of a node n, but $\sim$ changes during the execution of the loop. In fact, the body of the loop may call EnqueueAndPropagate(m,c) on line 16, which may create new $\sim$edges. However, note that the loop on line 14 is executed only after the loop on line 10, which finalizes the $\sim$neighbours of n (proposition 14.15). Therefore the loop on line 14 iterates on a set of $\sim$edges that does not change afterwards during the program run, and is therefore responsible for $O(\#edges(\sim, \mathsf{n}))$ transitions (where the relation $\sim$ is the one in $\mathcal{S}$).

Therefore the total number of transitions executing Lines 8–17 is big-O of:

$$\begin{aligned}
&\sum\{\#parents(\mathsf{n}) + \#edges(\sim, \mathsf{n}) + 1 \mid \text{Line 9 is executed with local } \mathsf{n}\} \\
\leq &\sum_{\mathsf{n}\in N} (\#parents(\mathsf{n}) + \#edges(\sim, \mathsf{n}) + 1) \\
= &\sum_{\mathsf{n}\in N} \#parents(\mathsf{n}) + \sum_{\mathsf{n}\in N} \#edges(\sim, \mathsf{n}) + \sum_{\mathsf{n}\in N} 1 \\
= &\ |E| + |\texttt{undir}| + |N|.
\end{aligned}$$

Note that $|\texttt{undir}| \in O(|\mathcal{Q}|+|N|)$ by proposition 14.33. Therefore the total number of transitions executing lines of BuildClass in a program run is $O(|N|+|E|+|\mathcal{Q}|)$.

In conclusion, we obtain the following asymptotic bound on the number of transitions in a program run:

$$O(|N|) + O(|N|) + O(|N| + |E| + |\mathcal{Q}|) = O(|N| + |E| + |\mathcal{Q}|).$$

14.2 The Variables Check

Our second algorithm (algorithm 3) takes in input the output of the Blind check, that is, a pre-bisimulation on a λ-graph G represented via canonic edges, and checks whether the variable nodes of G satisfy the variables conditions for a sharing equivalence—free variable nodes on line 4, and bound variable nodes on line 5.

The Variables check is based on the fact that to compare a node with all the other nodes in its equivalence class it is enough to compare it with the canonical representative of the class: note that this fact is used twice, for the variable nodes and for their binders (line 4). The check fails in two cases:

- When two distinct free variable nodes are related: in this case $\mathcal{Q}^{\#}$ is not an open relation.
- When two related bound variable nodes have binders that are not related: in this case $\mathcal{Q}^{\#}$ is not closed under  .

Theorem 14.35 \ Correctness & completeness of the Variables check

Let $\mathcal{Q}$ be a query over a λ-graph G passing the Blind check, and let c be the canonic assignment produced by that check.

- *Completeness:* if the Variables check fails then there exists no sharing equivalence containing $\mathcal{Q}$.

Algorithm 3 \ Variables Check

Data: $\texttt{canonic}(\cdot)$ representation of $\mathcal{Q}^{\#}$
Result: is $\mathcal{Q}^{\#}$ a sharing equivalence?

```
1 Procedure VarsCheck()
2     foreach variable node n do
3         case n, canonic(n) of
4             bVar(l), bVar(l′) ⇒ assert canonic(l) = canonic(l′)
5             fVar(x), fVar(y) ⇒ assert x = y
6             _, _ ⇒ // impossible
```

- *Correctness:* otherwise, $=_{\mathsf{c}}$ is the smallest sharing equivalence containing $\mathcal{Q}$.

Moreover, the Variables check terminates in time linear in the size of G.

Proof. First note that since the Blind check succeeded, both c and $=_{\mathsf{c}}$ are homogeneous relations. We also use the fact that $(=_{\mathsf{c}}) = (\mathsf{c}^*)$ (proposition 14.18).

- *Completeness:* If the Variables check fails on line 5, then c is not an open relation, and therefore also $=_{\mathsf{c}}$ is not open by proposition 12.19. If the Variables check fails on line 4, then c is not a bisimulation upto $=_{\mathsf{c}}$, and therefore $=_{\mathsf{c}}$ is not a bisimulation (because $\mathsf{c} \subseteq =_{\mathsf{c}}$), and therefore it is not a sharing equivalence.
- *Correctness:* Since also the Variables Check succeeded, c is an open relation, and closed under  upto $=_{\mathsf{c}}$. In order to show that $=_{\mathsf{c}}$ is a sharing equivalence, it suffices to show that it is open, and closed under  . $=_{\mathsf{c}}$ is closed under  by proposition 12.20. $=_{\mathsf{c}}$ is open by proposition 12.19.
- *Termination in linear time:* obvious, since the algorithm simply iterates on all variable nodes of the λ-graph. ∎

Finally, composing with the sharing equality theorem (corollary 12.30) one obtains that the two provided algorithms indeed test the equality of the readbacks of the query, as expected:

Theorem 14.36 \ Sharing equality is linear

Let $\mathcal{Q}$ be a query over a λ-graph G. The sharing equality algorithm obtained by combining algorithm 2 and algorithm 3 runs in time proportional to the sizes of G and $\mathcal{Q}$, and it succeeds if and only if there exists a sharing equivalence containing $\mathcal{Q}$. Moreover, if it succeeds, it outputs a concrete[3] representation of the smallest such sharing equivalence.

Proof. By theorems 14.26, 14.31 and 14.35 and proposition 14.34. ∎

[3] And linear (in space).

14.3 Combined Algorithm

As a final remark, algorithms 2 and 3 can be combined back into a single algorithm. To do so, one can simply inline the check on variables performed by algorithm 3 directly in the body of EnqueueAndPropagate in algorithm 2. This results in algorithm 4.

Algorithm 4 \ Sharing Equality Algorithm

Data: an initial state
Result: $\mathcal{F}ail$ or a final state

```
1  Procedure SharingEqualityCheck()
2      for every node n do
3          if canonic(n) undefined then BuildClass(n)
4  Procedure BuildClass(c)
5      canonic(c) := c
6      building(c) := true
7      queue(c) := {c}
8      while queue(c) is non-empty do
9          n := queue(c).pop()
10         for every parent m of n do
11             case canonic(m) of
12                 undefined ⇒ BuildClass(m)
13                 c' ⇒ if building(c') then FAIL
14         for every ∼neighbour m of n do
15             case canonic(m) of
16                 undefined ⇒ EnqueueAndPropagate(m, c)
17                 c' ⇒ if c' ≠ c then FAIL
18     building(c) := false
19 Procedure EnqueueAndPropagate(m, c)
20     case m, c of
21         Abs(m'), Abs(c') ⇒ create edge m' ∼ c'
22         App(m1, m2), App(c1, c2) ⇒
23             create edges m1 ∼ c1 and m2 ∼ c2
24         bVar(l), bVar(l') ⇒ assert canonic(l) = canonic(l')
25         fVar(x), fVar(y) ⇒ assert x = y
26         _, _ ⇒ FAIL
27     canonic(m) := c
28     queue(c).push(m)
```

Part IV

Conclusions

Recap

In this dissertation, we investigated how to improve the efficiency of β-conversion in the pure λ-calculus. To do so, we decomposed Conversion in two subproblems:

1. Computation (part II). This problem is solved for the Call-by-Value λ-calculus in the open case: the Crumble GLAMs that we present in part II run in time linear in the size of the term to be evaluated and the number of CbV β-steps. The strong case required by conversion can be obtained by iterating the Crumble GLAMs by levels, but we have ongoing research into generalizing our crumbling machines directly to the strong case, which is harder and still an open problem: in the next chapter, we outline our abstract machine for Strong CbV evaluation, which — we believe — also runs in bilinear time (of course, with respect to the number of Strong CbV β-steps, see section 15.1). That machine is much more involved than the Crumble GLAMs, requiring stronger invariants; it also relies heavily on crumbling, and its proofs crucially rest upon the technical tools that we have developed in part II.
2. Comparison (part III). We solved this problem by providing an efficient algorithm for sharing equality, the first one running in time linear in the size of the shared terms to be compared. Our algorithm is based on Paterson-Wegman's, but we also provide an abstract and clean study of the theory of sharing equality, independent of the chosen algorithm.

There is plenty of future work, also from the more practical or implementative points of view. Like many before us, we tackled conversion with the ultimate goal of improving the performance of the Coq proof assistant: therefore a necessary step is extending our results to Coq. As we discuss in section 15.2, however, Coq poses multiple challeges. First, conversion in Coq is a more complex *subtyping* relation, and the complexity of Comparison in the presence of subtyping is unknown. Second, as the Coq library is quite massive, bounding the complexity of our abstract machines through the size of the initial term provides impractical upper bounds, because the initial term usually depends on various Coq libraries which cause the size of inital term to become as big as the whole library.

We conclude this dissertation with an outline for future research (chapter 15).

Chapter 15

Future Research

15.1 Strong CbV Evaluation

In collaboration with Sacerdoti Coen and Accattoli, we have preliminary results on how to generalize the Crumble GLAMs presented in part II to the more difficult case of strong evaluation.

