Alma Mater Studiorum — Università di Bologna
Dottorato di Ricerca in Informatica, Ciclo XXVI
Settore Concorsuale di afferenza: 01/B1 Settore Scientifico disciplinare: INF/01

Université Paris Diderot — Paris 7
École Doctorale Sciences Mathématiques de Paris Centre
Thèse de doctorat Spécialité informatique

CERTIFICATES FOR INCREMENTAL TYPE CHECKING

Matthias Puech

April 2013

Coordinatore Dottorato:
Maurizio Gabbrielli

Relatori:
Andrea Asperti
Hugo Herbelin

Esame finale anno 2013

[...] *mais il me semble que je ne pourrai jamais saisir précisément cette image, qu'elle est pour moi un au-delà de l'écriture, un «pourquoi j'écris» auquel je ne peux répondre qu'en écrivant, différant sans cesse l'instant même où, cessant d'écrire, cette image deviendrait visible, comme un puzzle inexorablement achevé.*

Georges Perec, Penser/Classer

SUMMARY

The central topic of this thesis is the study of algorithms for type checking, both from the programming language and from the proof-theoretic point of view. A type checking algorithm takes a program or a proof, represented as a syntactical object, and checks its validity with respect to a specification or a statement. It is a central piece of compilers and proof assistants. We postulate that since type checkers are at the interface between proof theory and program theory, their study can let these two fields mutually enrich each other. We argue by two main instances: first, starting from the problem of proof reuse, we develop an incremental type checker; secondly, starting from a type checking program, we evidence a novel correspondence between natural deduction and the sequent calculus.

We begin in Chapter 1 by presenting what will be the main tool on which relies our work: the LF metalanguage [Harper et al., 1993]. It is a generic and expressive notation for proofs, based on the central notion of *hypothetical judgment*, that can serve as a *data structure* to represent encoded proofs. We first introduce it informally, and then formally define it and develop its metatheory. The contributions of this introductory chapter are:

- a self-contained presentation of SLF, a spine-form variant of LF, based on Martin-Löf [1996]'s analysis of the notion of judgment;
- the formal development of its metatheory with some novel aspects

In Chapter 2, we are interested in *proof certificates*. A proof certificate is the result (or byproduct) of a computation which validity with respect to a specification can be communicated and checked independently. It allows the trust we put on complex computation's results to depend only on a small verification program. In particular, we study the generation and manipulation of proof certificates in LF. This study leads to the design of a framework to easily develop certifying programs. Its main contributions are:

- the formal description of DLF, an extension of LF designed to write certifying programs and check the produced proof certificates;

- within this context, a novel proposition to manipulate open terms: the "*environment-free*" programming style, syntactically supported by *function inverses*;
- the description of a lightweight framework for computing certificates, implemented as an OCaml library, and tested on some documented, full-scale examples.

Relying on the grounds of Chapter 2, Chapter 3 shows how the idea of providing certificates for a typing judgment (certifying type checking) can be extended to a mechanism for producing these certificates incrementally. Practically, this makes possible a form of *incremental type checking* by reuse of previously-computed subderivations. Given the certifying implementation of a type checker for a chosen language, the user can provide in turn program *deltas* representing modifications on a large program: the well-typing of the whole program will be certified by verifying only the modified parts. The contributions of this chapter are:

- a conservative extension of Chapter 1's SLF allowing to share and reuse pieces of SLF objects, based on a subset of Contextual Modal Type Theory [Nanevski et al., 2008] and giving a formal account of *memoization* in a higher-order setting where data contains binders and substitution is built-in;
- a conservative extension of Chapter 2's DLF which allows turn any certifying type checker into an incremental one;
- the description of its implementation as an OCaml library.

Finally, we depart from proof certificates in Chapter 4 to present a proof-theoretic result, byproduct of the work on type checking algorithms. Using and extending tools developed by Danvy and Nielsen [2001], we bring to light a functional correspondence between intuitionistic natural deduction NJ and (a variant of) the intuitionistic sequent calculus LJ. Starting from the functional implementation of a type checker for the extended λ-calculus and using off-the-shelf program transformations, we show that it can be transformed into a type checker for a well-known sequent calculus-style term assignment system (LJT, Herbelin [1994]). The contributions of this chapter are:

- a systematic transformation of natural deduction-style type checkers into sequent calculus-style ones, bringing out the relative natures of eliminations and right rules;
- the development of a general notion of data structure *reversal* and the corresponding semantics-preserving program transformations, which allows to derive, from a recursive program, the composition of a reversal function and an accumulator-passing style program;
- a novel proof of completeness of LJT with respect to NJ through this transformation.

REMERCIEMENTS

Pour donner à cette page la sincérité que j'aimerais qu'elle témoigne, j'invoque temporairement ma langue maternelle, le français.

De façon quelque peu non conventionnelle, je voudrais avant tout adresser ma profonde reconnaissance à Yann Régis-Gianas, qui m'a accompagné depuis près de trois ans et a suivi la majorité de mon travail de thèse. C'est notre fascination commune pour la "belle programmation" qui a élevé irrésistiblement nos discussions informelles sur le typage et la gestion de versions en une collaboration suivie, puis en la supervision qui a mené à cette thèse. J'espère que cette expérience d'encadrement, qui n'a su se nommer, lui aura apporté autant de satisfaction que j'ai eu de plaisir à découvrir son enthousiasme et sa curiosité intellectuelle.

Je remercie bien sûr Andrea Asperti pour m'avoir accueilli dans son laboratoire et pour la confiance qu'il m'a témoigné en me permettant de mener à bien cette thèse. Je remercie également Hugo Herbelin qui a accepté de la co-encadrer après le stage de Master que j'ai effectué sous sa direction attentive, et avec qui les échanges scientifiques, même quand ils sont ponctuels, sont toujours extrêmement riches et m'ont appris qu'en matière de recherche, une interrogation bien formulée vaut mille affirmations.

Je suis très reconnaissant envers les deux rapporteurs de ce manuscrit, Olivier Danvy et Dale Miller, d'avoir accepté cette responsabilité, mais surtout de l'intérêt qu'ils ont porté à mon travail bien avant l'existence de ce manuscrit, de leurs encouragements, de la minutie de leur lecture et des commentaires précieux qu'ils m'ont apporté.

L'environnement scientifique est le liant d'une thèse, et son exhausteur de goût. Qu'auraient été ces quatre années sans la stimulation intellectuelle et l'émulation continue avec mes passionnés et passionnants collègues? Je remercie tous les membres de l'équipe πr^2 et du laboratoire PPS, mon *alma mater*, particulièrement les doctorants de toutes les générations. De l'autre côté des Alpes, merci aux membres du DISI, en particulier Simone Martini et Maurizio Gabbrielli pour avoir assuré la coordination du programme de doctorat.

À mes exceptionnels amis, je dois l'inconditionnel soutien et la richesse du partage sans lequel je ne serais certainement pas arrivé au bout de cette aventure.

À ma chère famille, mes parents, ma soeur, pour tout leur amour.

Enfin, à Anne-Louise: ces années de thèse ne sont qu'un chapitre d'une aventure qui elle, reste entièrement à écrire.

Paris, le 15 mars 2013

CONTENTS

INTRODUCTION

If Computing Science is the study of *computation*, it is just as much the study of *knowledge*. Computation, understood as the external process of producing information, was familiar to the human being long before the computer was invented: from numeric calculations for trading and bookkeeping to mathematical models for predicting physical phenomena, systematic methods to compute have been a *tool* to humans alongside the hand axe and the plow. Computing efficiently, without appealing to too much external tools, developing simple methods for computing complex results, describing them concisely and unambiguously... These are obviously the sought goal of every computer scientist of former and current days.

The essence of an effective method of computation, or *algorithm*, is that it is devised once, as a finite sequence of "steps", but can be reused indefinitely in different situations, or with different input: the procedure for adding two numbers m and n by counting on your fingers—start with a closed fist, raise m fingers, then raise n fingers, then count how many fingers you have raised—works as well for computing $2+2$ than for computing $3+5$. The procedure itself can then be conveyed to another person which will be able in turn to add any two numbers. But now a second, more efficient algorithm may be taught to compute addition, for instance by organizing them in a positional numbering system, and adding them digit-by-digit with a carry:

$$\begin{array}{rr} & {\scriptstyle 1} \\ & 1\,2\,3 \\ + & 3\,9 \\ \hline & 1\,6\,2 \end{array}$$

Surely, it is faster, and requires less space (or fewer fingers) to compute the addition of two large numbers. But is this new addition *the same* as the first one? Does it *always* compute the same result, for any two input numbers? This is a legitimate question for a skeptical pupil who just learned this method. She can try on some examples (2, 10, 100) to compute additions with the first and second method and compare the results, but nothing would prevent the 101th test to fail giving the same

result. After having explained the algorithm, how can the teacher, who long ago comprehended it, convey the idea that it really captures the notion of addition we know? How can one *experience the truth* of such an assertion, and how can this *acquired knowledge* be transmitted so as to convince another skeptical person? Is it even conceivable to convey such a "belief" unerringly?

These epistemic questions, who interested the philosophers since Aristotle, are put vastly at stress by the advent of the computer, and the use of it we make today. The computer is, as its name implies, the "ultimate" external computing device: its one and only purpose is to take any algorithm, provided that it is explicit enough to be formulated in a fixed language (the *program*), any input data and execute the program step-by-step on the data. The algorithm can be arbitrarily complex for a human being to understand, the data arbitrarily large, no questions will be asked; yet, how can the user of the computer be *convinced* that the result of a computation is conform to its expectations?

Imagine a large spreadsheet of thousands lines of data, cell values depending heavily on each other by complex formulae. Now imagine changing the value of one cell on line one. The result we expect from that change is that all cells will take the same values as if we had entered the *new* value in place of the *old* when we constructed the spreadsheet in the first place. Instead, in that case, the algorithm that is triggered by the spreadsheet program is to invalidate all cells which value depend on the changed one, and recompute only these by propagating the changes; this way, the computer is spared the costly recomputation of already computed values. Of course, the corresponding program is expressed in much more detail, probably thousands of lines of program code. Will this complex program always behave as if we had reentered by hand the whole spreadsheet again? Is it conform to its *specification*? Or does it have a *bug*? If this spreadsheet is used to compute our yearly taxes, it better not! But if it is used to compute the trajectory of a space rocket carrying people, then it *must* not. On a less dramatic yet equally critical note, each time we view a web page, code is sent over from the site and executed on our very computer. This piece of program has been written by someone we do not know and cannot trust; she could have intentionally exploited a security hole in our browser to retrieve our personal data, or a bug may have crept into that code that makes our computer unusable for an annoying couple of seconds. How can we be *convinced*, how can we *know* for sure, that a program fulfills its specification?

This thesis explores several aspects of the relationship between a program and its specification, under the lens and resting upon the solid foundations of *proof theory*. It is the discipline of mathematics and computer science that has for object proofs. Proofs is intended here not only as mathematical proofs, but more generally as *evidences* in the juridical sense, as any process able to convince (a judge, a user, a skeptical pupil) of a certain statement. Like a legal evidence, it can be direct—

immediate, or through a series of steps—mediate. In Per Martin-Löf's words:

> Thus, if proof theory is construed, not in Hilbert's sense, as metamathematics, but simply as the study of proofs in the original sense of the word, then proof theory is the same as theory of knowledge, which, in turn, is the same as logic in the original sense of the word, as the study of reasoning, or proof, not as metamathematics.
>
> [Martin-Löf, 1996]

We shall convince the reader that it is this very object, the proof, that bridges the gap between specifications and programs, and moreover in a very powerful way: if given a syntactical incarnation, it constitutes an indisputable evidence of the truth of a statement that can be communicated to convey *knowledge* about programs and computations.

One could object right away that our two instances of proofs—a proof about a computation method that the pupil may have the curiosity to investigate, and a proof that large and complex program respects its specification, and that the user want to trust—are of a very different nature. In the first case, its purpose is to convey to another human being the *understanding* of a certain object; with that understanding, she might adapt the method and intuit how to add two binary numbers for instance. The second instance is in a sense much more mundane: the user of a spreadsheet program has no intention of understanding the inner workings of her program, nor use that knowledge to build a new, enhanced one (at least, most users do not). Her intentions are simply to be *ensured* of its result. Will she read the proof herself to eventually be convinced that it computes well? If the proof is such an indisputable evidence of this fact, could she not rather let her computer itself verify the proof? If she trusts the program that verifies the proof, and if this program says that a proof is correct, then she will know that the proof is indeed correct. Such a proof that is written to convince with such a fine reasoning granularity that it is verifiable by a computer is a *formal proof*, and such a verification program for formal proofs is called a *proof checker*, or, in the community of *type theory* to which the author belongs, a *type checker*.

But let us go back to the nature of a proof, and how it can be represented in a computer. There are many different views on what constitutes an indisputable evidence, and even many formal languages in which to express the statement itself: first-order logic is the most well known, but intuitionistic, modal, temporal, linear logics are of utmost importance when carrying formal proofs about software. However, all share a common core: reasoning is made by atomic steps, or *inferences*, each step possibly involving already known facts; these facts are either built up themselves from other inferences or posed as hypotheses. They vary however by the set of valid

inferences they allow; fixing a set of inferences fixes the way we will be allowed to reason.

If I want to communicate a proof from a computer to another, that is if I want a proof to have a concrete existence in the computer's memory, I will have first to define a *data structure* for it, a *format*. Will I have to make a choice right away on which particular logic the proofs will need to be written in? This thesis begins by exposing such a syntactical notation for proofs in the first chapter. It has the particularity of not committing to any idiosyncratic way of reasoning, but instead to be logic agnostic: it is not a logic, it is a metalanguage for proofs, or *logical framework* (hence its name, LF). It fixes only a convenient notation, close to the λ-calculus, for the small reasoning core common to all logics; a proof written in this format, or *proof object* will begin by stating its reasoning principle, its logic. How do I know that an LF proof is correct? With this data structure comes a *type checking* algorithm that describes how to validate (or not) a given proof object.

Contrarily to mathematical proofs, proofs about computer software are often very large and shallow: they do not involve sophisticated mathematical objects, but the number of cases to verify is usually huge. Will the programmer of the spreadsheet have to write the proof entirely by hand before sending it to the user? In the second chapter, we are interested in how to *generate* proof certificates. A proof certificate is an object that is output by a program together with its result, certifying that its result is correct with respect to a given specification. For instance, a programmer might work in a high-level programming language that ensures a certain safety property on the execution of its programs: they do not corrupt memory, they always output an even integer... The compiler of this programming language could issue a certificate stating that the code produced respects this property. How should we write such a *certifying* program? It turns out that general-purpose programming languages are not adapted to the task of writing programs that issue proofs: even if it is possible, it is hard because of the hypothetical nature of proofs. In this chapter, we will design a core programming language that has built-in facilities to deal with proof certificates.

A type checker is a piece of software taking a program or a proof, possibly very large, and simply answering whether it is correct or not. Yet, in mathematics like in computer science, the process of *elaboration* or discovery of a proof or program is much less linear: one does not write, in a single action, a large proof and finally submit it to the computer for approbation. Instead, the working mathematician or computer scientist constructs his objects by successive approximations, trial and error... This piece of software, the type checker, does however not allow this kind of interaction: it only awaits a final, definitive object. The third chapter of this document investigates the question of designing an *incremental type checker* able to accept pieces of unfinished proofs, refining previous versions of the same document. If the document is very large, it is important for this interaction to take place "in real

time": the user cannot wait for whole document to be re-validated at each change, but would like the computer to be able to only validate (or invalidate) the small increments building the whole edifice.

There are many different logics or reasoning styles that serve different, sometimes orthogonal purposes, but two of them occupy a very special place in the landscape of formal logics: *natural deduction* and *sequent calculus*. The inferences of natural deduction are known to reflect more faithfully the way mathematicians write (non-formal) proofs; the inferences of sequent calculus are known to to be more fine-grained, and better adapted to the automatic search of proofs by a computer. Ironically enough, these two represent the same "abstract" notion of evidence, they were devised in 1934 by the same logician, Gerhard Gentzen, and they were even presented in the very same article. So, what is the precise relationship between natural deduction and sequent calculus? Finally, in the last chapter of this thesis, we will draw from our experience of writing type checking programs and propose a partial answer to this question, seen from a computer scientist point of view. This last incursion into proof theory will eventually close the loop of our investigation of the nature of programs and their specification by turning upside down our very own presupposition: we embarked on the quest to explain programs with formal proofs, we will end up explaining a proof theoretic phenomenon by a program.

NOTATIONS AND CONVENTIONS

A | LEVELS OF METALANGUAGE

In this document, we will pervasively switch between levels of metalanguage, or universes, that one ought to differentiate clearly before beginning.

The topmost one, the *level of discourse*, is used to convey informally ideas to the reader; it is written in English or in mathematical notations, in black. For instance: "a set", "$1, 2 \in \mathbb{N} \times \mathbb{N}$", or "$\frac{\quad}{\quad}$" (an inference bar). Mathematical notation include all classical notions from basic set theory: set membership "$\_ \in \_$", intersection "$\_ \cap \_$", union "$\_ \cup \_$", comprehension "$\{\_ \mid \_\}$", function domain "dom(_)", syntactic equality "$\_ = \_$", function application "_(_)" ordered pair "_;_" (note this last, slight unusual notation).

Our main occupation will be, at the discourse level, to define inductive sets and recursive functions; elements of these sets belong to the *syntactic level*, and is also typeset in black, mathematical notations, for instance "$*$" or "$\cdot \vdash \cdot : \cdot$". The ambiguity is intentional: sometimes, mixing the discourse and the syntactic levels will help the understanding by avoiding excess of formalism. A notable exception are *keywords*, i.e., (English) words that have a special, syntactical meaning, and which will be typeset in upright bold blue to avoid confusion, for instance "**let**" or "**match**".

We call *discourse variables* any name or placeholder, bound at the discourse level, given for a syntactic object. They are named after the set they belong to. Discourse variables are typeset in dark blue italics, for instance "M", "Γ" or "σ"; they allow to form the discourse-level statements like "for all M, there exists a V such that $M \downarrow V$". Alternate names for discourse variables belonging to set X include X', X''... and X_0, X_1, ..., X_n. All free discourse variables are implicitly bound at the beginning of the statement.

Because this is a computer science thesis, there is another level of discourse that manipulates syntactic objects: the *computational level*. Programs, which will here be mostly written in the OCaml language, are typeset in brown, typewriter case, for instance "`rev_append`". There will thus be a clear distinction between the languages that are the *object* (defined, syntactic objects) and the *subject* (the metalanguage) of

the discourse.

B | SYNTAX

We will define inductive sets, or languages, by two different means. The first, and weakest, is by a *context-free grammar* in BNF form [Backus, 1959], for instance:

$$T \ ::= \ \cdot \ \Big| \ T ; T \ \Big| \ \pi_T(A)$$

When it can be disambiguated from context, we will use the name of the non-terminal (here T) to refer both to the inductive set and a particular element of this set.

A notable degree of freedom we take from BNF grammars is the use of names. A *name* is an element of a predefined set of infinite cardinality on which equality is decidable. Typographic appearance and color determines the name set to which a name belongs; for instance, a is different from a. *Constants* denote names that are not bound in an expression; they are typeset in a sans-serif color font, for instance "Cons" or "tp". By convention, dark green constants denote *type* constants and dark red constants denote *term* or *object* constants. *Variables* denote names that are bound in the same expression. They are typeset in dark teal italics, for instance "x" or "acc". This allows to define the set of λ-terms M with constants:

$$M, N \ ::= \ x \ \Big| \ \mathsf{c} \ \Big| \ M \ N \ \Big| \ \lambda x. M$$

Variable bindings will always be followed by a period "." or a comma ",". Very often, binders will be the last argument of an application spine, because we will use them *à la* Church [1940]. For concision, we take a slightly unusual priority convention: λ-abstraction body will not need parentheses when they are the last argument of an application spine. For instance the term $\mathsf{lam}\ \lambda x. M\ N$ will be thus implicitly understood as $\mathsf{lam}\ (\lambda x. M\ N)$. We adopt Barendregt [1992]'s convention: any two terms differing only by the name of their bound variables will be considered equal with respect to to the mathematical equality $=$ (α-equivalence). We also suppose defined the set $\mathsf{FV}(M)$ of all free variables of a term M.

Finally, *functions* defined syntactically will have their own name set, typeset in bold red italics, for instance "*infer*".

C | LISTS

We distinguish a particular family of inductive definitions, lists. A *list* is any inductive set L that can be expressed by a context-free grammar with all right-hand sides of L containing *at most one* occurrence of L. Rules containing one are called *cons* cells, and rule containing zero are called *nil* cells. All other non-terminals are called its *elements*. Often, lists will have one *nil* cell written "$\cdot$" and one *cons* cell written "X, L"

where X is the element. We suppose known a set of common decidable predicates and operations on lists and their properties: length "$|L|$", reverse "rev(L)", access to the n-th element "nth(n, L)" for $n \in \mathbb{N}$. Besides, we will allow ourselves an implicit coercion from lists to multisets, so that "$X \in L$" is a valid and decidable predicate if L is a list and X is an element of Y. If clear from context, we allow the terminals of the *nil* cells to be eluded.

D | INFERENCES AND FUNCTIONS

The second, most general kind of inductive definition are *inferences*. We call a *judgment form* a family of sets indexed by one or more syntactic entities; for instance $\vdash A$ is a judgment form on propositions. A *judgment* is an instantiated judgment form, for instance $\vdash \mathsf{p} \supset \mathsf{q}$. A judgment form is defined inductively by a finite set of *inference rules*: they are syntactic constructs mapping a finite set of judgments, the *premises*, to the judgment they define, the *conclusion*. Each rule has a name, typeset in small capitals. For instance, the *modus ponens* inference rule is written:

$$\frac{\vdash A \supset B \qquad \vdash A}{\vdash B}\ \textsc{ImpE}$$

An element of a set defined by inference rules is called a *derivation*. We say that a derivation *judges* the instance of a judgment if it is the instance of an inference rule and if each premise itself judges the corresponding instantiated judgment. A judgment is *inhabited* if there is a derivation judging it. A judgment will be implicitly coerced to a mathematical proposition at the discourse level, meaning that it is inhabited; we will say for instance "if $\vdash A$ then $\vdash A \wedge A$.

We give the syntactic shorthand to define recursively *functions* $f : X_1 \to X_2 \to \ldots \to X_n$. Recursive definition is only a convenient notation for the inductive definition of a judgment $f(X_1, \ldots, X_{n-1}) = X_n$ that is decidable and deterministic. For instance, given a syntactic definition of Peano naturals, the addition function definition:

$$\begin{aligned} \mathsf{0} + n &= n \\ \mathsf{S}(m) + n &= \mathsf{S}(m + n) \end{aligned}$$

is shorthand for the two inference rules:

$$\frac{}{\mathsf{0} + n = n} \qquad \frac{m + n = p}{\mathsf{S}(m) + n = \mathsf{S}(p)}$$

and the intuitionistic property that for each naturals m and n, either there exists a unique p and a derivation judging $m+n = p$, or there is no such p. It is a *total function*

if the first case always holds. We then simply write $m + n$ to refer to the natural p. We say that we reason by *functional induction* on the computation of $f(X_1, \ldots, X_{n-1})$ to mean that we reason by induction over the judgment of $f(X_1, \ldots, X_{n-1}) = X_n$.

1 | THE LF NOTATION FOR PROOFS

¶ | ABSTRACT How can we represent proofs—any proof in any logic—in a computer? What data structure should we use to represent and manipulate proofs in memory? In this chapter, we commit to a particularly elegant representation, the logical framework LF, that will be the basis our work in the next two chapters. This chapter consists in two parts. In the first, we introduce informally this notation in our terms, not relying on any prerequisites: it should constitute a gentle introduction to type theory, and how it corresponds to a *hypothetical notation* for proofs. In the second part, we present, formally this time, a variant of LF called SLF, and show its properties.

1.1 | PRESENTATION

In the tradition emerged from the seminal work of Frege, and Gentzen at the beginning of the XXth century [Frege, 1879, Gentzen, 1935], a proof is considered to be a tree, its leaves being hypotheses, and each of its internal nodes being a correct deduction, from the n premises represented by the n subtrees children of that node, to the conclusion—the node itself. The breakthrough of this idea is that the notion of deduction depends here on the logic we choose to reason into (classical, intuitionistic, modal...) and is not fixed in advance: the representation of deductions as trees is agnostic from the particular logic used.

Many years later, Per Martin-Löf sets the basis of a *metalanguage*, inspired by the work of Church on the λ-calculus [Church, 1940] and a methodical analysis of the concept of *judgment* inherited from analytical philosophy: *Intuitionistic type theory* [Martin-Löf and Sambin, 1984]. In this metalanguage, one can encode many common logics (then called *object logics*) by providing their deduction rules, encoded as *typed constants*, and write proofs in these logics, encoded as *typed terms*. If the encoding the logic is correct, then the validity or *well-typing* of a term encoding a proof implies the validity of that proof.