In this section we sketch our machine for Strong CbV, which we call SCAM for *Strong Crumbling Abstract Machine*. The SCAM dates back to 2018, but at the time the technical background of crumbled forms was simply not developed enough to allow the clear formulation of the machine invariants that we present in the following pages. The reader will note that all proofs are elided: in fact, we leave the full development with complete proofs for a future publication. Still, we decided to include here our preliminary results about the SCAM because the theory on crumbled forms developed in part II makes us confident that they are indeed correct.

Calculus. When looking for a machine to implement Strong CbV, clearly the first step is to design a proper Strong CbV calculus. Unfortunately the fireball calculus λ_{fire} that we have introduced in section 3.2 is not ready-for-use to perform strong reduction. In fact, naïvely allowing fireball reductions under abstractions makes the calculus undesirably not confluent (Accattoli and Guerrieri, 2016), as the following critical pair cannot be joined:

$$I \;{}_{\beta/v}\!\leftarrow (\lambda y.I)\,\delta \;{}_{\beta/i}\!\leftarrow (\lambda x.(\lambda y.I)\,(xx))\,\delta \rightarrow_{\beta/v} (\lambda y.I)\,(\delta\delta) \rightarrow_{\beta/v} \cdots$$

(Note that the term on the far left is the identity, while the one on the far right diverges.)

Another point against λ_{fire} comes from semantics. In fact, two of the properties of any well-behaved denotational semantics are *invariance under evaluation*, *i.e.* that denotations are stable under evaluation, and *compositionality*, *i.e.* that terms with the same denotation when plugged in a same context still have the same denotation. The example above also shows that no denotational semantics for λ_{fire} can have these properties: the term $(\lambda y.I)\,(xx)$ reduces by $\rightarrow_{\beta/i}$ to I, but in the context $(\lambda x.\langle\cdot\rangle)\delta$ the first diverges and the second converges, hence they should not have the same denotation.

Denotational studies like the one by Accattoli and Guerrieri, 2018 show that the reduction rule $\to_{\beta/i}$ (the one substituting inert terms) is the one to blame. The problem is not only the erasure of an inert term, like the inert xx in the example above. Their semantics via *multi types* (that also provides quantitative analyses) forbids also the duplication caused by a β/i step.

The solution by Accattoli and Guerrieri, 2018 is to modify the operational semantics of λ_{fire} in such a way that a β/i step consumes the redex without actually substituting the inert argument: they accomplish this by using a λ-calculus with ES where a term is accompanied by a sort of environment made up of inert ES, whose only role is to provide compositionality to denotations. Clearly, this "inert garbage" is motivated by denotational semantics but not so by implementations: for instance, since the λ-terms of proof assistants are all strongly-normalizing, one may get rid of the inert garbage without impacting on termination and compositionality. However, we rather prefer a machine that implements a well-behaved Strong CbV calculus. One can always simplify later the machine, adapting it to the strongly-normalizing setting by garbage-collecting the "inert garbage".

For the reasons discussed above, the Strong CbV calculus that we sketch here is a calculus with ES that does not "recycle" inert garbage (it also is a fragment of the *value substitution calculus* by Accattoli and Paolini, 2012):

Syntax

$$\begin{array}{rrcl} \text{Terms} & t, s & ::= & x \mid v \mid ts \mid t[x{\leftarrow}i] \\ \text{Values} & v & ::= & \lambda x.t \\ \text{Fireballs} & f & ::= & v \mid i \mid f[x{\leftarrow}i] \\ \text{Inert terms} & i & ::= & x \mid if \mid i[x{\leftarrow}i] \end{array}$$

The only difference with λ_{fire} is that here fireballs and inerts can contain ES of the form $[x{\leftarrow}i]$ where i is an inert, justified by the fact that inert substitutions are harmless (see proposition 3.9) and thus they can be explicitly recorded without firing them. The reductions rules are:

Reduction rules (top-level)

$$\begin{array}{rcl} E\langle\lambda x.t\rangle E'\langle v\rangle & \mapsto_{\beta/v} & E'\langle E\langle t\{x{\leftarrow}v\}\rangle\rangle \\ E\langle\lambda x.t\rangle i & \mapsto_{\beta/i} & E\langle t[x{\leftarrow}i]\rangle \end{array}$$

where E stands for environment contexts defined in fig. 4.1, *i.e.* reduction is at a distance. As one can see, a β/i step creates an inert ES which will never be actually substituted, while values can be fired and substituted smoothly.

Since inerts are not substituted, the strong normal forms of this calculus are slightly different than the ones defined in definition 5.1; the only difference is in the presence of (strong) inert ES.

Strong normal forms

$$\begin{array}{rrcl} \text{Strong values} & v_s & ::= & \lambda x.f_s \\ \text{Strong fireballs} & f_s & ::= & i_s \mid v_s \mid f_s[x{\leftarrow}i_s] \\ \text{Strong inert terms} & i_s & ::= & x \mid i_s f_s \mid i_s[x{\leftarrow}i_s] \end{array}$$

Essential strategy. We formulate our Strong CbV calculus by means of an "essential strategy", in the terminology of Accattoli, Faggian, and Guerrieri, 2019. We define an essential strategy $\to_{\mathsf{es}}$ which is non-deterministic: the SCAM then implements a sub-strategy of $\to_{\mathsf{es}}$. Determinism is not actually required by our quantitative analyses, but it is fundamental the property of $\to_{\mathsf{es}}$ known as *random descent*, *i.e.* that all maximal $\to_{\mathsf{es}}$reduction sequences from a term have the same length (a consequence of confluence).

To define $\to_{\mathsf{es}}$, we first introduce *rigid terms*, a generalization of inert terms:

Rigid terms

$$\text{Rigid terms} \quad r, r' \quad ::= \quad x \mid rt \mid r[x{\leftarrow}r']$$

As we see, while inert terms consist of a variable applied to any number of fireballs (and possibly interspered with inert explicit substitutions), rigid terms consist of a variable applied to any number of terms (not necessarily fireballs), possibly interspered with rigid explicit substitutions. Note that, like inert terms, rigid terms are closed under substitutions.

Strong evaluation contexts

$$\begin{array}{rrcl} \text{Weak} & W & ::= & \langle\cdot\rangle \mid Wt \mid tW \mid W[x{\leftarrow}t] \mid t[x{\leftarrow}W] \\ \text{Strong} & S & ::= & \langle\cdot\rangle \mid \lambda x.S \mid R \mid S[x{\leftarrow}r] \mid t[x{\leftarrow}R] \\ \text{Rigid} & R & ::= & rS \mid Rt \mid R[x{\leftarrow}r] \mid r[x{\leftarrow}R] \end{array}$$

Essential strategy $\to_{\mathsf{es}}$

$$\frac{t \mapsto_r s}{S\langle W\langle t\rangle\rangle \to_r S\langle W\langle s\rangle\rangle} \text{ for } r \in \{\beta/v, \beta/i\}$$

We also leave for future work a denotational study of this calculus via multi-types, on which we are already collaborating with Accattoli and Guerrieri.

Machine. The SCAM evaluates crumbled forms that are a slight variant of the ones introduced in chapter 7: here we need to push crumbling to extremes and disallow any nesting of syntax constructs whatsoever. A bite is not anymore either a value or a value applied to a value, but we only allow to apply variables to variables, as follows:

Crumbled forms

$$\begin{array}{rrcl} \text{Bites} & b & ::= & x \mid xy \mid \lambda x.e \quad \text{with } e \text{ non-empty} \\ \text{Crumbles} & e & ::= & \epsilon \mid e[x{\leftarrow}b] \end{array}$$

The reason for this difference is that allowing abstractions inside applications increases the number of machine transitions that are necessary to perform strong evaluation: for instance, to strongly evaluate either one of the bites $\lambda x.e$ or $y(\lambda x.e)$, there would be necessary two different search transitions, one focusing in the body e of $\lambda x.e$, and another one focusing on e on the right part of $y(\lambda x.e)$. By using the crumbling xy we can also get rid of the transition $\to_{\mathsf{sub}/\mathit{left}}$ (that in Crumble GLAMs substitutes abstraction on the left of applications) by simply merging it with $\to_\beta$.

We assume a translation function $\underline{t}$ from λ-terms to crumbles. Note that, like in the Pointed Crumble GLAMs of chapter 10, we identify crumbles with crumbled environments; here we also use a dedicate name $\star$ for the leftmost variable of a crumbled environment.

Crumbling variables. Unlike in part II, here we partition variable names in two disjoint sets: "crumbling variables" and "calculus variables", *i.e.* $\mathcal{V} = \mathcal{V}_{\mathsf{cr}} \uplus \mathcal{V}_{\mathsf{calc}}$. $\mathcal{V}_{\mathsf{cr}}$ contains the variables introduced by the crumbling transformation (and also $\star \in \mathcal{V}_{\mathsf{cr}}$), and $\mathcal{V}_{\mathsf{calc}}$ the variables that are either bound or introduced by β-steps. We carry this distinction because the readback procedure (from crumbled forms to λ-terms with ES, see below) is more complex than the one in part II, as it needs to *undo* the crumbling: the readback turns to real substitutions the ES due to the crumbling transformation, and leave as explicit substitions the ES due to β steps.