Harper et al. [1993] propose a physical realization of intuitionistic type theory in the form of *logical framework* LF. LF is a language for representing proof systems (set

of inference rules) and derivations. It features a built-in notion of *hypothetical reasoning* which facilitates greatly the formalization of languages and logics supporting this kind of reasoning. It is also the basis of Twelf [Pfenning and Schürmann, 1999], a popular interactive theorem prover specialized in the formalization of programming and proof languages.

We present this notation for proofs and discuss its advantages as a basis for mechanically-checked proof certificates.

1.1.1 | FIRST-ORDER TERM REPRESENTATION

A | OBJECTS AND TYPES

Each node of a proof tree corresponds to the application of a rule, taken in the set of inference rule of the logic. For instance, the particular proof tree $\mathscr{D}$:

$$\dfrac{\begin{array}{cc}\mathscr{D}_1 & \mathscr{D}_2\\ \vdash \mathsf{p}\wedge\mathsf{p}' & \vdash \mathsf{q}\end{array}}{\vdash (\mathsf{p}\wedge\mathsf{p}')\wedge\mathsf{q}}$$

ends with an instance of the rule:

$$\dfrac{\begin{array}{cc}\vdash A & \vdash B\end{array}}{\vdash A\wedge B}\ \textsc{ConjI}$$

of NJ, the logic of natural deductions [Gentzen, 1935], and is thus correct. To testify that it is, we can provide the instantiation of the metavariables A and B appearing in the rule ConjI: $A/\mathsf{p}\wedge\mathsf{p}', B/\mathsf{q}$. Then, the verification amounts to instantiate the rule with the substitution, verify that the conclusion judgment is syntactically equal to the instantiated form, and that recursively the subtrees verify their respective instantiated judgments.[1] A convenient notation $[\![\mathscr{D}]\!]$ of this tree as a *first-order term* could thus be:

$$\left[\!\!\left[\ \dfrac{\begin{array}{cc}\mathscr{D}_1 & \mathscr{D}_2\\ \vdash \mathsf{p}\wedge\mathsf{p}' & \vdash \mathsf{q}\end{array}}{\vdash (\mathsf{p}\wedge\mathsf{p}')\wedge\mathsf{q}}\ \right]\!\!\right] = \mathsf{ConjI}[A/\mathsf{p}\wedge\mathsf{p}', B/\mathsf{q}]([\![\mathscr{D}_1]\!], [\![\mathscr{D}_2]\!]) \tag{1.1}$$

or more simply, omitting the names of metavariables A and B:

$$= \mathsf{ConjI}(\mathsf{p}\wedge\mathsf{p}', \mathsf{q}, [\![\mathscr{D}_1]\!], [\![\mathscr{D}_2]\!]) \tag{1.2}$$

[1] Note that knowing only the rule we applied, the respective judgments proved by the subtrees $\mathscr{D}_1$ and $\mathscr{D}_2$ and the substitution to apply to the rule is enough to conclude that the first node of $\mathscr{D}$ is a correct deduction of the judgment $\vdash (\mathsf{p}\wedge\mathsf{p}')\wedge\mathsf{q}$: there is no need to know neither the actual content of $\mathscr{D}_1$ and $\mathscr{D}_2$, nor the exact judgment proved, which can be deduced by substitution. This will be of crucial importance in Chapter 3

Propositions are themselves first-order terms, so they are easily encoded by themselves.[2] We introduce constant ConjI, which applied to its arguments expresses that if $[\![\mathscr{D}_1]\!]$ (*resp.* $[\![\mathscr{D}_2]\!]$) is an evidence of the judgment $\vdash \mathsf{p} \wedge \mathsf{p}'$ (*resp.* $\vdash \mathsf{q}$), then it is itself an evidence of the judgment $\vdash (\mathsf{p} \wedge \mathsf{p}') \wedge \mathsf{q}$. We call *object* terms that are evidences of judgments. To encode judgment forms, let us introduce another constant is which, applied to an encoded formula, forms a valid judgment:

$$[\![\vdash A]\!] = \mathsf{is}([\![A]\!]) \tag{1.3}$$

We call *types* encodings of judgments (e.g., $\mathsf{is}((\mathsf{p}\wedge\mathsf{p}')\wedge\mathsf{q})$), and *type families* encodings of judgment forms (e.g., is alone). The *classification* relation between objects and types is pronounced "has type" and written:

$$\mathsf{ConjI}(\mathsf{p} \wedge \mathsf{p}', \mathsf{q}, [\![\mathscr{D}_1]\!], [\![\mathscr{D}_2]\!]) \ : \ \mathsf{is}((\mathsf{p} \wedge \mathsf{p}') \wedge \mathsf{q})$$

B | OBJECT CONSTANTS

To be able to verify this relation, we need to declare its constants. The declaration of constant ConjI should encode rule ConjI. For this we assign it a type family. It expects four arguments, so it should be a function type. Notice that the value of its first two arguments—the substitution part—determines the judgment of derivations (the type of objects) expected in place of the last two arguments. This dependency from *object* to *type* is expressed in the syntax of type families by a special function type construct, *dependent product* $\Pi x : A.B$. An object classified by type family $\Pi x : A.B$ is an applicable object with a codomain B dependent on the value of its argument of type A. If applied to an object M of type A, it will have type $B[x/M]$. Note that in $\Pi x : A.B$, x binds in B; if x does not appear free in B, we write simply $A \to B$, recovering the notation for the usual function type. The rule ConjI is encoded as a constant ConjI with type:

$$\mathsf{ConjI} \ : \ \Pi x : \mathsf{prop}.\, \Pi y : \mathsf{prop}.\, \mathsf{is}(x) \to \mathsf{is}(y) \to \mathsf{is}(x \wedge y)$$

The first two (dependent) arguments of ConjI must be of type prop. Constant prop, unlike is, forms a type by itself: it does not have to be applied. It corresponds to the judgment for an object to be a proposition. The last two arguments of ConjI are given the types $\mathsf{is}(x)$ and $\mathsf{is}(y)$, corresponding respectively to the judgments $\vdash A$ and $\vdash B$ in the premises of the rule.[3] Similarly for the logical constants, we assign them a

[2] We write them in infix notation for clarity

[3] The metavariables A, B become variables x, y in the metalanguage.

type family to declare them:[4]

$$(\wedge) \ : \ \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}$$

C | TYPE CONSTANTS

As for object constants (i.e., encodings of rules), type constants (i.e., encodings of judgment forms), must also be declared. We write:

$$\mathsf{prop} \ : \ *$$

to declare type prop. What about is, a constant that will form a valid type, but only applied to a prop? Let us use the same functional notations as before:

$$\mathsf{is} \ : \ \mathsf{prop} \to *$$

so that the application $\mathsf{is}(\mathsf{p} \wedge \mathsf{q})$ forms a valid type. What is the nature of the expression on the right of the colon? It is not a type, because it does not code for a logical rule or syntactical construct: let us call it a *kind* K. Therefore these declarations are very different from the one of ConjI: this time the relation, written $A : K$, reads "A is a type family classified by kind K". We can say that $*$ is the kind of types: only when we know that a certain type family A is classified by $*$ is it correct to state the classification $M : A$ for a given term M.

D | SUMMARY

So far, we constructed an encoding of proof trees into first-order terms, and we hinted at a simple way to verify them. Given a *signature*, i.e., a set of object and type constant declarations, we can check if a given object is a valid derivation in this signature. Let us sum up what we gathered until now:

- ☐ A derivation or syntactic object is encoded by an object.
- ☐ A syntactic or logical rule is encoded by a object constant declaration.
- ☐ A judgment form is encoded by a type family.
- ☐ A judgment or syntactic category is encoded by a type.
- ☐ A type is a type family classified by the kind $*$.
- ☐ An evident judgment, that is a judgment together with its justification, is encoded by a typing relation, that is an object together with its type.

[4] This highlights the conceptual similarity between *(i)* declaring a judgment and the logical rules justifying it on one side, and *(ii)* a syntactic category and its constructors on the other: the abstract syntax of a language is the characterization of the property of being a term of that language. For a term to be of type prop expresses the judgment that it is a proposition, the same as for a term to be of type $\mathsf{is}(\mathsf{p})$ expresses the fact that it is the evidence that $\vdash \mathsf{p}$.

1.1.2 | HIGHER-ORDER TERM REPRESENTATION

This encoding is sufficient to soundly code all languages and proof systems, but at the cost of a heavy treatment of *hypothetical judgments* [Martin-Löf and Sambin, 1984] . For instance, the rule of natural deduction

$$\dfrac{\begin{array}{c}[\vdash A]\\ \vdots \\ \vdash B\end{array}}{\vdash A \supset B}\ \textsc{ImpI}$$

expresses the fact that in the elided part of the proof (the " $\vdots$ "), a proof of A is available and can be used (or *discharged*) any number of times. We say that the judgment $\vdash B$ is *hypothetical* in the judgment $\vdash A$.

A | CUMBERSOME FIRST-ORDER ENCODING

If we want to encode such a hypothetical proof—a higher-order structure—in the previous first-order term syntax, we can syntactically adopt a local encoding: add an explicit hypothesis *environment* to the encoding of our judgment, which becomes $\Gamma \vdash A$, and thread it throughout the judgments until the leaves. An environment is a type containing lists of formulae; to discharge a particular list element, we need a new Var rule (sometimes called Init) and a look judgment to look up in this list. It gives the following signature:

$$\mathsf{env} : * \tag{1.4}$$
$$\mathsf{nil} : \mathsf{env} \tag{1.5}$$
$$(::) : \mathsf{prop} \to \mathsf{env} \to \mathsf{env} \tag{1.6}$$
$$\mathsf{look} : \mathsf{env} \to \mathsf{prop} \to * \tag{1.7}$$
$$\mathsf{LookO} : \Pi\gamma : \mathsf{env}.\, \mathsf{look}(x :: \gamma, x) \tag{1.8}$$
$$\mathsf{LookS} : \Pi\gamma : \mathsf{env}.\, \Pi xy : \mathsf{prop}.\, \mathsf{look}(x :: \gamma, y) \to \mathsf{look}(\gamma, y) \tag{1.9}$$
$$\mathsf{is} : \mathsf{env} \to \mathsf{prop} \to * \tag{1.10}$$
$$\mathsf{ImpI} : \Pi\gamma : \mathsf{env}.\, \Pi xy : \mathsf{prop}.\, \mathsf{is}(x :: \gamma, y) \to \mathsf{is}(\gamma, x \supset y) \tag{1.11}$$
$$\mathsf{ImpE} : \Pi\gamma : \mathsf{env}.\, \Pi xy : \mathsf{prop}.\, \mathsf{is}(\gamma, x \supset y) \to \mathsf{is}(\gamma, x) \to \mathsf{is}(\gamma, y) \tag{1.12}$$
$$\mathsf{Var} : \Pi\gamma : \mathsf{env}.\, \Pi x : \mathsf{prop}.\, \mathsf{look}(\gamma, x) \to \mathsf{is}(\gamma, x) \tag{1.13}$$

This is one among many choices of higher-order to first-order encoding that chooses to represent a hypothesis discharge as its position in the current list of hypotheses. This encoding is to the higher-order encoding presented below what De Bruijn indices [de Bruijn, 1972] are to the usual λ-calculus. It poses several problems.

First, it departs from the proof notation used to represent the system, leaving more space for errors or complicating the soundness proof; secondly and more importantly, it is not *adequate* [Harper and Licata, 2007]: there is no isomorphism between a proof and its encoding.

Example 1.1. The hypothetical proof

$$\frac{[\vdash \mathsf{p}] \qquad [\vdash \mathsf{q}]}{\vdash \mathsf{p} \wedge \mathsf{q}}\ \textsc{ConjI}$$

can be encoded as:

$$\begin{array}{l}\mathsf{ConjI}(\mathsf{p} :: \mathsf{q} :: \mathsf{nil}, \mathsf{p}, \mathsf{q},\\ \quad \mathsf{Var}(\mathsf{p} :: \mathsf{q} :: \mathsf{nil}, \mathsf{p}, \mathsf{LookO}(\mathsf{p} :: \mathsf{q} :: \mathsf{nil}, \mathsf{p})),\\ \quad \mathsf{Var}(\mathsf{p} :: \mathsf{q} :: \mathsf{nil}, \mathsf{q}, \mathsf{LookS}(\mathsf{p} :: \mathsf{q} :: \mathsf{nil}, \mathsf{p}, \mathsf{q}, \mathsf{LookO}(\mathsf{q} :: \mathsf{nil}, \mathsf{q}))))\end{array}$$

or in the reversed environment as:

$$\begin{array}{l}\mathsf{ConjI}(\mathsf{q} :: \mathsf{p} :: \mathsf{nil}, \mathsf{p}, \mathsf{q},\\ \quad \mathsf{Var}(\mathsf{q} :: \mathsf{p} :: \mathsf{nil}, \mathsf{p}, \mathsf{LookS}(\mathsf{p} :: \mathsf{q} :: \mathsf{nil}, \mathsf{p}, \mathsf{q}, \mathsf{LookO}(\mathsf{q} :: \mathsf{nil}, \mathsf{q}))),\\ \quad \mathsf{Var}(\mathsf{p} :: \mathsf{q} :: \mathsf{nil}, \mathsf{q}, \mathsf{LookO}(\mathsf{p} :: \mathsf{q} :: \mathsf{nil}, \mathsf{p})))\end{array}$$

In other words, the usual hypothetical notation for proofs makes a quotient on the actual local representation of proofs, assimilating all proofs that are equal modulo the so-called *structural rules*: the environment is implicit, and treated as a *set* and not as a list as in the example above. To go from the first to the second explicit proof is to observe this quotient as a metatheorem about the explicit system: weakening, permutation, etc. To achieve an hypothetical encoding that is adequate to proofs in hypothetical logics, we must extend the metalanguage beyond bare first-order terms.

B | λ-TREE NOTATION

To palliate the heaviness (or inadequacy) of local, first-order encodings, let us use a technique known as *higher-order abstract syntax* (HOAS) since Pfenning and Elliott [1988] or *λ-tree notation* [Miller and Palamidessi, 1999]. It actually dates back to Church [1940], who used it to encode the binding structure of quantifiers in its *simple theory of types*. Besides the application of constants, we introduce a λ construct into the terms of our notation, serving as a way to introduce (or *bind*) hypotheses by a "silent" variable name. To refer to these hypotheses, we extend the set of applicable names to these variables. The set of terms, or *(canonical) objects* is then defined by the following grammar:

$$\begin{array}{rcll} M & ::= & \lambda x.M \mid H\,(S) & \text{Canonical object} \\ H & ::= & x \mid \mathsf{c} & \text{Head} \\ S & ::= & \cdot \mid M, S & \text{Spine} \end{array}$$

An object can be a λ-abstraction, or the application of a constant or variable to a list of arguments being themselves objects; this list of arguments can be empty ($H\,(\cdot)$), in which case we write simply H. This syntax is isomorphic to the normal λ-calculus (see Chapter 4); for this reason, we allow to omit the parentheses and semicolons between arguments and write $H\ M_1\ M_2\ M_3$ for $H\,(M_1, M_2, M_3)$. The idea of λ-tree notation is to encode the charging of a hypothesis as a λ-abstraction, and its discharge as a variable; in essence, it amounts to use the built-in binding facility $\lambda x.\,M$ to represent the object logic's notion of binding, instead of coding it by hand as before. Let us study a couple of examples:

Example 1.2. The proofs

$$\dfrac{\dfrac{[\vdash \mathsf{p}] \quad [\vdash \mathsf{p}]}{\vdash \mathsf{p} \wedge \mathsf{p}}\ \textsc{ConjI}}{\vdash \mathsf{p} \supset \mathsf{p} \wedge \mathsf{p}}\ \textsc{ImpI} \qquad \text{and} \qquad \dfrac{\dfrac{[\vdash \mathsf{p}]}{\vdash \mathsf{q} \supset \mathsf{p}}\ \textsc{ImpI}}{\vdash \mathsf{p} \supset \mathsf{q} \supset \mathsf{p}}\ \textsc{ImpI}$$

are encoded respectively as the ground objects

$$\mathsf{Impl}\,(\mathsf{p}, \mathsf{conj}\,(\mathsf{p}, \mathsf{p}), \lambda x.\,\mathsf{ConjI}\,(\mathsf{p}, \mathsf{p}, x, x)) : \mathsf{is}\,(\mathsf{imp}\,(\mathsf{p}, \mathsf{conj}\,(\mathsf{p}, \mathsf{p})))$$
$$\mathsf{Impl}\,(\mathsf{p}, \mathsf{imp}\,(\mathsf{q}, \mathsf{p}), \lambda x.\,\mathsf{Impl}\,(\mathsf{q}, \mathsf{p}, \lambda y.\,x)) : \mathsf{is}\,(\mathsf{imp}\,(\mathsf{p}, \mathsf{imp}\,(\mathsf{q}, \mathsf{p})))$$

By convention, we assimilate all objects that differ only by the choice of variable names (they are quotiented by α-conversion, the so-called Barendregt convention). Provided we prove adequacy of this encoding, it indeed can rule out the possibility of several codes for the same proof. Quite naturally, the classifier of an object $\lambda x.\,M$ is a dependent product $\Pi x : A.\,B$; which dictates the final syntax of type families and kinds:

$$\begin{array}{rcll} K & ::= & \Pi x : A.\,K \mid * & \text{Kind} \\ A, B & ::= & \Pi x : A.\,B \mid P & \text{Type family} \\ P & ::= & \mathsf{a}\,(S) & \text{Atomic type} \end{array}$$

and the new signature for our propositional natural deduction on Figure 1.1.

Remark that domains of functional types can themselves be functional (higher-order types). This is used in rules introducing hypotheses (Impl and DisjE). A way to understand their types is to give them an informal game semantics; for instance the type of Impl as an instruction from the system to the user writing objects in this signature reads:

> "Impl: Given two proposition x and y, I'll give you a proof of x with which you should build a proof of y. If you succeed, you will have in your hands a proof of $\mathsf{imp}\,(x, y)$.",

$$
\begin{aligned}
\mathsf{prop} &: * \\
\mathsf{p} &: \mathsf{prop} \\
\mathsf{q} &: \mathsf{prop} \\
\mathsf{conj}\ \mathsf{disj}\ \mathsf{imp} &: \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop} \\
\mathsf{is} &: \mathsf{prop} \to * \\
\mathsf{ConjI} &: \Pi x : \mathsf{prop}.\, \Pi y : \mathsf{prop}.\, \mathsf{is}(x) \to \mathsf{is}(y) \to \mathsf{is}(\mathsf{conj}\,(x,y)) \\
\mathsf{ConjE1} &: \Pi x : \mathsf{prop}.\, \Pi y : \mathsf{prop}.\, \mathsf{is}(\mathsf{conj}\,(x,y)) \to \mathsf{is}(x) \\
\mathsf{ConjE2} &: \Pi x : \mathsf{prop}.\, \Pi y : \mathsf{prop}.\, \mathsf{is}(\mathsf{conj}\,(x,y)) \to \mathsf{is}(y) \\
\mathsf{ImpI} &: \Pi x : \mathsf{prop}.\, \Pi y : \mathsf{prop}.\, (\mathsf{is}(x) \to \mathsf{is}(y)) \to \mathsf{is}(\mathsf{imp}\,(x,y)) \\
\mathsf{ImpE} &: \Pi x : \mathsf{prop}.\, \Pi y : \mathsf{prop}.\, \mathsf{is}(\mathsf{imp}\,(x,y)) \to \mathsf{is}(x) \to \mathsf{is}(y) \\
\mathsf{DisjI1} &: \Pi x : \mathsf{prop}.\, \Pi y : \mathsf{prop}.\, \mathsf{is}(x) \to \mathsf{is}(\mathsf{disj}\,(x,y)) \\
\mathsf{DisjI2} &: \Pi x : \mathsf{prop}.\, \Pi y : \mathsf{prop}.\, \mathsf{is}(y) \to \mathsf{is}(\mathsf{disj}\,(x,y)) \\
\mathsf{DisjE} &: \Pi x : \mathsf{prop}.\, \Pi y : \mathsf{prop}.\, \Pi z : \mathsf{prop}.\, \mathsf{is}(\mathsf{disj}\,(x,y)) \to (\mathsf{is}(x) \to \mathsf{is}(z)) \to \\
&\qquad (\mathsf{is}(y) \to \mathsf{is}(z)) \to \mathsf{is}(z)
\end{aligned}
$$

Figure 1.1: LF signature for propositional NJ

where the user builds *objects*, and the system gives *variables*. Secondly, note that there is no Var rule as in the local encoding: the discharge of a hypothesis is handled by the metalanguage as the application of a variable. Finally, the use of λ-abstraction implies that checking the well-typing of an object will require to pass an *environment* Γ of all variables in scope:

$$\Gamma \ ::= \ \cdot \ \Big| \ \Gamma, x : A$$

This presentation of encoding propositional NJ is now over. The LF methodology suggests to prove that there is *adequacy* between the paper notation and its encoding. We skip the detailed proof, which can be adapted from Harper and Licata [2007].

Theorem 1.1 (Adequacy for propositions)**.** *There is a compositional isomorphism between propositions A of* NJ *and* LF *objects M of type* prop*.*

Theorem 1.2 (Adequacy for proofs)**.** *There is a compositional isomorphism between proofs of a proposition A of* NJ *and* LF *objects of type* $\mathsf{is}(M)$ *where M is the encoding of proposition A.*

Adequacy is a paper proof, showing that the particular encoding of your logic of choice is fully and faithfully represented in LF. If a logic is adequately represented

in LF, then any well-typed LF objects of the right type will correspond to a unique pen-and-paper proof of your logic, and conversely.

C | SUBSTITUTIONS IN TYPES

We hinted at the beginning of this presentation that the verification of well-typing of an encoded proof requires to *(i)* look up the expected type of the head constant (or variable) *(ii)* check recursively if the type of its first argument equals its expected type *(iii)* if it is a "substitution argument", substitute its value in the rest of the expected type *(iv)* continue with the second argument etc. This is precisely what we will describe in the next section, but let us first see how this substitution is defined. The problem is that the grammar given above for canonical objects is not stable by substitution: textually replacing variable x by M in e.g., $x\,(N)$ leads in general to the ill-formed object $M\,(N)$. In the running example of our signature (Figure 1.1), variables always appear in argument position, i.e., as objects, so that textual substitution results in a grammatically valid object (for instance, the expected type of M in $\mathsf{DisjI1}\,(\mathsf{conj}\,(\mathsf{p},\mathsf{q}),\mathsf{p},M)$ is $(\mathsf{is}\,(x))[x/\mathsf{conj}\,(\mathsf{p},\mathsf{q})] = \mathsf{is}\,(\mathsf{conj}\,(\mathsf{p},\mathsf{q}))$). But it is not always the case: consider the addition of a universal quantifier to the signature. We need to add terms, and encode the rules of $\forall$:

$$
\begin{aligned}
\mathsf{term} &: * \\
\mathsf{f} &: \mathsf{term} \to \mathsf{term} \to \mathsf{term} \\
\mathsf{a} &: \mathsf{term} \\
\mathsf{p} &: \mathsf{term} \to \mathsf{prop} \\
\mathsf{forall} &: (\mathsf{term} \to \mathsf{prop}) \to \mathsf{prop} \\
\mathsf{AllE} &: \Pi f : \mathsf{term} \to \mathsf{prop}.\, \Pi x : \mathsf{term}.\, \mathsf{is}\,(\mathsf{forall}\,(\lambda y.f\,(y))) \to \mathsf{is}\,(f\,(x)) \\
\mathsf{AllI} &: \Pi f : \mathsf{term} \to \mathsf{prop}.\, (\Pi x : \mathsf{term}.\, \mathsf{is}\,(f\,(x))) \to \mathsf{is}\,(\mathsf{forall}\,(\lambda y.f\,(y)))
\end{aligned}
$$

We add a syntactic class of terms, two symbols f and a, and a predicate p taking a term to a proposition. The syntactic construct forall is of a higher-order type: it provides a term—a variable x acting as a "hole"—to construct a proposition with holes. This is an example of the binding structure in a *term*, handled by the metalanguage.[5] For instance, the proposition $\forall x.\,\mathsf{p}(x)$ is encoded into $\mathsf{forall}\,(\lambda x.\,\mathsf{p}\,(x))$ (as in Church's simple type theory). Observe the type of AllE: the variable f appears in function position; thus its textual substitution by an actual "holey" proposition like $\lambda x.f\,(x)$ would lead to the ill-formed object $(\lambda x.f\,(x))\,(x)$. What is the expected value of this

[5] Notice the use of the higher-order type in rule AllI: its argument is a *term* variable x, ensuring that it occurs only in this subproof. This side-condition, usually explicit in rule $\forall_{\mathrm{I}}$ is here handled by the metalanguage: it is called parametric reasoning

substitution? The encoding of proof:

$$\dfrac{\begin{array}{c}\mathscr{D}\\ \vdash \forall x.\, \mathsf{p}(x)\end{array}}{\vdash \mathsf{p}(\mathsf{f}(\mathsf{a},\mathsf{a}))}$$

is the following classification:

$$\mathsf{AllE}\,(\lambda x.\,\mathsf{p}\,(x),\mathsf{f}\,(\mathsf{a},\mathsf{a}),M)\ :\ \mathsf{is}\,(\mathsf{p}\,(\mathsf{f}\,(\mathsf{a},\mathsf{a})))\,,$$

where M codes for $\mathscr{D}$. The type of this object must be the this one (it is the encoding of the judgment above), so substitution $(\mathsf{is}\,(f\,(x)))[f/\lambda x.\,\mathsf{p}\,(x), x/\mathsf{f}\,(\mathsf{a},\mathsf{a})]$ must be $\mathsf{is}\,(\mathsf{p}\,(\mathsf{f}\,(\mathsf{a},\mathsf{a})))$. Such a notion of substitution performing on-the-fly reduction of β-redexes is called *hereditary* [Watkins et al., 2003]: its result is always a canonical object, all intermediate redexes being reduced recursively, much like big-step operational semantics [Kahn, 1987].