We now define the readback of crumbled forms to λ-terms with ES:

Definition 15.1 \ Readback

The readback of a bite b and environment e are, respectively, the terms $b_\downarrow$ and $e_\downarrow$ defined by:

$$\begin{array}{rcl} x_\downarrow & := & x \\ (xy)_\downarrow & := & xy \\ (\lambda x.e)_\downarrow & := & \lambda x.e_\downarrow \\ \\ \epsilon_\downarrow & := & \star \\ e[x{\leftarrow}b]_\downarrow & := & e_\downarrow \begin{cases} \{x{\leftarrow}b_\downarrow\} & \text{if } b = v \text{ or } x \in \mathcal{V}_{\mathsf{cr}} \\ [x{\leftarrow}b] & \text{otherwise} \end{cases} . \end{array}$$

As mentioned, the readback handles each entry $[x{\leftarrow}b]$ of a crumbled environment in different ways: if x is a crumbling variable we perform the actual meta-level substitution; if instead x is a calculus variable, then if v is an abstraction we still perform the substitution (because the calculus substitutes values), otherwise we leave the ES in the readback.

Machine states. We now turn to define the states of the SCAM. A state $\mathcal{S}$ has the form $K\langle l\rangle$, where K is a context and l is what we call a "locus", which stands for the subterm on which the machine is focused.

Machine states

$$\begin{array}{rrcl} \text{Contexts} & K & ::= & \langle\cdot\rangle \mid K[x{\leftarrow}b] \mid e[x{\leftarrow}\lambda y.K] \\ \text{Locus} & l & ::= & e \blacktriangleleft \mid e \blacktriangleright \\ \text{State} & \mathcal{S} & ::= & K\langle l\rangle \end{array}$$

First of all, let us provide some intuitions about the execution of the SCAM. Like Pointed Crumble GLAMs, the SCAM uses a pointer $\blacktriangleleft$ that marks the ES that is being evaluated, scrolling through the environment from right to left. Whenever the SCAM encounters an ES containing an abstraction — say $e[x{\leftarrow}\lambda y.e']\blacktriangleleft$ — the body e' of that abstraction cannot be yet evaluated: in fact, if the variable x occurs in e, the abstraction $\lambda y.e'$ may be need to be copied later, and evaluating its body would break the fundamental subterm property. For this reason, the SCAM operates by levels, alternating two phases, and the $\blacktriangleleft$ phase simply ignores abstractions. During the $\blacktriangleright$ phase, instead, the machine scans the environment again, but from left to right: this time, when it encounters an abstraction like $e \blacktriangleright [x{\leftarrow}\lambda y.e']$, the SCAM can evaluate the body e' because it is sure that that abstraction will not need to be copied anymore.

Let us go back to the syntax. A *locus* is a crumbled environment together with one of the symbols "$\blacktriangleleft$" or "$\blacktriangleright$", denoting the *phase* of the machine. Note that we abuse slightly the notation $K\langle l\rangle$ in the syntax of states: here $K\langle l\rangle$ does not denote the usual plugging but it is more like a pair K, l. However if we remove the pointer from a locus, then the plugging is correct and results in a crumbled form.

About K contexts, the production "$e[x{\leftarrow}\lambda y.K]$" enables strong evaluation, because it allows to evaluate inside the bodies of abstractions. Note that if one removes the production "$e[x{\leftarrow}\lambda y.K]$" and only considers loci having left phase (*i.e.* of the form "$e \blacktriangleleft$") then the syntax collapses to the one of Pointed Crumble GLAMs. In fact, K contexts generalize the evaluated environments that are on the right of the pointer in Pointed Crumble GLAMs. It follows that the notion of *lookup* of a variable in a context K is more involved than usual:

Definition 15.2 \ Lookup in K

We denote by $K(z)$ the lookup of z in the environment induced by the context K, and we define it as follows:

$$\begin{array}{rcl} \langle\cdot\rangle(z) & := & \text{undefined} \\ K[z{\leftarrow}b](z) & := & b \\ K[x{\leftarrow}b](z) & := & K(z) \text{ if } x \neq z \\ e[x{\leftarrow}\lambda y.K](z) & := & K(z). \end{array}$$

Transitions. We provide in fig. 15.1 the transitions for the SCAM.

$$\to_{\mathsf{SCAM}} := \to_{\beta/v} \cup \to_{\beta/i} \cup \to_{\mathsf{sub}/v} \cup \to_{\mathsf{gc}} \cup \to_{\mathsf{src}/1} \cup \cdots \cup \to_{\mathsf{src}/5}$$

$$\begin{array}{rcll}
K\langle e[x{\leftarrow}y\ z] \blacktriangleleft\rangle & \to_{\beta/v} & K\langle e[x{\leftarrow}b]e'\{w{\leftarrow}z\} \blacktriangleleft\rangle & \text{i} \\
K\langle e[x{\leftarrow}y\ z] \blacktriangleleft\rangle & \to_{\beta/i} & K\langle e[x{\leftarrow}b]e' \blacktriangleleft [w{\leftarrow}z]\rangle & \text{ii} \\
K\langle e[x{\leftarrow}yz] \blacktriangleleft\rangle & \to_{\mathsf{sub}/\mathit{left}} & K\langle e[x{\leftarrow}yw] \blacktriangleleft\rangle & \text{iii} \\
K\langle e[x{\leftarrow}z] \blacktriangleleft\rangle & \to_{\mathsf{sub}/v} & K\langle e[x{\leftarrow}w] \blacktriangleleft\rangle & \text{iii} \\
K\langle e[x{\leftarrow}b] \blacktriangleleft\rangle & \to_{\mathsf{src}/1} & K\langle e \blacktriangleleft [x{\leftarrow}b]\rangle & \text{iv} \\
K\langle \blacktriangleleft\rangle & \to_{\mathsf{src}/2} & K\langle \blacktriangleright\rangle & \\
K\langle e \blacktriangleright [x{\leftarrow}b]\rangle & \to_{\mathsf{src}/3} & K\langle e[x{\leftarrow}b] \blacktriangleright\rangle & \text{v} \\
K\langle e \blacktriangleright [x{\leftarrow}v]\rangle & \to_{\mathsf{gc}} & K\langle e \blacktriangleright\rangle & \text{vi} \\
K\langle e \blacktriangleright [x{\leftarrow}\lambda y.e']\rangle & \to_{\mathsf{src}/4} & K\langle e[x{\leftarrow}\lambda y.e' \blacktriangleleft]\rangle & \text{vii} \\
K\langle e[x{\leftarrow}\lambda y.e' \blacktriangleright]\rangle & \to_{\mathsf{src}/5} & K\langle e[x{\leftarrow}\lambda y.e'] \blacktriangleright\rangle &
\end{array}$$

i $K(y)^\alpha = \lambda w.([\star{\leftarrow}b]e')$ and $K(z)$ is a value

ii $K(y)^\alpha = \lambda w.([\star{\leftarrow}b]e')$ and $K(z)$ is not a value

iii if $K(z) = w$ (where w is a variable)

iv if none of the other rules is applicable, *i.e.* when

- b is an abstraction, or
- when b is y or yz, and $K(y)$ is not a variable or an abstraction

v if b is a variable or an application

vi if $x \notin \mathsf{fv}(e)$ and $x \neq \star$

vii if $x \in \mathsf{fv}(e)$

Note: we require that α-equality can rename a name in $\mathcal{V}_{\mathsf{cr}} \setminus \{\star\}$ (resp. $\mathcal{V}_{\mathsf{calc}}$) only with names in $\mathcal{V}_{\mathsf{cr}} \setminus \{\star\}$ (resp. $\mathcal{V}_{\mathsf{calc}}$), and $\star$ cannot be renamed.

Transitions of the SCAM / **Figure 15.1**

As already mentioned, the machine operates in two modes, noted by ▶ and ◀:

- Left ◀: the transitions in this phase (β/v, β/i, $\to_{\mathsf{sub}/\mathit{left}}$, $\to_{\mathsf{sub}/v}$, $\to_{\mathsf{src}/1}$) correspond tightly to the transitions of Pointed Crumble GLAMs, the only difference being that values and inerts are handled by the distinct transitions β/v, β/i.

After a (local) crumbled environment has been weakly evaluated, a $\to_{\mathsf{src}/2}$ transition changes the phase to ▶.

- Right ▶: the transitions in this phase are mainly search transitions that walk through the environment entries on the right of the cursor. Unused abstractions are garbage-collected by $\to_{\mathsf{gc}}$ transitions, and used abstractions are evaluated strongly by entering the environment in their body and switching phase. Unused inert terms, instead, are never garbage-collected because as discussed our Strong CbV calculus keeps the inert garbage.

Garbage collection. The SCAM performs garbage collection, which is unusual for abstract machines. In this case garbage collection is fundamental in order for the SCAM to simulate the calculus that we have provided above. In fact, since the readback always substitutes abstractions, an abstraction that is not used is erased during readback; therefore evaluating the body of an unused abstraction would not correspond to any reduction in the readback, breaking the simulation. Garbage collection is thus necessary to propagate the information about what variables are actually used, so that the side-condition $x \in \mathsf{fv}(e)$ can be checked in time $O(1)$.

Readback. The first step to prove that the SCAM implements the given calculus is defining a readback from states to crumbled terms.