Does this operation always terminate? This is one of the main result of the metatheory of LF, that we describe in the next section.

1.2 | SLF: SPINE-FORM LF

We now describe in detail the syntax and verification algorithm of *spine-form* LF, and develop its metatheory.

The original presentation of LF [Harper et al., 1993] defined it as a simply typed lambda-calculus, with richer types depending over terms, and an additional *conversion rule*, whose role was to identify all types being β-equivalent.[6] Proving the decidability of type checking followed from strong normalization of well-typed terms and the Church-Rosser property of β-reduction. Watkins et al. [2003, 2004] first proposed a system—*Canonical* LF—in which only *canonical forms* have an existence, i.e., where a well-typed term is necessarily β-normal and η-*long*. Substitution is then hereditary, β-normal form is enforced syntactically and η-long form by typing. It allows to directly reason by induction over these canonical forms, easing metatheory, adequacy proofs and implementation. Its main metatheorical result consists in the decidability of type checking, which reduces to the termination of hereditary substitutions.

The system we present here is a variant of canonical LF, called *spine-form* LF (or SLF for short) in Cervesato and Pfenning [2003], Pfenning and Simmons [2007]. In this variant, application is not a binary construct $R\ M$ but n-ary: a "function symbol"

[6] It also featured a type-level λ-abstraction, which was rapidly dropped due to the complication this induces to the metatheory of the language (see Pfenning [2001] for a survey and history of logical frameworks).

applied to a list, called a *spine* S, of arguments. The functional part of the application can neither be a λ-abstraction (it would form a redex), nor another application (both spines could be concatenated into a single, longer spine), so it must be a *head* H, i.e., a variable or a constant. Proof-theoretically, this calculus can be seen through the Curry-Howard isomorphism as a notation for proofs of LJT [Herbelin, 1994] with a universal quantifier, which is a focused sequent calculus (see Chapter 4). The advantages of exposing the head H of an n-ary application, which was buried under n binary applications in canonical LF are of practical nature (unification and proof search are more efficient, as was suggested by an empirical study in Michaylov and Pfenning [1992]) as well as theoretical: applications (atomic objects) are classified by (atomic) types, and η-long normal form-preserving hereditary substitution is easier to define and reason about.

The metatheory of SLF has been carried out by Sarnat [2010, section 5.1], that of LF by Harper and Licata [2007]. In Section 1.2.2, we give a slight variation of it that is novel by several aspects.

- First, existing developments involve only single substitution; we extend it to parallel substitutions, since it will come useful in Chapter 2.
- Secondly, existing developments rely on a notion hereditary substitution that is indexed by a decreasing simple type argument to make it terminate. We sketch that by modifying slightly its definition, we can get rid of this type, and clearly separate substitution from typing.
- Finally, we give a novel proof of η-expansion that is more compact than those of Sarnat [2010], Cervesato and Pfenning [2003] and showcases the relationship between LF and SLF.

1.2.1 | DEFINITION

A | SYNTAX

Figure 1.2 presents the syntax of SLF. As sketched in previous section, a *kind* K is either a dependent product or $*$, the classifier of types. A *type family* A is either a dependent product (written $A \to B$ in the degenerate case where it is not dependent) or an *atomic type*, i.e., the classifier of objects. For future reference, we isolate the syntactic category of *atomic type* P, a *type constant* a applied to a *spine* S, i.e., a list of arguments which are canonical objects M. A *canonical object* M is classified by a type family and is either an abstraction or an atomic object. An *atomic object* F is classified by an atomic type and is the application of a *head* H (*object constant* c or variable x) to a spine S. When unambiguous, we will write $H\ M_1\ M_2$ for $H\,(M_1, M_2)$ to help relate it to canonical LF, which has binary, curryfied applications; however bear in mind that on the contrary to them, spines "grow on the right" (they are *cons-list*): the first argument is the topmost in the AST. Similarly, an atomic object $H\,(\cdot)$ that has

$$
\begin{array}{rcll}
K & ::= & \Pi x : A.K \mid * & \text{Kind} \\
A,B & ::= & \Pi x : A.A \mid P & \text{Type family} \\
P & ::= & \mathsf{a}(S) & \text{Atomic type} \\
\\
M,N & ::= & \lambda x.M \mid F & \text{Canonical object} \\
F & ::= & H(S) & \text{Atomic object} \\
H & ::= & x \mid \mathsf{c} & \text{Head} \\
S & ::= & \cdot \mid M,S & \text{Spine} \\
\\
\sigma & ::= & \cdot \mid \sigma, x/M & \text{Parallel Substitution} \\
\Gamma & ::= & \cdot \mid \Gamma, x : A & \text{Environment} \\
\Sigma & ::= & \cdot \mid \Sigma, \mathsf{c} : A \mid \Sigma, \mathsf{a} : K & \text{Signature}
\end{array}
$$

Figure 1.2: Syntax of SLF: spine-form LF

an empty spine will be written simply H. For all syntactic class X among K, A, P, M, F, H, S, we define the set $\mathsf{FV}(X)$ of all free variables in X in the customary manner.

A *signature* Σ defines an object language: it is a list of declarations of object or type constants. A local environment Γ is used while type checking a term; it is a list of variable declarations with their types. Both signatures and environments are represented, as is as customary, as *snoc-list* (they "grow on the left") because they bind "reversely" to $\lambda x.M$: in binding $\Gamma, x : A$, type family A has in scope all variables declared in Γ. To emphasize their relation with environments, *parallel substitutions* are also presented as snoc-list, but unlike them, they do not bind: in $\sigma, x/M$, variables in $\mathrm{dom}(\sigma)$ are free in M. As customary, for each X among σ, Σ and Γ, we define $\mathrm{dom}(X)$ to be the set of all the names bound by X, and assume that all these names are different; the set $\mathsf{FV}(X)$ is extended to X in the obvious way, respecting its binding structure.

B | HEREDITARY SUBSTITUTION

Next we present the algorithm of hereditary, parallel substitution $K[\sigma]$ used in typing and defined on Figure 1.3. It is also defined on A, P, M, F and S. A *parallel substitution*[7] substitutes all its *substituends* in the term at once, i.e., each without affecting the free variables of other substituends; for instance, $x[x/y, y/z] = y$. It

[7] Although we could have defined single substitution here, parallel one is a generalization that will be needed in Section 2.2

$$
\begin{aligned}
(\lambda x.M)[\sigma] &= \lambda x.M[\sigma] && \text{if } x \notin \mathsf{FV}(\sigma) \text{ and } x \notin \mathrm{dom}(\sigma) && (1.14)\\
(\mathsf{c}(S))[\sigma] &= \mathsf{c}(S[\sigma]) && && (1.15)\\
(x(S))[\sigma] &= x(S[\sigma]) && \text{if } x \notin \mathsf{FV}(\sigma) \text{ and } x \notin \mathrm{dom}(\sigma) && (1.16)\\
(x(S))[\sigma] &= M \star S[\sigma] && \text{if } x/M \in \sigma && (1.17)\\
\cdot[\sigma] &= \cdot && && (1.18)\\
(M,S)[\sigma] &= M[\sigma], S[\sigma] && && (1.19)\\
*[\sigma] &= * && && (1.20)\\
(\Pi x : A.B)[\sigma] &= \Pi x : A[\sigma].B[\sigma] && \text{if } x \notin \mathsf{FV}(\sigma) \text{ and } x \notin \mathrm{dom}(\sigma) && (1.21)\\
(\mathsf{a}(S))[\sigma] &= \mathsf{a}(S[\sigma]) && && (1.22)\\
\lambda x.M \star N, S &= M[x/N] \star S && && (1.23)\\
F \star \cdot &= F && && (1.24)
\end{aligned}
$$

Figure 1.3: Hereditary substitution of SLF

takes a kind K and a substitution σ to either a kind K or an error: it is a partial function. It distributes homomorphically on subterms of kinds, types, objects and spines (avoiding capture). The interesting case happens when we encounter a variable application $x(S)$ when $x/M \in \sigma$ (Equation (1.17)): then we trigger a chain of $|S|$ *cuts* computed by the auxiliary function $M \star S$ (Equations (1.23) and (1.24)). It recursively "consumes" the $|S|$ argument if and only if they are in front of $|S|$ λ-abstractions in M, stopping when the S is empty or in front of an atomic term F.

Example 1.3. If σ is the substitution $f/\lambda x.\mathsf{p}(x), x/\mathsf{f}(\mathsf{a},\mathsf{a})$ then:

$$
\begin{aligned}
(\mathsf{is}(f(x)))[\sigma] &= \mathsf{is}((f(x))[\sigma])\\
&= \mathsf{is}((\lambda x.\mathsf{p}(x)) \star x[\sigma])\\
&= \mathsf{is}((\lambda x.\mathsf{p}(x)) \star (\mathsf{f}(\mathsf{a},\mathsf{a}) \star \cdot))\\
&= \mathsf{is}((\lambda x.\mathsf{p}(x)) \star \mathsf{f}(\mathsf{a},\mathsf{a}))\\
&= \mathsf{is}((\mathsf{p}(\mathsf{f}(\mathsf{a},\mathsf{a}))) \star \cdot)\\
&= \mathsf{is}(\mathsf{p}(\mathsf{f}(\mathsf{a},\mathsf{a})))
\end{aligned}
$$

This notion of substitution, analogous to the reduction of Cervesato and Pfenning [1997]'s spine calculus, is identical to that implemented in Twelf [Pfenning and Simmons, 2007]. It is not equivalent, however, to the more standard definition of Watkins et al. [2003], Harper and Licata [2007] as it *presupposes* that all objects are in η-*long* normal form [Huet, 1976]: once a cut starts, there must be exactly as many formal and actual arguments (Equation (1.23)) for it to succeed (Equation (1.24),

called Nil-reduction in Cervesato and Pfenning [1997]) . A non-empty spine cut against another non-empty spine $H\,(S)\star S'$ blocks the substitution (in practice, returns an error), instead of reducing to $H\,(S@S')$ (an implicit η-expansion); conversely, exhausting actual but not formal arguments as in $\lambda x.\,M\star\cdot$ blocks the substitution too, instead of returning $\lambda x.\,M$.

The usual hereditary substitution for the untyped, canonical λ-calculus is easily shown to diverge; Watkins et al. [2003] index this operation by some type information to make it terminating, and prove that it is *complete*, i.e., that it computes appropriately substitution on well-typed term. In the next section, we conjecture that restricting it as we did to η-long forms is sufficient for it to terminate, as hinted in this example:

Example 1.4. The term $(\lambda x.x\,(x))\,(\lambda y.y)$ is syntactically not an canonical object. Instead we write it as the substitution operation $(x\,(x))[x/\lambda y.y]$, which is ill-defined and blocks after a few steps: the second x in the left part should be substituted by a function, yet it is not itself a function (it is not applied):

$$\begin{aligned}(x\,(x))[x/\lambda y.y] &= \lambda y.y\star(x[x/\lambda y.y])\\ &= \lambda y.y\star(\lambda y.y\star\cdot)\\ &\neq \quad \text{(error)}\end{aligned}$$

A fortiori, the substitution standing for object $(\lambda x.x\,(x))\,(\lambda x.x\,(x))$, i.e., the prototypical example of non terminating λ-term, terminates with an error. To simulate its non-terminating behaviour, we would have to build the infinitely η-expanded substitution:

$$x\,(M)[x/M] \quad \text{where} \quad M=\lambda y.\,M\star y$$

C | TYPING ALGORITHM

Finally we can describe the typing algorithm of SLF. It is itself presented as a syntax-directed inference system on Figures 1.4 and 1.5. This algorithm is *bidirectional* in the sense of Pierce and Turner [2000]: since the syntax is stratified to contain only canonical objects, we distinguish two kinds of judgment modes, those *synthesizing* the type of an object, and those *checking* that it has a given type, and consequently allows to omit type annotations on λ-abstractions. The type of canonical object is *checked* (it must be provided), whereas the type of atomic objects is *inferred* (synthesized by the algorithm). The judgments involved are thus by convention of the form $\_\vdash\_\Leftarrow\_$ if placeholders have a positive mode (they are input) or $\_\vdash\_\Rightarrow\_$ when the first two are positive and the last is negative (it is an output). All judgments except $\vdash\Sigma$ sig have an implicit parameter Σ, the current signature, and should be read $\_\vdash_\Sigma\_\Rightarrow\_$; all judgments assume that this signature is well-typed: $\vdash\Sigma$ sig.

$\boxed{\Gamma \vdash M \Leftarrow A}$ Canonical object

$$\textsc{MLam}\ \frac{\Gamma, x : A \vdash M \Leftarrow B}{\Gamma \vdash \lambda x.\, M \Leftarrow \Pi x : A.\, B} \qquad \textsc{MAtom}\ \frac{\Gamma \vdash F \Rightarrow A}{\Gamma \vdash F \Leftarrow A}$$

$\boxed{\Gamma \vdash F \Rightarrow P}$ Atomic object

$$\textsc{FVar}\ \frac{x : A \in \Gamma \qquad \Gamma; A \vdash S \Rightarrow P}{\Gamma \vdash x(S) \Rightarrow P} \qquad \textsc{FConst}\ \frac{\mathsf{c} : A \in \Sigma \qquad \Gamma; A \vdash S \Rightarrow P}{\Gamma \vdash \mathsf{c}(S) \Rightarrow P}$$

$\boxed{\Gamma; A \vdash S \Rightarrow P}$ Object spine

$$\textsc{SCons}\ \frac{\Gamma \vdash M \Leftarrow A \qquad \Gamma; B[x/M] \vdash S \Rightarrow P}{\Gamma; \Pi x : A.\, B \vdash M, S \Rightarrow P} \qquad \textsc{SNil}\ \frac{}{\Gamma; P \vdash \cdot \Rightarrow P}$$

$\boxed{\Gamma \vdash \sigma \Leftarrow \Gamma'}$ Substitution

$$\sigma\textsc{Cons}\ \frac{\Gamma \vdash M \Leftarrow A[\sigma] \qquad \Gamma \vdash \sigma \Leftarrow \Gamma'}{\Gamma \vdash \sigma, x/M \Leftarrow \Gamma', x : A} \qquad \sigma\textsc{Nil}\ \frac{}{\Gamma \vdash \cdot \Leftarrow \cdot}$$

Figure 1.4: Bidirectional typing algorithm for SLF (1/2)

To lighten the syntax of the rules, and since it is invariant throughout typing, this signature is not mentioned.

The main judgment is $\Gamma \vdash F \Rightarrow P$ which infers the classification of an atomic object (or fails if it is ill-typed). Looking at the head symbol H, we look up its type, either in the local environment Γ or in the signature Σ, and compares it against the spine of argument, one argument at a time, until the end of the spine (judgment $\Gamma; A \vdash S \Rightarrow P$). Notice the form of this judgment: it has a distinguished type A which is the type awaited by the head symbol, and an inferred atomic type P, the return type computed for the whole application. The type of every application must be an atomic type, therefore it must be *total*, or in other words, objects are enforced to be in η-*long* normal form [Huet, 1976]. For each argument, the remaining expected type is substituted with the actual argument as discussed before (rule SCons); if the argument is not dependent then x does not appear in B and this substitution has no effect. At the end of the spine (rule SNil), the remaining atomic type is returned

as the type of the application. Canonical objects are typed as follows: the type of a λ-abstraction is a dependent product (MLam), an atomic object is canonical provided its expected and inferred types match (MAtom); the notion of *definitional equality* here being α-equivalence.

Although not part of an implementation of an SLF type checker, we provide a typing judgment for substitutions. A substitution σ is classified by an environment Γ' (written $\Gamma \vdash \sigma \Leftarrow \Gamma'$): for each variable $x \in \mathrm{dom}(\sigma)$, its *substituend* M must be (almost) of the type A associated to x in Γ (rule σCons). Since each binding in a parallel substitution is independent (it does not bind in the rest of the substitution), but bindings in environments do, we need to substitute type A by the rest of the substitution, hence the type $A[\sigma]$ in the first premise of σCons. There is an interesting way of reading this judgment, that will be proved later on: if $\Gamma \vdash \sigma \Leftarrow \Gamma'$, then σ is the substitution taking an object M well-typed in environment Γ' to an object $M[\sigma]$ well-typed in environment Γ (note the reversal).

Signatures must be verified by judgment $\vdash \Sigma$ sig before they are used. An empty signature is well-formed (SigNil), and a binding $\Sigma, \mathsf{c} : A$ (*resp.* $\Sigma, \mathsf{a} : K$) is well-formed if Σ is well-formed and A is a well-formed type family (*resp.* K kind) in the empty environment and signature Σ (SigType and SigKind). Note that each constant name in a signature must be unique.[8]

Verifying kinds or types necessitates an environment, since they both contain bindings. An atomic type $\mathsf{a}(S)$ is well-formed if the argument objects match the types mentioned in the kind of a (rule AAtom); this is done with the judgment $\Gamma; K \vdash S \Rightarrow *$ which is the type-level analogous to $\Gamma; A \vdash S \Rightarrow P$.

1.2.2 | METATHEORY

The metatheory of SLF mainly concerns *substitution*. It consists of two main theorems. First, the typing rules of Figures 1.4 and 1.5 form an algorithm; this relies on the fact that substitution is decidable, that is, substitution is *not too strong*. Secondly, SLF enjoys the equivalent of the *substitution principle* [Martin-Löf and Sambin, 1984], which states that a hypothetical proof can always become a categorical proof when we get proofs for its hypotheses; this amounts to say that substitution is *not too weak*.

A | SUBSTITUTION

First, let us look at the termination of the substitution operation. In modern metatheories of LF and its variants [Watkins et al., 2003, Harper and Licata, 2007, Sarnat, 2010], substitution is a defined as a predicate. A fixed point substitution like $\Delta\Delta$ would accept an infinite derivation, except for the fact that this predicate is indexed

[8] Otherwise we would lose the property that if $\Gamma \vdash F \Leftarrow P$ then $\Gamma \vdash P$ type.

$\boxed{\Gamma \vdash K \text{ kind}}$ Kind

$$\textsc{KProd}\ \frac{\Gamma \vdash A \text{ type} \qquad \Gamma, x : A \vdash K \text{ kind}}{\Gamma \vdash \Pi x : A.\, K \text{ kind}} \qquad \textsc{KType}\ \frac{}{\Gamma \vdash * \text{ kind}}$$

$\boxed{\Gamma \vdash A \text{ type}}$ Type

$$\textsc{AProd}\ \frac{\Gamma \vdash A \text{ type} \qquad \Gamma, x : A \vdash B \text{ type}}{\Gamma \vdash \Pi x : A.\, B \text{ type}} \qquad \textsc{AAtom}\ \frac{\mathsf{a} : K \in \Sigma \qquad \Gamma; K \vdash S \Rightarrow *}{\Gamma \vdash \mathsf{a}\,(S) \text{ type}}$$

$\boxed{\Gamma; K \vdash S \Rightarrow *}$ Type spine

$$\textsc{ASCons}\ \frac{\Gamma \vdash M \Leftarrow A \qquad \Gamma; K[x/M] \vdash S \Rightarrow *}{\Gamma; \Pi x : A.\, K \vdash M, S \Rightarrow *} \qquad \textsc{ASNil}\ \frac{}{\Gamma; * \vdash \cdot \Rightarrow *}$$

$\boxed{\vdash \Sigma \text{ sig}}$ Signature

$$\textsc{SigNil}\ \frac{}{\vdash \cdot \text{ sig}} \qquad \textsc{SigType}\ \frac{\vdash \Sigma \text{ sig} \qquad \cdot \vdash_\Sigma A \text{ type} \qquad \mathsf{c} \notin \mathrm{dom}(\Sigma)}{\vdash \Sigma, \mathsf{c} : A \text{ sig}}$$

$$\textsc{SigKind}\ \frac{\vdash \Sigma \text{ sig} \qquad \cdot \vdash_\Sigma K \text{ kind} \qquad \mathsf{a} \notin \mathrm{dom}(\Sigma)}{\vdash \Sigma, \mathsf{a} : K \text{ sig}}$$

$\boxed{\vdash \Gamma \text{ env}}$ Environment

$$\textsc{EnvNil}\ \frac{}{\vdash \cdot \text{ env}} \qquad \textsc{EnvCons}\ \frac{\vdash \Gamma \text{ env} \qquad \Gamma \vdash A \text{ type}}{\vdash \Gamma, x : A \text{ env}}$$

Figure 1.5: Bidirectional typing algorithm for SLF (2/2)

by the type of the *substituend*, that is strictly decreasing (which makes substitution a partial operation). Since substitution is always fed with well-typed objects, this type can be passed to from typing to substitution and ensure termination and totality.

We conjecture that our definition of hereditary substitution on Figure 1.3, that is the slight variant where "implicit" η-expansion is not allowed, is indeed terminating, even if it does not have to rely on typing information.

Conjecture 1.1 (Decidability of substitution)**.**

1. *For all M, σ, either there is a unique M' such that $M[\sigma] = M'$ or there is no such M';*
2. *For all F, σ, either there is a unique F' such that $F[\sigma] = F'$ or there is no such F';*
3. *For all S, σ, either there is a unique S' such that $S[\sigma] = S'$ or there is no such S';*
4. *For all M, S, either there is a unique F such that $M \star S = F$ or there is no such F.*
5. *For all A, σ, either there is a unique A' such that $A[\sigma] = A'$ or there is no such A';*
6. *For all K, σ, either there is a unique K' such that $K[\sigma] = K'$ or there is no such K';*

We follow by a few easy lemmas on inoperative substitutions. There are three ways for a substitution to be the inoperative, that is to be equivalent to the identity: to be empty, to be vacuous, and to be the identity substitution. The last one will be dealt with further below.

Lemma 1.1 (Empty substitution)**.** *For X in $\{M, S, F, A, K\}$, $X[\cdot] = X$*

Proof. Easy induction over X. □

Lemma 1.2 (Vacuous substitution)**.** *For X in $\{M, S, F, A, K\}$, if $\mathsf{FV}(X) \cap \mathrm{dom}(\sigma) = \emptyset$, then $X[\sigma] = X$*

Proof. Easy induction over X. □

The following lemma is non-trivial, but will be useful to prove the substitution principle. Composing two substitutions amounts to applying the outer one to each binding of the inner one. We need to know that two successive substitutions can be applied in any order.

Definition 1.1 (Substitution composition)**.** We define the composition $\sigma_0[\sigma]$ of two substitutions by recursion on σ_0:

$$\cdot[\sigma] = \cdot \tag{1.25}$$

$$(\sigma', x/M)[\sigma] = (\sigma'[\sigma]), x/(M[\sigma]) \qquad \text{if } x \notin \mathsf{FV}(\sigma) \text{ and } x \notin \mathrm{dom}(\sigma) \tag{1.26}$$

Lemma 1.3 (Substitution composition)**.**

1. *For any X in $\{M, S, F, A\}$, $X[\sigma_0][\sigma] = X[\sigma][\sigma_0[\sigma]]$*

2. If $M \star S = F$ *and* $M[\sigma] = M'$ *and* $S[\sigma] = S'$*, then* $M' \star S' = F'$ *where* $F' = F[\sigma]$

Proof. We first prove the clauses *1.* with $X \in \{M, S, F\}$ and *2.*, and then the clause *1.* where $X = A$. Clause *1.* is proved by mutual functional induction over the computation of $X[\sigma_0]$, similarly *2.* is proved over $M[\sigma]$. □

B | TYPING

The first important result now becomes easy: typing forms an algorithm. As before, we prove the following theorem in an intuitionistic informal logic, which provides a decision algorithm.