As usual, we define the substitution σ_K induced by a context K, seen as an environment. Like for the readback of crumbled forms (definition 15.1), the following definition either fires or discards each ES:

Definition 15.3 \ Induced substitution σ_K

Let K be a context. Then the *substitution σ_K induced by K* is defined by:

$$\begin{array}{rcl} \sigma_{\langle\cdot\rangle} & \coloneqq & \{\} \\ \sigma_{K[x\leftarrow b]} & \coloneqq & \begin{cases} \sigma_K \circ \{x \leftarrow b_{\downarrow}\} & \text{if } b = v \text{ or } x \in \mathcal{V}_{\mathsf{cr}} \\ \sigma_K & \text{otherwise} \end{cases} \\ \sigma_{e[x\leftarrow \lambda y.K]} & \coloneqq & \sigma_K. \end{array}$$

We now provide the readback of states and related objects:

Definition 15.4 \ Readback

The readback $l_\downarrow$, $K_\downarrow$, and $\mathcal{S}_\downarrow$ of a locus l, a context K, and a state $\mathcal{S}$ are defined by:

$$\begin{array}{rcl} e\blacktriangleleft_\downarrow & := & e_\downarrow \\ e\blacktriangleright_\downarrow & := & e_\downarrow \\ \\ \langle\cdot\rangle_\downarrow & := & \langle\cdot\rangle \\ K[x{\leftarrow}b]_\downarrow & := & K_\downarrow \begin{cases} \{x{\leftarrow}b_\downarrow\} & \text{if } b = v \text{ or } x \in \mathcal{V}_{\mathsf{cr}} \\ [x{\leftarrow}b] & \text{otherwise} \end{cases} \\ e[x{\leftarrow}\lambda y.K]_\downarrow & := & e_\downarrow\{x{\leftarrow}\lambda y.K_\downarrow\} \\ \\ K\langle l\rangle_\downarrow & := & K_\downarrow\langle l_\downarrow\sigma_K\rangle \end{array}$$

The readback of a state $K\langle l\rangle$ is defined as the readback of the context K in which we plug the crumbled environment $l_\downarrow\sigma_K$: the substitution σ_K captures the substitutions fired during the readback of K and impacting the hole, that need to be re-applied to $l_\downarrow$ before plugging it back in.

A crucial point of this definition is that the readback $K_\downarrow$ is not actually a context in the usual sense: the hole of K may be duplicated or erased during readback, since it may be contained in an ES that is fired during readback. (Consider for instance the case $e[x{\leftarrow}\lambda y.K]_\downarrow = e_\downarrow\{x{\leftarrow}\lambda y.K_\downarrow\}$ where x occurs multiple times in $e_\downarrow$, or none at all.)

Multi-contexts. We thus need to generalize the contexts S and R defined on page 221 to the multi-contexts $\mathbb{S}$ and $\mathbb{R}$, as follows:

Multi-contexts

$$\begin{array}{rrcl} \text{Strong} & \mathbb{S} & ::= & \langle\cdot\rangle \mid t \mid \lambda x.\mathbb{S} \mid \mathbb{R} \mid \mathbb{S}[x{\leftarrow}\mathbb{R}] \\ \text{Rigid} & \mathbb{R} & ::= & x \mid \mathbb{R}\mathbb{S} \mid \mathbb{R}[x{\leftarrow}\mathbb{R}] \end{array}$$

A multi-context is basically a context where the hole $\langle\cdot\rangle$ may occur not only once but multiple times (even zero times are allowed, but in this case we say that the context is not *proper*). Plugging in a multi-context amounts to replacing all the occurrences of $\langle\cdot\rangle$ with the plugged term.

The notion of plugging in multi-contexts is conceptually the same as with usual contexts, only in multi-contexts the plugging replaces *all* the holes present in the context (if any) with the given term.

A consequence of the duplication of holes during readback is that to each beta step of the SCAM may correspond multiple steps in the calculus, as stated by the property *principal projection* of upcoming theorem 15.7. This means that not only subterms are shared along the execution, but also *computation*. This is something that does not occur in usual weak/open machines for CbV, being instead a feature specific to CbNeed. Clearly the SCAM contains an

element of the "by-Need" attitude, and this suggests that the SCAM could be also adapted to the harder case of Strong CbNeed evaluation in the future.

Invariants. The invariants required to prove the implementation theorem include the usual ones for Pointed Crumble GLAMs, like the subterm property (*i.e.* that bodies of abstractions are sub-crumbles of the initial crumble), and *well-namedness* (to ensure that there are no duplicates among the variables in the domain of K contexts). The only fundamental difference is that now the *context decoding* property is much harder:

Theorem 15.5 \ Context decoding

Let $K\langle l\rangle$ be a reachable state. Then:

1. $K_{\downarrow}$ is a proper strong multi-context,
2. if $l = e[x{\leftarrow}b] \blacktriangleleft$ then $e[x{\leftarrow}\langle\cdot\rangle]_{\downarrow} = R$ for some right v-context R,
3. if $l = e \blacktriangleright$ then $e_{\downarrow}$ is a strong fireball compatible with σ_K.

Point 1 states that the readback of K should be a proper strong multi-context, defined above. Point 2 of theorem 15.5 is exactly the old context decoding requirement of Crumble GLAMs: the readback of the left part of a pointed environment should be a right v-context. This shows that, during the $\blacktriangleleft$ phase, the machine is basically performing Open CbV by simulating Crumble GLAMs.

Point 3 basically states that during the $\blacktriangleright$ phase the crumbled environment e on the left of the pointer should be in strong normal form. What it actually states is more precise: $e_{\downarrow}$ should also be compatible with σ_K, meaning that in some way the redexes caused by substitutions caused by K have already been fully reduced. The formal definition of compatibility follows:

Definition 15.6 \ Compatibility with a fireball substitution

Let f_s be a strong fireball. f_s is compatible with a fireball substitution σ if every variable x such that $\sigma(x) = v$ does not have applied free occurrences in f_s.

Implementation. To prove that the SCAM implements the essential strategy introduced above, we follow the distillation technique outlined in section 4.4:

Theorem 15.7 \ Implementation system

Let t be a λ-term, and let $\mathcal{S}$ be a reachable state.

1. *Initialization:* $\underline{t}_{\downarrow} = t$.
2. *Principal projection:*
   - if $\mathcal{S} \to_{\beta/v} \mathcal{S}'$ then $\mathcal{S}_{\downarrow} \equiv\!\to^{*}_{\beta/v}\!\equiv \mathcal{S}'_{\downarrow}$,

- if $\mathcal{S} \to_{\beta/i} \mathcal{S}'$ then $\mathcal{S}_\downarrow \equiv \to^*_{\beta/i} \equiv \mathcal{S}'_\downarrow$.

3. *Overhead transparency:* if $\mathcal{S} \to_r \mathcal{S}'$ then $\mathcal{S}_\downarrow = \mathcal{S}'_\downarrow$ for every rule $r \in \{\mathsf{sub}/v, \mathsf{src}_1, \mathsf{src}_2, \mathsf{src}_3, \mathsf{src}_4, \mathsf{src}_5, \mathsf{gc}\}$
4. *Determinism:* the transition $\to_{\mathsf{SCAM}}$ is deterministic.
5. *Confluence:* the reduction $\to_{\mathsf{es}}$ is confluent.
6. *Halt:* if $\mathcal{S}$ is $\to_{\mathsf{SCAM}}$-normal, then $\mathcal{S}_\downarrow$ is $\to_{\mathsf{es}}$-normal.
7. *Overhead termination:* $\to_r$ terminates for every rule $r \in \{\mathsf{sub}/v, \mathsf{src}_1, \mathsf{src}_2, \mathsf{src}_3, \mathsf{src}_4, \mathsf{src}_5, \mathsf{gc}\}$.