Theorem 1.3 (Decidability of typing)**.**

1. For all Σ*, either* $\vdash \Sigma$ sig *or not.*
2. Assume Σ *such that* $\vdash \Sigma$ sig*. Then:*

- *(a) For all* Γ *and* K*, either* $\Gamma \vdash K$ kind *or not.*
- *(b) For all* Γ *and* A*, either* $\Gamma \vdash A$ type *or not.*
- *(c) For all* Γ*,* K *and* S*, either* $\Gamma; K \vdash S \Rightarrow *$ *or not.*
- *(d) For all* Γ*,* M *and* A*, either* $\Gamma \vdash M \Leftarrow A$ *or not.*
- *(e) For all* Γ *and* F*, either there exists* P *such that* $\Gamma \vdash F \Rightarrow P$ *or not.*
- *(f) For all* Γ*,* A *and* S*, either there exists* P *such that* $\Gamma; A \vdash S \Rightarrow P$ *or not.*

Proof. First, we prove parts *2. (d)-(f)* by mutual induction over *resp.* M, F and S, then we prove *2. (a)-(c)* by induction over *resp.* K, A, S. We use conjecture 1.1 in the SCons and ASCons cases. □

Weakening is a standard property of "localized", or first-order representations of systems supporting hypothetical reasoning, like SLF. It states that if an object is well-typed under a given environment, it is also well-typed in an extended environment: new bindings just have to be ignored.

Lemma 1.4 (Weakening)**.**

1. If $\Gamma_1 @ \Gamma_2 \vdash M \Leftarrow A$ *and* $\vdash \Gamma$ env *then* $\Gamma_1 @ \Gamma @ \Gamma_2 \vdash M \Leftarrow A$
2. If $\Gamma_1 @ \Gamma_2 \vdash F \Rightarrow P$ *and* $\vdash \Gamma$ env *then* $\Gamma_1 @ \Gamma @ \Gamma_2 \vdash F \Leftarrow P$
3. If $\Gamma_1 @ \Gamma_2; A \vdash S \Rightarrow P$ *and* $\vdash \Gamma$ env *then* $\Gamma_1 @ \Gamma @ \Gamma_2; A \vdash S \Rightarrow P$

Proof. By easy mutual induction on Γ, followed by the given derivation. □

Finally, the substitution theorem states that we can always replace variables in the environment by actual terms, and not change the typing of the object. In other words, substitution is a total operation on well-typed terms.

Theorem 1.4 (Substitution)**.**

1. *If* $\Gamma; A \vdash S \Rightarrow P$ *and* $\Gamma \vdash M \Leftarrow A$ *then* $M \star S = F$ *and* $\Gamma \vdash F \Rightarrow P$*;*
2. *Let* σ *such that* $\Gamma_1 \vdash \sigma \Leftarrow \Gamma$*:*
   (a) *If* $\Gamma_1@\Gamma@\Gamma_2 \vdash F \Rightarrow P$ *then* $F[\sigma] = F'$, $P[\sigma] = P'$, $\Gamma_2[\sigma] = \Gamma_2'$ *and* $\Gamma_1@\Gamma_2' \vdash F' \Rightarrow P'$*;*
   (b) *If* $\Gamma_1@\Gamma@\Gamma_2 \vdash M \Leftarrow A$ *then* $M[\sigma] = M'$, $A[\sigma] = A'$, $\Gamma_2[\sigma] = \Gamma_2'$ *and* $\Gamma_1@\Gamma_2' \vdash M' \Leftarrow A'$*;*
   (c) *If* $\Gamma_1@\Gamma@\Gamma_2 \vdash A$ type *then* $A[\sigma] = A'$, $\Gamma_2[\sigma] = \Gamma_2'$ *and* $\Gamma_1@\Gamma_2' \vdash A'$ type*;*
   (d) *If* $\vdash \Gamma_1@\Gamma@\Gamma_2$ env *then* $\Gamma_2[\sigma] = \Gamma_2'$ *and* $\vdash \Gamma_1@\Gamma_2'$ env*;*

Proof. Clause *1.* is proved by mutual induction over A. If $A = P$, we conclude easily by inversion of the two hypotheses. If $A = \Pi x : A_1.A_2$, by inverting the hypotheses we get that $S = M_0, S_0$ with $\Gamma \vdash M_0 \Leftarrow A_1$ and $\Gamma; A_2[x/M_0] \vdash S_0 \Rightarrow P$ (rule SCons) and $M = \lambda x.N$ with $\Gamma, x : A_1 \vdash N \Leftarrow A_2$ (rule MLam). By clause *2. (b)*, $\Gamma \vdash N[x/M_0] \Leftarrow A_2[x/M_0]$, which by induction gives us an F such that $N[x/M_0] \star S = F$ and $\Gamma \vdash F \Rightarrow P$, by which we conclude.

Clauses *2. (a)-(d)* are proved similarly by mutual induction on Γ. If $\Gamma = \cdot$ then by inversion, $\sigma = \cdot$, and all clauses become trivial by lemma 1.1. If $\Gamma = \Gamma_0, x : C$, then we reason by induction on C, after generalizing the properties. Most cases are straightforward, except FVar, proved with the help of lemma 1.4, and SCons and ASCons proved with lemma 1.3. □

We are over with the strict core of the metatheory of SLF. The subsequent sections define operations that we will need to use in Chapters 2 and 3.

C | VARIABLE η-EXPANSION

We now show how to compute a canonical object M of type A when we are given a variable x of this type. This is non-trivial since our canonical objects must be η-expanded: if A is a product, then M will need to be a λ-abstraction. It is also non-trivial because η-expansion generates terms of the form $\lambda x.M\ x$ which can naturally be written with a binary application but are syntactically not well-formed with n-ary ones: η-expansion adds arguments *at the end* of a spine. Cervesato and Pfenning [1997], Sarnat [2010] show η-expansion in a spine calculus by appealing to a technical, ad-hoc construction of *partial spines*, which are "unfinished" spines that are allowed to be non-η-expanded; they then use the concatenation operation on spines, which introduces more properties like associativity and transitivity.

We prefer to introduce the notion of *reversed objects* which is more natural... in the sense of the natural deduction: they are the usual atomic terms of the standard, NJ-style presentation of canonical LF [Harper and Licata, 2007].[9] We get a more

[9] This is a good instance of the utility of a data structure reversal as studied in Chapter 4.

concise proof of η-expansion, and a more efficient implementation extracted from it, not relying on concatenation.

Definition 1.2 (Reversed objects)**.** Reversed objects are lists of canonical objects finished by a variable: they are either variables or the (binary) application of a reversed and a canonical object.

$$R ::= x \mid R\ M$$

They are typed by the following two rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x > A}\ \textsc{RVar} \qquad \frac{\Gamma \vdash M \Leftarrow \Pi x : A.B \qquad \Gamma \vdash R > A}{\Gamma \vdash R\ M > B[x/M]}\ \textsc{RApp}$$

The following definition and lemmas show the correspondence between reversed R and atomic F objects.

Definition 1.3 (Reversal)**.** The *reversal* $\mathrm{rev}_S(R)$ of a reversed object R in a spine S is the atomic object defined inductively by:

$$\frac{}{\mathrm{rev}_S(x) = x(S)}\ \textsc{RevVar} \qquad \frac{\mathrm{rev}_{M,S}(R) = F}{\mathrm{rev}_S(R\ M) = F}\ \textsc{RevApp}$$

Lemma 1.5 (Decidability of reversal)**.** *For all S and R, there exists an F such that* $\mathrm{rev}_S(R) = F$.

Proof. By easy induction on F. □

Lemma 1.6. *If $\Gamma \vdash R > A$ and $\Gamma; A \vdash S \Rightarrow P$ and $\mathrm{rev}_S(R) = F$ then $\Gamma \vdash F \Rightarrow P$.*

Proof. By induction on the derivation of $\mathrm{rev}_S(R) = F$.

In the RevVar case, we know for all Γ that if $\Gamma \vdash x > A$ then $\Gamma; A \vdash S \Rightarrow P$. By inversion of $\Gamma \vdash x > A$, (rule RVar), we have that $x : A \in \Gamma$ and by hypothesis, $\Gamma; A \vdash S \Rightarrow P$; therefore $\Gamma \vdash x(S) \Rightarrow A$ by rule FVar.

In the RevApp case, we have $\Gamma \vdash R\ M > A$. By inversion, we get (rule RApp) $\Gamma \vdash R > \Pi x : A_1.A$ and $\Gamma \vdash M \Leftarrow A$ by inversion of hypothesis $\Gamma \vdash R\ M > A$. We also have $\Gamma; A \vdash S \Rightarrow P$. We must show that $\Gamma \vdash F \Rightarrow P$ which by the induction hypothesis reduces to showing $\Gamma \vdash R > \Pi x : A_1.A$, which is one of our hypothesis, and $\Gamma; \Pi x : A_1.A \vdash M, S \Rightarrow P$. By SCons, this goal reduces to $\Gamma \vdash M \Leftarrow A_1$ and $\Gamma; A \vdash S \Rightarrow P$ which are both assumptions.

□

Corollary 1.1. *If $\Gamma \vdash R > P$ then $\Gamma \vdash \mathrm{rev}_{\cdot}(R) \Rightarrow P$.*

Proof. By instantiation of the previous lemma in the case where $S = \cdot$: the second derivation then has to be SNIL, which forces $A = P$. □

Thus, the $\mathrm{rev}_\cdot(R)$ operation maps a well-typed reversed object to a well-typed atomic object. We can now define the actual η-expansion operation $\eta_R(A)$. It maps a type A to a canonical object of that type, provided we give it an accumulator R of type A.

Definition 1.4 (reversed object η-expansion)**.** The η-expansion $\eta_R(A)$ of a type A with respect to a reversed atomic object R is defined inductively by:

$$\frac{\eta\textsc{Atom}}{\eta_R(P) = \mathrm{rev}_\cdot(R)} \qquad \frac{\eta\textsc{Prod} \quad \eta_x(A) = M_1 \qquad \eta_{R\,M_1}(B) = M_0}{\eta_R(\Pi x : A.\, B) = \lambda x.\, M_0}$$

Lemma 1.7 (Decidability of η-expansion)**.** *For all R and A, there exists M such that $\eta_R(A) = M$.*

Proof. By simple induction on A. □

Lemma 1.8. *If $\Gamma \vdash R > A$ and $\eta_R(A) = M$ then $\Gamma \vdash M \Leftarrow A$.*

Proof. By induction on the derivation of $\eta_R(A)$.

In the ηATOM case, $A = P$ and $M = \mathrm{rev}_\cdot(R)$, and we know that $\Gamma \vdash R > P$; by corollary 1.1, we conclude $\Gamma \vdash \mathrm{rev}_\cdot(R) \Leftarrow P$.

In the ηPROD case, we know $\Gamma \vdash R > \Pi x : A_1.A_2$, $\eta_x(A_1) = M_1$ and $\eta_{R\,M_1}(A_2) = M_0$, and we have by induction that if $\Gamma, x : A_1 \vdash x > A_1$ (which is true by ηATOM) then $\Gamma, x : A_1 \vdash M_1 \Leftarrow A_1$, and that if $\Gamma, x : A_1 \vdash R\ M_1 > A_2$ then $\Gamma, x : A_1 \vdash M_0 \Leftarrow A_2$. We need to show that $\Gamma \vdash \lambda x.\, M_0 \Leftarrow \Pi x : A_1.A_2$. By MLAM, it reduces to $\Gamma, x : A_1 \vdash M_0 \Leftarrow A_2$ and by the second induction hypothesis to $\Gamma, x : A_1 \vdash R\ M_1 > A_2$. By RAPP, we are left to prove $\Gamma, x : A_1 \vdash R > \Pi x : A_1.A_2$, which is our hypothesis, and $\Gamma, x : A_1 \vdash M_1 \Leftarrow A_1$ which is our first induction hypothesis. □

We will actually only need to η-expand variables, i.e., use $\eta_x(A)$. This corollary provides the first of the two following final results: if we know that $x : A$, then we can construct an object of that type.

Corollary 1.2 (Variable η-expansion)**.** *For all A, $\Gamma, x : A \vdash \eta_x(A) \Leftarrow A$.*

Proof. By instantiation of the previous lemma when $R = x$. Its second hypothesis becomes $\Gamma, x : A \vdash x > A$ which we can deduce by rule RVAR. □

The second part of this soundness result is that substituting a variable by its eta-expansion amounts to doing nothing:

Lemma 1.9 (η-expansion substitution)**.** *Let* $\Gamma = \Gamma', x : A$.