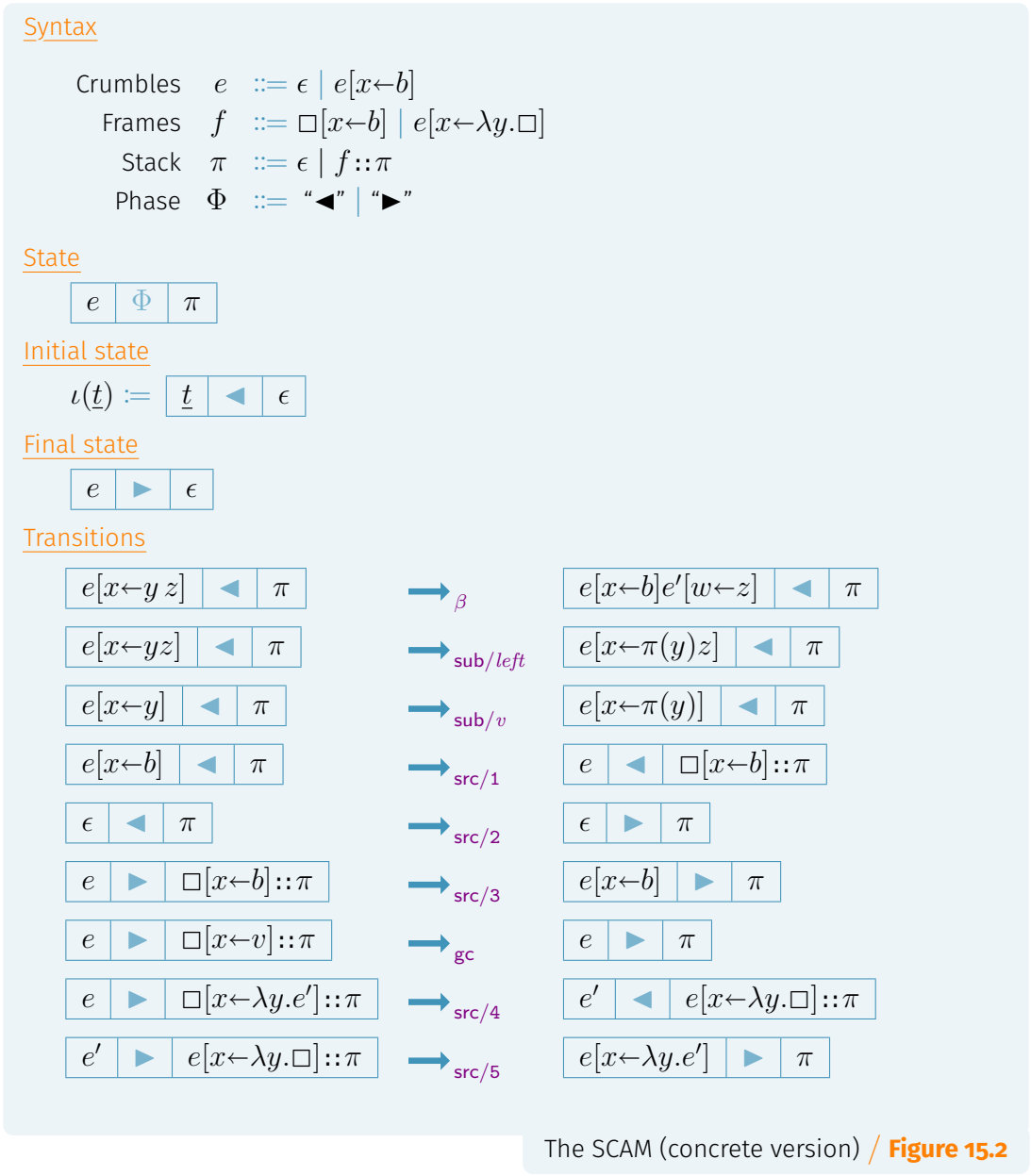
(Note: $\equiv$ is the structural equivalence on terms with ES mentioned in section 4.4.)

There is however a mismatch: the notion of distillation (page 57) requires both the transition relation of the machine and the reduction of the calculus to be deterministic. Our case is different, because the reduction $\to_{\mathsf{es}}$ is not deterministic: however $\to_{\mathsf{es}}$ is confluent (requirement 5 of theorem 15.7), thus it is necessary to generalize the notion of implementation system from a deterministic strategy to a confluent one.

Complexity. To obtain an efficient implementation we assume exactly the same underlying λ-graph implementation required by Pointed Crumble GLAMs, so that for instance the lookup in the environment can be performed in constant time.

As usual, the cost of each $\{\to_{\beta/v}, \to_{\beta/i}\}$ transition is proportional to the size of the initial term: these transitions copy the body of an unevaluated abstraction, which by the subterm property is a subterm of the initial term. The cost of search transitions is costant, because they simply move/switch the pointer around the crumble. Each $\{\to_{\mathsf{src}/4}, \to_{\mathsf{gc}}\}$ transition needs to check whether a variable x is free in the evaluated environment on the left of the ▶ cursor: this is achieved in costant time by checking whether the list of parents of the node corresponding to x in the λ-graph is empty.

Merged version. We leave the whole technical development outlined above for future work (actually, a paper in collaboration with Accattoli, Guerrieri, Leberle and Sacerdoti Coen about Strong CbV and containing the machine above is currently under peer review). Before concluding, we provide a more "concrete" version of the SCAM (fig. 15.2) where the evaluation context K is encoded inside-out, in a stack, as in the abstract machines of previous chapters. (The side-conditions of the transitions are the same as in fig. 15.1.)



The SCAM (concrete version) / **Figure 15.2**

15.2 Convertibility in pCiC

As mentioned in the introduction and in chapter 5, every proof assistant based on a dependent type theory relies during type-checking on a subprocedure that decides whether two given terms are *convertible*.

This dissertation provides evidence that obtaining an efficient algorithm for β-convertibility is already very difficult in the classical λ-calculus, let alone in proof assistants whose type theories are much richer. Let us consider for instance the *Coq* proof assistant, based on the *predicative Calculus of (Co-)Inductive Constructions* (pCiC).

In pCiC, convertibility is not simply β-convertibility. To start with, pCiC contains — other than the usual β — the following additional reduction rules:[1]

- ι ι-reduction basically says that a destructor applied to an object built from a constructor behaves as expected. This reduction regulates the interaction between (fully applied) constructors, terms defined by (co)fixpoint, and pattern matching (`match•with•end`).

  Basically, ι can be seen as a generalization of the reduction rules for the `if • then • else` construct that we have already studied under crumbling in Accattoli et al., 2019b.

- δ δ-reduction expands the value of a defined costant. For instance, $c \to_\delta t$ if the current global environment contains the definition "$c := t$".

- ζ ζ-reduction removes a local definition occurring in a term (*i.e.* a `let`-expression) by replacing all the occurrences of the defined variable by its value:

$$\texttt{let}\ x := u\ \texttt{in}\ t \quad \to_\zeta \quad t\{x \leftarrow u\}.$$

- η η-expansion is the following rule:

  η-expansion

$$t \quad \mapsto_\eta \quad \lambda x.(t\,x) \qquad \text{if } x \notin \mathsf{fv}(t)$$

  which can be applied only if t has functional type, and only lazily during the convertibility check because η as a standalone reduction is non-terminating. Note: the converse rule, called η-reduction, is not allowed in pCiC, as it breaks a desirable property of the type system called *subject reduction* (it does not preserve types).

Therefore the correct notion for conversion in Coq is $\beta\iota\delta\zeta\eta$-convertibility; this however is still not the whole story, as we see in the next paragraph.

Cumulativity. The type theory of Coq features so-called *sorts*: `Prop`, `Set`, and a $\texttt{Type}_i$ for each natural number i. Sorts are *cumulative*:

$$\begin{aligned} &\texttt{Prop} \leq \texttt{Set} \\ &\texttt{Set} \leq \texttt{Type}_0 \\ &\texttt{Type}_i \leq \texttt{Type}_j \quad (\text{if } i \leq j) \end{aligned}$$

meaning that each term of type `Prop` is also of type `Set`, that each term of type `Set` is also of type $\texttt{Type}_0$, and that each term of type $\texttt{Type}_i$ is also of type $\texttt{Type}_j$ when $i \leq j$. The cumulativity relation is also an order, thus reflexive and transitive.

Because of cumulativity, the correct notion to use during type-checking is not plain convertibility but a *subtyping* relation that encompasses both conversion and cumulativity (see Luo, 1990), obtained by closing $\leq$ under the following additional rules:[2]

[1] See https://coq.inria.fr/refman/language/cic.html.
[2] Where Γ is a typing context.

$$\frac{\Gamma \vdash t =_{\beta\iota\delta\zeta\eta} s}{\Gamma \vdash t \leq s} \qquad \frac{\Gamma \vdash T =_{\beta\iota\delta\zeta\eta} U \qquad \Gamma, x\colon T \vdash t \leq u}{\Gamma \vdash \forall x\colon T.\, t \leq \forall x\colon U.\, u}$$

The rule on the left says that two convertible terms are always in the subtyping relation. The rule on the right says that $(\forall x\colon T.\, t)$ is a subtype of $(\forall x\colon U.\, u)$ whenever T and U are convertible[3] and t is a subtype of u.

More features. Coq is a rapidly evolving software, and currently its type system contains many more features than the ones scketched above: for instance, universe polymorphism, variance for general inductive types, sort polymorphism, and more constructs (like primitive record projections). All these features clearly require to extend the subtyping relation even further.

Future work. As for Comparison, our sharing equality algorithm can be easily extended to the additional syntactical constructs of Coq. We were not successful in extending sharing equality to the case of cumulativity. The algorithm presented in this dissertation[4], being based on Paterson-Wegman's algorithm, relies heavily on a mechanism of equivalence classes in order to have linear-time complexity: in particular, it requires that the computed relation is symmetric by considering query edges bidirectional. While convertibility is an equivalence relation, the subtyping relation is not symmetric, therefore it is not evident how PW could be adapted to this case.

As for Computation, it is not difficult to extend the abstract machines introduced in this dissertation to the additional constructs present in CoC: see for instance Accattoli et al., 2019b, where we show that our crumbling machines scale effortlessly to a calculus with booleans and the `if•then•else` construct. We supervised a master student, Andrea Pasquali, which investigated in his thesis how to extend evaluation to the additional constructs of Coq. One of the challenges of our crumbling representation, which disallows iterated applications, is to access in constant time to the n-th applied argument: this is necessary for instance for fixpoints and pattern matchings, which can reduce according to the n-th applied argument.

Practical convertibility

What is difficult is interweaving Computation and Comparison while staying efficient with respect to the chosen strategy. As already mentioned, proof assistants never evaluate terms to their full normal form, but limit reduction as much as possible guiding it through finely tuned heuristics: the impact of these optimizations is fundamental in practice for the performance of these tools but not much studied. The basic behaviour to check convertibility of given terms is to first compute their weak head normal form, which exposes the top-level rigid structure of the normal form; only later, if the outermost structure of the types agree, iterate the process over arguments and abstractions. A fundamental optimization is to limit when possible the costly unfolding of definitions (δ), because each new unfolding forces

[3] To obtain contravariant subtyping one can require that $U \leq T$.
[4] See algorithm 4.

Comparison to traverse again the definition: some heuristics of Coq delay the unfolding, and only in case of failure (for instance when conversion fails due to universe constraints) then the conversion test is performed again after unfolding. Another important mechanism in Coq is the one of *opaque* definitions, which allow to block their unfolding, improving performance.

Moreover, further research is necessary to refine the bounds on the asymtotic complexity of our abstract machines: we bounded the cost of evaluation according to the size of the initial term, but — if interpreted literally — in the case of Coq the initial term is the current Coq library together with its complete dependecies, whose size is prohibitive.

Conversion, PW-style In order to interweave evaluation and sharing equality so to fail early, we also investigated the extension of our sharing equality algorithm from part III to an algorithm for β-convertibility working directly on a λ-graph and following a "Paterson-Wegman strategy". For the sake of clarity, let us call this potential algorithm "PWConversion".

The basic idea is to allow as inputs of PWConversion λ-graphs that are not necessarily in β-normal form. An initial query $\mathsf{r}\ \mathcal{Q}\ \mathsf{r}'$ between two root nodes r, r' then should correspond to requesting whether $[\![\mathsf{r}]\!] =_\beta [\![\mathsf{r}']\!]$. PWConversion can proceed non-deterministically, by iterating over all nodes, picking one that has not been processed yet, and calling a procedure similar to BuildClass. Clearly at some point a node n which is being processed by BuildClass will need to be evaluated, in order to expose what its true constructor is, so that it can be compared to other possible nodes in the equivalence class. Following the PW strategy by first evaluating the parents of a node n, and only later n (if necessary) seems to respect a property similar to the *subterm property*, which in abstract machines is the key to bilinearity.

Unfortunately, the interplay between the mechanism of equivalence classes and PW's cycle detection complicates the study of such an algorithm, and this is why we leave PWConversion for future research.

Bibliography

Church, Alonzo (1941). *The Calculi of Lambda Conversion. (AM-6)*. Princeton University Press. ISBN: 9780691083940.

Ershov, Andrei P. (1958). "On Programming of Arithmetic Operations". In: *Commun. ACM* 1.8, pp. 3–9. DOI: 10.1145/368892.368907. URL: https://doi.org/10.1145/368892.368907.

Landin, P. J. (1964). "The Mechanical Evaluation of Expressions". In: *Comput. J.* 6.4, pp. 308–320. DOI: 10.1093/comjnl/6.4.308.

Landin, Peter J. (1965). "Correspondence between ALGOL 60 and Church's Lambda-notation: part I". In: *Commun. ACM* 8.2, pp. 89–101. DOI: 10.1145/363744.363749.

Hopcroft, J. and R. Karp (1971). *A Linear Algorithm for Testing Equivalence of Finite Automata*. Tech. rep. 0. Dept. of Computer Science, Cornell University. URL: https://ecommons.cornell.edu/handle/1813/5958.

Wadsworth, Christopher P. (1971). "Semantics and pragmatics of the lambda-calculus". Chapter 4. PhD Thesis. Oxford.

Boyer, Robert S and Jay S Moore (1972). "The sharing of structure in theorem-proving programs". In: *Machine intelligence* 7, pp. 101–116.

Huet, Gérard P. (1973). "The Undecidability of Unification in Third Order Logic". In: *Information and Control* 22.3, pp. 257–267. DOI: 10.1016/S0019-9958(73)90301-X.

Berkling, Klaus J. (1974). "Reduction Languages for Reduction Machines". In: *Proceedings of the 2nd Annual Symposium on Computer Architecture, Houston, TX, USA, December 1974*, pp. 133–140. DOI: 10.1145/642089.642112.

Goto, Eiichi (1974). *Monocopy and associative algorithms in extended Lisp*. Technical report TR 74âĂŞ03. University of Tokyo.

Plotkin, Gordon D. (1975). "Call-by-Name, Call-by-Value and the lambda-Calculus". In: *Theor. Comput. Sci.* 1.2, pp. 125–159.

Tarjan, Robert Endre (1975). "Efficiency of a Good But Not Linear Set Union Algorithm". In: *J. ACM* 22.2, pp. 215–225. DOI: 10.1145/321879.321884. URL: https://doi.org/10.1145/321879.321884.

Wadsworth, Christopher P. (1976). "The Relation Between Computational and Denotational Properties for Scott's D_{infty}-Models of the Lambda-Calculus". In: *SIAM J. Comput.* 5.3, pp. 488–521.

Martelli, Alberto and Ugo Montanari (1977). "Theorem Proving with Structure Sharing and Efficient Unification". In: *Proceedings of the 5th International Joint Conference on Artificial*

Intelligence. Cambridge, MA, USA, August 22-25, 1977, p. 543. URL: http://ijcai.org/Proceedings/77-1/Papers/096.pdf.

Allen, John (1978). *Anatomy of LISP*. New York, NY, USA: McGraw-Hill, Inc. ISBN: 0-07-001115-X.

Levy, Jean-Jacques (1978). "RÃľductions correctes et optimales dans le lambda-calcul". PhD thesis. These d'Etat, Universite Paris 7.

Paterson, Mike and Mark N. Wegman (1978). "Linear Unification". In: *Journal of Computer and System Sciences* 16.2, pp. 158–167. DOI: 10.1016/0022-0000(78)90043-0.

Martelli, Alberto and Ugo Montanari (1982). "An Efficient Unification Algorithm". In: *ACM Trans. Program. Lang. Syst.* 4.2, pp. 258–282. DOI: 10.1145/357162.357169.

Barendregt, Hendrik Pieter (1984). *The Lambda Calculus – Its Syntax and Semantics*. Vol. 103. North-Holland.

Cousineau, Guy, Pierre-Louis Curien, and Michel Mauny (1985). "The Categorical Abstract Machine". In: *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, pp. 50–64. DOI: 10.1007/3-540-15975-4_29.

Kieburtz, Richard B. (1985). "The G-Machine: A Fast, Graph-Reduction Evaluator". In: *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, pp. 400–413. DOI: 10.1007/3-540-15975-4_50.

Champeaux, Dennis de (1986). "About the Paterson-Wegman linear unification algorithm". In: *Journal of Computer and System Sciences* 32.1, pp. 79 –90. ISSN: 0022-0000.

Felleisen, Matthias and Daniel P. Friedman (Aug. 1986). "Control operators, the SECD-machine, and the lambda-calculus". In: *3rd Working Conference on the Formal Description of Programming Concepts*.

Girard, Jean-Yves (1987). "Linear Logic". In: *Theoretical Computer Science* 50, pp. 1–102.

Crégut, Pierre (1990). "An Abstract Machine for Lambda-Terms Normalization". In: *LISP and Functional Programming*, pp. 333–340.

Gabow, Harold N. (1990). "Data Structures for Weighted Matching and Nearest Common Ancestors with Linking". In: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*. Pp. 434–443. URL: http://dl.acm.org/citation.cfm?id=320176.320229.

Lamping, John (1990). "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*, pp. 16–30. DOI: 10.1145/96709.96711.

Leroy, Xavier (1990). *The ZINC experiment: an economical implementation of the ML language*. Technical report 117. INRIA. URL: http://gallium.inria.fr/~xleroy/publi/ZINC.pdf.

Luo, Zhaohui (1990). "An extended calculus of constructions". PhD thesis. University of Edinburgh, UK. URL: http://hdl.handle.net/1842/12487.

Abadi, Martín et al. (1991). "Explicit substitutions". In: *Journal of Functional Programming* 4.1, pp. 375–416.

Curien, Pierre-Louis (1991). "An Abstract Framework for Environment Machines". In: *Theor. Comput. Sci.* 82.2, pp. 389–402. DOI: 10.1016/0304-3975(91)90230-Y. URL: https://doi.org/10.1016/0304-3975(91)90230-Y.

Cytron, Ron et al. (1991). “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4, pp. 451–490. DOI: 10.1145/115372.115320.

Moggi, Eugenio (1991). “Notions of Computation and Monads”. In: *Information and Computation* 93.1, pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4.

Danvy, Olivier and Andrzej Filinski (1992). “Representing Control: A Study of the CPS Transformation”. In: *Mathematical Structures in Computer Science* 2.4, pp. 361–391. DOI: 10.1017/S0960129500001535.

Gonthier, Georges, Martín Abadi, and Jean-Jacques Lévy (1992). “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*, pp. 15–26. DOI: 10.1145/143165.143172.

Jones, Simon L. Peyton (1992). “Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine”. In: *J. Funct. Program.* 2.2, pp. 127–202. DOI: 10.1017/S0956796800000319.

Regnier, Laurent (1992). “Lambda-calcul et réseaux”. PhD thesis. Univ. Paris VII.

Rose, Kristoffer Høgsbro (1992). “Explicit Cyclic Substitutions”. In: *Conditional Term Rewriting Systems, Third International Workshop, CTRS-92, Pont-à-Mousson, France, July 8-10, 1992, Proceedings*, pp. 36–50. DOI: 10.1007/3-540-56393-8_3.

Consel, Charles and Olivier Danvy (1993). “Tutorial Notes on Partial Evaluation”. In: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pp. 493–501. DOI: 10.1145/158511.158707.

Flanagan, Cormac et al. (1993). “The essence of compiling with continuations (with retrospective)”. In: *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*, pp. 502–514. DOI: 10.1145/989393.989443.

Launchbury, John (1993). “A Natural Semantics for Lazy Evaluation”. In: *POPL*, pp. 144–154.

Qian, Zhenyu (1993). “Linear Unification of Higher-Order Patterns”. In: *TAPSOFT’93: Theory and Practice of Software Development, International Joint Conference CAAP/FASE, Orsay, France, April 13-17, 1993, Proceedings*, pp. 391–405. DOI: 10.1007/3-540-56610-4_78.

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

Danvy, Olivier (1994). “Back to Direct Style”. In: *Science of Computer Programming* 22.3, pp. 183–195. DOI: 10.1016/0167-6423(94)00003-4.

Hatcliff, John and Olivier Danvy (1994). “A Generic Account of Continuation-Passing Styles”. In: *21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1994*. ACM Press, pp. 458–471. DOI: 10.1145/174675.178053.

Papadimitriou, Christos (1994). *Computational Complexity*. Addison Wesley.

Blelloch, Guy E. and John Greiner (1995). “Parallelism in Sequential Functional Languages”. In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pp. 226–237. DOI: 10.1145/224164.224210.

Bloo, Roel and Kristoffer Rose (1995). “Preservation of Strong Normalization in Named Lambda Calculi with Explicit Substitution and Garbage Collection”. In: *Computing Science in the Netherlands*. Netherlands Computer Science Research Foundation, pp. 62–72.

Lescanne, Pierre and Jocelyne Rouyer-Degli (1995). “Explicit Substitutions with de Bruijn’s Levels”. In: *Rewriting Techniques and Applications, 6th International Conference, RTA-95, Kaiserslautern, Germany, April 5-7, 1995, Proceedings*, pp. 294–308. DOI: 10.1007/3-540-59200-8_65.

Melliès, Paul-André (1995). “Typed lambda-calculi with explicit substitutions may not terminate”. In: *TLCA*, pp. 328–334.

Ariola, Zena M. and Jan Willem Klop (1996). “Equational Term Graph Rewriting”. In: *Fundam. Inform.* 26.3/4, pp. 207–240. DOI: 10.3233/FI-1996-263401. URL: https://doi.org/10.3233/FI-1996-263401.

Ferreira, Maria C. F., Delia Kesner, and Laurence Puel (1996). “Lambda-Calculi with Explicit Substitutions and Composition Which Preserve Beta-Strong Normalization”. In: *Algebraic and Logic Programming, 5th International Conference, ALP’96, Aachen, Germany, September 25-27, 1996, Proceedings*, pp. 284–298. DOI: 10.1007/3-540-61735-3_19.

Ariola, Zena M. and Matthias Felleisen (1997). “The Call-By-Need lambda Calculus”. In: *J. Funct. Program.* 7.3, pp. 265–301.

Ariola, Zena M. and Jan Willem Klop (1997). “Lambda Calculus with Explicit Recursion”. In: *Inf. Comput.* 139.2, pp. 154–233. DOI: 10.1006/inco.1997.2651. URL: https://doi.org/10.1006/inco.1997.2651.

Di Cosmo, Roberto and Delia Kesner (1997). “Strong Normalization of Explicit Substitutions via Cut Elimination in Proof Nets (Extended Abstract)”. In: *LICS*, pp. 35–46.

Huet, Gérard P. (1997). “The Zipper”. In: *J. Funct. Program.* 7.5, pp. 549–554. URL: http://journals.cambridge.org/action/displayAbstract?aid=44121.

Leroy, Xavier (June 1997). “The effectiveness of type-based unboxing”. In: *TIC 1997: workshop Types in Compilation*. Technical report BCCS-97-03, Boston College, Computer Science Department. URL: http://xavierleroy.org/publi/unboxing-tic97.pdf.

Sestoft, Peter (1997a). “Deriving a Lazy Abstract Machine”. In: *J. Funct. Program.* 7.3, pp. 231–264. URL: http://journals.cambridge.org/action/displayAbstract?aid=44087.

— (1997b). “Deriving a Lazy Abstract Machine”. In: *J. Funct. Program* 7.3, pp. 231–264.

Asperti, Andrea and Stefano Guerrini (1998). *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press.

Asperti, Andrea and Harry G. Mairson (1998). “Parallel Beta Reduction is not Elementary Recursive”. In: *POPL*, pp. 303–315.

Buchsbaum, Adam L. et al. (1998). “A New, Simpler Linear-Time Dominators Algorithm”. In: *ACM Trans. Program. Lang. Syst.* 20.6, pp. 1265–1296. DOI: 10.1145/295656.295663. URL: https://doi.org/10.1145/295656.295663.

Kutzner, Arne and Manfred Schmidt-Schauß (1998). “A Non-Deterministic Call-by-Need Lambda Calculus”. In: *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP ’98), Baltimore, Maryland, USA, September 27-29, 1998*. Pp. 324–335. DOI: 10.1145/289423.289462.

Maraist, John, Martin Odersky, and Philip Wadler (1998). "The Call-by-Need Lambda Calculus". In: *J. Funct. Program.* 8.3, pp. 275–317.

Alstrup, Stephen et al. (1999). "Dominators in Linear Time". In: *SIAM J. Comput.* 28.6, pp. 2117–2132. DOI: 10.1137/S0097539797317263.

Okasaki, Chris (1999). *Purely functional data structures*. Cambridge University Press. ISBN: 978-0-521-66350-2.

Paolini, Luca and Simona Ronchi Della Rocca (1999). "Call-by-value Solvability". In: *ITA* 33.6, pp. 507–534.

Barendregt, Henk and Silvia Ghilezan (2000). "Lambda terms for natural deduction, sequent calculus and cut elimination". In: *J. Funct. Program.* 10.1, pp. 121–134. URL: http://journals.cambridge.org/action/displayAbstract?aid=44279.

Curien, Pierre-Louis and Hugo Herbelin (2000). "The duality of computation". In: *ICFP*, pp. 233–243.

Blom, Stefanus Cornelis Christoffel (2001). "Term Graph Rewriting. Syntax and semantics". PhD thesis. Vrije Universiteit Amsterdam.

Grégoire, Benjamin and Xavier Leroy (2002). "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*. Pp. 235–246. DOI: 10.1145/581478.581501.

Sands, David, Jörgen Gustavsson, and Andrew Moran (2002). "Lambda Calculi and Linear Speedups". In: *The Essence of Computation*, pp. 60–84.

Danvy, Olivier (2003a). "A Journey from Interpreters to Compilers and Virtual Machines". In: *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, p. 117. DOI: 10.1007/978-3-540-39815-8_7.

— (2003b). "A New One-Pass Transformation into Monadic Normal Form". In: *Compiler Construction, 12th International Conference, CC 2003*. Vol. 2622. Lecture Notes in Computer Science. Springer, pp. 77–89. DOI: 10.1007/3-540-36579-6_6.

Urban, Christian, Andrew M. Pitts, and Murdoch Gabbay (2003). "Nominal Unificaiton". In: *Computer Science Logic, 17th International Workshop, CSL 2003, 12th Annual Conference of the EACSL, and 8th Kurt Gödel Colloquium, KGC 2003, Vienna, Austria, August 25-30, 2003, Proceedings*, pp. 513–527. DOI: 10.1007/978-3-540-45220-1_41. URL: https://doi.org/10.1007/978-3-540-45220-1_41.

Danvy, Olivier and Lasse R. Nielsen (2004). *Refocusing in Reduction Semantics*. Tech. rep. RS-04-26. BRICS.

Ronchi Della Rocca, Simona and Luca Paolini (2004). *The Parametric λ-Calculus*. Springer Berlin Heidelberg.

Lassen, Søren B. (2005). "Eager Normal Form Bisimulation". In: *20th IEEE Symposium on Logic in Computer Scienc, LICS 2005*. IEEE Computer Society, pp. 345–354. DOI: 10.1109/LICS.2005.15.

Friedman, Daniel P. et al. (2007). "Improving the lazy Krivine machine". In: *Higher-Order and Symbolic Computation* 20.3, pp. 271–293.

Gonthier, Georges (2007). "The Four Colour Theorem: Engineering of a Formal Proof". In: *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, p. 333. DOI: 10.1007/978-3-540-87827-8_28.

Kennedy, Andrew (2007). "Compiling with continuations, continued". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pp. 177–190. DOI: 10.1145/1291151.1291179.

Kesner, Delia (2007). "The Theory of Calculi with Explicit Substitutions Revisited". In: *CSL*, pp. 238–252.

Krivine, Jean-Louis (2007). "A call-by-name lambda-calculus machine". In: *Higher-Order and Symbolic Computation* 20.3, pp. 199–207.

Milner, Robin (2007). "Local Bigraphs and Confluence: Two Conjectures". In: *Electr. Notes Theor. Comput. Sci.* 175.3, pp. 65–73.

Wand, Mitchell (2007). "On the correctness of the Krivine machine". In: *Higher-Order and Symbolic Computation* 20.3, pp. 231–235. DOI: 10.1007/s10990-007-9019-8.

Selinger, Peter (2008). "Lecture notes on the lambda calculus". In: *CoRR* abs/0804.3434. arXiv: 0804.3434. URL: http://arxiv.org/abs/0804.3434.

Ariola, Zena M., Aaron Bohannon, and Amr Sabry (2009). "Sequent calculi and abstract machines". In: *ACM Trans. Program. Lang. Syst.* 31.4.

Accattoli, Beniamino and Delia Kesner (2010). "The Structural λ-Calculus". In: *CSL*, pp. 381–395.

Calvès, Christophe and Maribel Fernández (2010a). "Matching and alpha-equivalence check for nominal terms". In: *J. Comput. Syst. Sci.* 76.5, pp. 283–301. DOI: 10.1016/j.jcss.2009.10.003.

— (2010b). "The First-Order Nominal Link". In: *Logic-Based Program Synthesis and Transformation - 20th International Symposium, LOPSTR 2010, Hagenberg, Austria, July 23-25, 2010, Revised Selected Papers*, pp. 234–248. DOI: 10.1007/978-3-642-20551-4_15.

Danvy, Olivier et al. (2010). "Defunctionalized Interpreters for Call-by-Need Evaluation". In: *FLOPS*, pp. 240–256.

Levy, Jordi and Mateu Villaret (2010). "An Efficient Nominal Unification Algorithm". In: *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*, pp. 209–226. DOI: 10.4230/LIPIcs.RTA.2010.209.

Accattoli, Beniamino (2012). "An Abstract Factorization Theorem for Explicit Substitutions". In: *RTA*, pp. 6–21.

Accattoli, Beniamino and Delia Kesner (2012). "The Permutative λ-Calculus". In: *LPAR*, pp. 23–36.

Accattoli, Beniamino and Ugo Dal Lago (2012). "On the Invariance of the Unitary Cost Model for Head Reduction". In: *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, pp. 22–37. DOI: 10.4230/LIPIcs.RTA.2012.22.

Accattoli, Beniamino and Luca Paolini (2012). "Call-by-Value Solvability, revisited". In: *FLOPS*, pp. 4–16.

Chang, Stephen and Matthias Felleisen (2012). "The Call-by-Need Lambda Calculus, Revisited". In: *ESOP*, pp. 128–147.

Charguéraud, Arthur (2012). "The Locally Nameless Representation". In: *J. Autom. Reasoning* 49.3, pp. 363–408. DOI: 10.1007/s10817-011-9225-2. URL: https://doi.org/10.1007/s10817-011-9225-2.

Ehrhard, Thomas (2012). "Collapsing non-idempotent intersection types". In: *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, pp. 259–273. DOI: 10.4230/LIPIcs.CSL.2012.259.

Balabonski, Thibaut (2013). "Weak optimality, and the meaning of sharing". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pp. 263–274. DOI: 10.1145/2500365.2500606.

Calvès, Christophe (2013). "Unifying Nominal Unification". In: *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, pp. 143–157. DOI: 10.4230/LIPIcs.RTA.2013.143.

García-Pérez, Álvaro, Pablo Nogueira, and Juan José Moreno-Navarro (2013). "Deriving the full-reducing Krivine machine from the small-step operational semantics of normal order". In: *PPDP*, pp. 85–96.

Schmidt-Schauß, Manfred, Conrad Rau, and David Sabel (2013). "Algorithms for Extended Alpha-Equivalence and Complexity". In: *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, pp. 255–270. DOI: 10.4230/LIPIcs.RTA.2013.255. URL: https://doi.org/10.4230/LIPIcs.RTA.2013.255.

Accattoli, Beniamino, Pablo Barenbaum, and Damiano Mazza (2014a). "Distilling abstract machines". In: *ICFP 2014*, pp. 363–376. URL: http://doi.acm.org/10.1145/2628136.2628154.

— (2014b). *Distilling Abstract Machines (Long Version)*. arXiv: 1406.2370.

Accattoli, Beniamino and Claudio Sacerdoti Coen (2014). "On the Value of Variables". In: *WoLLIC 2014*, pp. 36–50. URL: http://dx.doi.org/10.1007/978-3-662-44145-9_3.

Accattoli, Beniamino and Ugo Dal Lago (2014). "Beta reduction is invariant, indeed". In: *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, 8:1–8:10. DOI: 10.1145/2603088.2603105.

Accattoli, Beniamino et al. (2014). "A nonstandard standardization theorem". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pp. 659–670. DOI: 10.1145/2535838.2535886.

Carraro, Alberto and Giulio Guerrieri (2014). "A Semantical and Operational Account of Call-by-Value Solvability". In: *Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pp. 103–118. DOI: 10.1007/978-3-642-54830-7_7.

Grabmayer, Clemens and Jan Rochel (2014). "Maximal sharing in the Lambda calculus with letrec". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pp. 67–80. DOI: 10.1145/2628136.2628148.

Accattoli, Beniamino, Pablo Barenbaum, and Damiano Mazza (2015). "A Strong Distillery". In: *APLAS 2015*, pp. 231–250. URL: http://dx.doi.org/10.1007/978-3-319-26529-2_13.

Accattoli, Beniamino and Claudio Sacerdoti Coen (2015). "On the Relative Usefulness of Fireballs". In: *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pp. 141–155. DOI: 10.1109/LICS.2015.23.

Accattoli, Beniamino (2016). "The Useful MAM, a Reasonable Implementation of the Strong λ-Calculus". In: *WoLLIC 2016*, pp. 1–21. DOI: 10.1007/978-3-662-52921-8_1.

Accattoli, Beniamino and Giulio Guerrieri (2016). "Open Call-by-Value". In: *APLAS 2016*, pp. 206–226. DOI: 10.1007/978-3-319-47958-3_12. URL: http://dx.doi.org/10.1007/978-3-319-47958-3_12.

Accattoli, Beniamino and Bruno Barras (2017). "Environments and the complexity of abstract machines". In: *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017*, pp. 4–16. DOI: 10.1145/3131851.3131855.

Accattoli, Beniamino and Giulio Guerrieri (2017). "Implementing Open Call-by-Value". In: *Fundamentals of Software Engineering - 7th International Conference, FSEN 2017, Tehran, Iran, April 26-28, 2017, Revised Selected Papers*, pp. 1–19. DOI: 10.1007/978-3-319-68972-2_1.

Balabonski, Thibaut et al. (2017). "Foundations of strong call by need". In: *PACMPL* 1.ICFP, 20:1–20:29. DOI: 10.1145/3110264. URL: http://doi.acm.org/10.1145/3110264.

Accattoli, Beniamino (2018). "Proof Nets and the Linear Substitution Calculus". In: *Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings*, pp. 37–61. DOI: 10.1007/978-3-030-02508-3_3. URL: https://doi.org/10.1007/978-3-030-02508-3_3.

Accattoli, Beniamino and Giulio Guerrieri (2018). "Types of Fireballs". In: *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018*. Vol. 11275. Lecture Notes in Computer Science. Springer, pp. 45–66. DOI: 10.1007/978-3-030-02768-1_3.

Accattoli, Beniamino, Claudia Faggian, and Giulio Guerrieri (2019). "Factorization and Normalization, Essentially". In: *CoRR* abs/1908.11289. arXiv: 1908.11289.

Accattoli, Beniamino et al. (2019a). "Crumbling Abstract Machines". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 4:1–4:15. ISBN: 978-1-4503-7249-7. DOI: 10.1145/3354166.3354169.

Accattoli, Beniamino et al. (2019b). "Crumbling Abstract Machines". In: *CoRR* abs/1907.06057. arXiv: 1907.06057.

Condoluci, Andrea, Beniamino Accattoli, and Claudio Sacerdoti Coen (2019). "Sharing Equality is Linear". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 9:1–9:14. ISBN: 978-1-4503-7249-7. DOI: 10.1145/3354166.3354174.

Condoluci, Andrea, Beniamino Accattoli, and Claudio Sacerdoti Coen (2019). "Sharing Equality is Linear". In: *arXiv e-prints*. arXiv: 1907.06101.

Komendantskaya, Ekaterina, ed. (2019). *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. ACM. ISBN: 978-1-4503-7249-7. DOI: 10.1145/3354166.

List of Symbols

$\lvert t\rvert$	size of λ-term t 17
$e(x)$	lookup in environment 47
σ_e	substitution of environment 50
$\mathtt{index}(\mathsf{l} \mid \tau : \mathsf{n})_G$	de Bruijn index according to a access path 79
$[\![\tau : \mathsf{r} \rightsquigarrow \mathsf{n}]\!]_G$	readback of access path 79
$@$	appending 96
$\underline{t}$	crumbling translation of t 97
$c_{\downarrow}$	readback of crumble c 98
$\perp$	disjoint 100
$\lvert c\rvert$	size of crumbled form 107
$\iota(\bullet)$	embedding of crumbles into pointed environments 140
$(\bullet)_{\Downarrow}$	readback from pointed environments to crumbles 140
$\mathcal{Q}$	query 164
$\mathcal{R}^{\Downarrow}$	propagation of $\mathcal{R}$ 168
$\mathcal{R}^{\#}$	spreading of $\mathcal{R}$ 172
$=_{\mathsf{c}}$	same canonic 198
$\prec$	staircase order 203

List of Figures