1. *If* $\Gamma \vdash F \Rightarrow P$ *then* $F[x/\eta_x(A)] = F$
2. *If* $\Gamma; B \vdash S \Rightarrow P$ *then* $S[x/\eta_x(A)] = M$
3. *If* $\Gamma \vdash M \Leftarrow B$ *then* $M[x/\eta_x(A)] = M$
4. *If* $\Gamma'; A \vdash S \Rightarrow P$ *then* $\eta_x(A) \star S = x(S)$
5. *If* $\Gamma \vdash B$ type *then* $B[x/\eta_x(A)] = B$
6. *If* $\Gamma \vdash K$ kind *then* $K[x/\eta_x(A)] = K$

Proof. We first prove the first four cases by mutual induction over the derivations of *resp.* $\Gamma \vdash F \Rightarrow P$, $\Gamma; B \vdash S \Rightarrow P$, $\Gamma \vdash M \Leftarrow B$ and $\eta_x(A) = M$. Cases 5 and 6 are proved by easy induction over *resp.* B and K. □

D | IDENTITY SUBSTITUTION

Next, we define the *operation* of building a inoperative substitution, the *identity substitution*, and prove that it is in fact inoperative. Because our objects are constrained to be η-expanded, it is not a simple matter of mapping any variable to itself: the variables must be η-expanded with respect to their types, which are given in an environment Γ.

Definition 1.5 (Identity substitution)**.** We define id_Γ by:

$$\mathrm{id}_\cdot = \cdot \tag{1.27}$$

$$\mathrm{id}_{\Gamma,x:A} = \mathrm{id}_\Gamma, x/\eta_x(A) \tag{1.28}$$

As for any substitution, the identity substitution is classified by an environment, and as for any classification, it happens in an environment. The identity substitution is precisely the substitution that make these two environments equal $\Gamma \vdash \mathrm{id}_\Gamma \Leftarrow \Gamma$. It should be read as: id_Γ is the substitution that takes an object M well-typed in Γ into an object $M[\mathrm{id}_\Gamma]$ well-typed in the same Γ.

Lemma 1.10 (Identity typing)**.** *For all* Γ, $\Gamma \vdash \mathrm{id}_\Gamma \Leftarrow \Gamma$

Proof. By functional induction on the derivation of $\mathrm{id}_\Gamma = \Gamma'$. The first case is trivially solved by σNIL. In the second case, by inversion $\Gamma = \Gamma', x : A$ and by induction, $\Gamma \vdash \mathrm{id}_{\Gamma'} \Leftarrow \Gamma'$. Combined with corollary 1.2, we conclude $\Gamma', x : A \vdash \mathrm{id}_{\Gamma'}, x/\eta_x(A) \Leftarrow \Gamma', x : A$. □

Of course, its most important property is that it transports an object M into, well, the same object M:

Lemma 1.11 (Identity substitution)**.**

1. *If* $\Gamma \vdash F \Rightarrow P$ *then* $F[\mathrm{id}_\Gamma] = F$.

2. *If* $\Gamma \vdash M \Leftarrow A$ *then* $M[\mathrm{id}_\Gamma] = A$.
3. *If* $\Gamma; A \vdash S \Rightarrow P$ *then* $S[\mathrm{id}_\Gamma] = S$.
4. *If* $\Gamma \vdash A$ type *then* $A[\mathrm{id}_\Gamma] = A$.

Proof. All clauses are proved by mutual induction over the computation of id_Γ, with the help of lemma 1.9. □

We chose here to *implement* the identity substitution as a function; this is one choice, another one could have been to make it part of the *data structure* of substitution, and have a special rule for it. This is the approach generally taken by calculi of *explicit substitutions* [Abadi et al., 1991, Abel and Pientka, 2010] that we discuss in Chapter 3.

E | STRENGTHENING

Finally, we will define and prove an operation of *strengthening*. First, it requires to prove a property of the same name. Strengthening is the dual of *weakening* (lemma 1.4): it states that if an object is well-typed in an environment, but does not use some of the bindings of its environment, then it is well-typed in a contracted environment without these bindings.

Lemma 1.12 (Strengthening)**.** *Let* M *(resp* F*,* S*) and* Γ *such that* $\mathrm{dom}(\Gamma) \cap \mathsf{FV}(M) = \emptyset$ *(resp* F*,* S*). Then:*

1. *If* $\Gamma_1@\Gamma@\Gamma_2 \vdash M \Leftarrow A$ *then* $\Gamma_1@\Gamma_2 \vdash M \Rightarrow A$.
2. *If* $\Gamma_1@\Gamma@\Gamma_2 \vdash F \Rightarrow P$ *then* $\Gamma_1@\Gamma_2 \vdash F \Rightarrow P$.
3. *If* $\Gamma_1@\Gamma@\Gamma_2; A \vdash S \Rightarrow P$ *then* $\Gamma_1@\Gamma_2; A \vdash S \Rightarrow P$.

Proof. By lemma 1.2, it suffices to show $\Gamma_1@\Gamma_2 \vdash F[\mathrm{id}_\Gamma] \Rightarrow P[\mathrm{id}_\Gamma]$; by theorem 1.4, it reduces to proving $\Gamma_1@\Gamma@\Gamma_2 \vdash F \Rightarrow P$ which is our hypothesis, and $\Gamma_1 \vdash \mathrm{id}_\Gamma \Leftarrow \Gamma$ which is lemma 1.10. □

This is a somewhat surprising result for two reasons. First, it implies that an object always has the same or more free variables than its type; the type cannot contain more free variables than the object, otherwise the conclusion judgments would be ill-formed. Secondly, it means that if an object M does not use one of Γ's variables $x : A$, we are free to remove it, and the environment will still be valid. This is surprising seen that environments are dependent (a binding can be used in the types of all subsequent bindings); yet, this tells us that if M uses a variable x, it also uses all its dependencies in Γ.

Finally, thanks to this lemma, the strengthening *operation* reduces a typing environment to its minimum with respect to an object:

Definition 1.6 (Strengthening)**.** *Strengthening* takes an environment Γ and an atomic object F to an environment Γ' containing only the free variables of F:

$$\frac{}{\text{stren}_F(\cdot) = \cdot}\ \textsc{StrenNil} \qquad \frac{x \in \mathsf{FV}(F) \quad \text{stren}_F(\Gamma) = \Gamma'}{\text{stren}_F(\Gamma, x : A) = \Gamma', x : A}\ \textsc{StrenCons1}$$

$$\frac{x \notin \mathsf{FV}(F) \quad \text{stren}_F(\Gamma) = \Gamma'}{\text{stren}_F(\Gamma, x : A) = \Gamma'}\ \textsc{StrenCons2}$$

Lemma 1.13 (Decidability of strengthening)**.** *For all Γ and F, there exists Γ' such that* $\text{stren}_\Gamma(F) = \Gamma'$.

Proof. By easy induction over Γ. □

Lemma 1.14. *If $\Gamma @ \Gamma_0 \vdash F \Rightarrow P$ and $\text{stren}_F(\Gamma) = \Gamma'$ then $\Gamma' @ \Gamma_0 \vdash F \Rightarrow P$.*

Proof. By induction on the derivation of $\text{stren}_F(\Gamma) = \Gamma'$, generalized on Γ_0. The property is trivially true in the StrenNil case. In the StrenCons1 case, we know $x \in \mathsf{FV}(F)$, $\text{stren}_F(\Gamma_1) = \Gamma_2$ and $\Gamma_1, x : A @ \Gamma'_0 \vdash F \Rightarrow P$, and we must show $\Gamma_2, x : A @ \Gamma'_0 \vdash F \Rightarrow P$. By the induction hypothesis taking $\Gamma_0 = x : A @ \Gamma'_0$, this reduces to proving $\text{stren}_F(\Gamma_1) = \Gamma_2$, which is a hypothesis, and $\Gamma_1 @ (x : A @ \Gamma'_0) \vdash F \Rightarrow P$, which is equivalent to our hypothesis $\Gamma_1, x : A @ \Gamma'_0 \vdash F \Rightarrow P$. In the StrenCons2 case, we know $x \notin \mathsf{FV}(F)$, $\text{stren}_F(\Gamma_1) = \Gamma_2$ and $\Gamma_1, x : A @ \Gamma'_0 \vdash F \Rightarrow P$, and we must prove $\Gamma_2 @ \Gamma'_0 \vdash F \Rightarrow P$. By the induction hypothesis, this reduces to $\Gamma_1 @ \Gamma'_0 \vdash F \Rightarrow P$. By lemma 1.12, it reduces to our hypothesis $\Gamma_1, x : A @ \Gamma'_0 \vdash F \Rightarrow P$. □

Corollary 1.3. *If $\Gamma \vdash F \Rightarrow P$ then* $\text{stren}_F(\Gamma) \vdash F \Rightarrow P$.

Proof. By the previous lemma, taking $\Gamma_0 = \cdot$. □

2 | COMPUTING AND VERIFYING PROOF CERTIFICATES

¶ | ABSTRACT How can we write software that users can formally trust? How can a user be *convinced* that the result of a computation is correct, without even knowing precisely *how* it was computed? Today, this question is too important to be left to human appreciation, and requires to be based on formal proofs. Relying on the proof representation exposed in Chapter 1, we develop in this chapter a language for writing *certifying* programs: along with their result, they provide a *proof certificate*, that can be verified independently by the users. After motivating the need for certifying software, we present DLF, a language for dynamically computing SLF proof objects. It introduces notably a novel way of computing hypothetical proofs, which is notably hard, and on-the-fly testing of programs. We then study in detail a couple of example programs written in it, to demonstrate its usage.

2.1 | MOTIVATIONS

Bugs, security holes, malware have sadly taken a important part in our everyday life, as much as computers have happily made it easier. We rely every day more heavily on highly complex software, for banking, traveling, health... but also on our very own computers, that should protect our privacy and the integrity of our personal data.

How can I *trust* a piece software? How can I be certain that it will not endanger my budget, privacy, anonymity or worse my health? If I wrote the software myself, then I can maybe convince myself that it is bug-free, perhaps using automated *static analysis* tools, but what if someone else wrote it, or thousands different people like the Linux kernel we all use everyday? What if I cannot review its code, because it is too large, because I do not have the skills or simply because the authors refuse to communicate it?

These questions are today of paramount importance for informatics in general, not only when it comes to *high assurance software*—programs on which rely critical

stakes, like embedded systems in avionics or automated transport, automated trading systems etc.—but also in our very browser, operating systems, communication devices. Every day, we execute mobile code on our browsers, written by people we do not know and cannot trust, yet we need to be assured that these programs do not have a bad behaviour on our computer, intentionally or unintentionally.

¶ | PROOF CERTIFICATES Several different answers exist to this legitimate fear. One is to establish a network of trust with a certain set of programmers, information sources, companies or institutions and accept programs only coming from these sources, or from sources trusted by them. This kind of *certification* is used in practice and widespread techniques exists to ensure the provenance of the information, relying mostly on cryptography. If it reduces considerably the risk of malicious attacks, human mistakes from these trusted entities is still possible.

In this chapter, we study a different approach relying on logical grounds: *proof certificates*. When we receive an information from an untrusted source, from a large piece of software to a single number resulting from a complex computation, we could ask for a *mathematical proof* that it respects our criterion for it to be "acceptable". We could then check the validity of this proof, either manually or automatically, to convince ourselves that this information can be trusted, and reject it if the proof was erroneous, accept and use it if it was correct.

The idea of proof certificates emerged from the work on *Proof-Carrying Code* [Necula, 1997] in the field of compilation. Take a compiler that translates a high-level programming language into machine code. Even if some invariants are semantically guaranteed on the high-level code (say, no memory access outside the reserved stack and heap space), what can be said about machine code? How can I, the receiver of such a machine code, be sure that it has the same guarantees, that it stemmed from a valid high-level program, that a buggy compiler did not introduce faulty memory access? The breakthrough of Necula [1997] was to propose to design compilers that issue, together with the machine code, a formal proof of the safety of that code, given a certain *safety policy* previously agreed upon. I could then verify myself, or with a small proof checker that the code respects my safety policy. This is true in the situation of mobile code, but also when dealing with pluggable, heterogeneous components on a unique machine.

But the very same idea can be applied far outside the field of compilation, and is generally called a *certifying* scheme: for instance, tools like proof search engines, satisfiability solvers, type inference procedures, termination checkers, network packet filters, memory managers all have in common that the relationship between their input (a program, a formula...) and their output (another program, a single boolean, a route, a memory span) is a mathematically expressible statement; the proof of this statement constitutes a certificate of the validity of the answer that can be issued by

the tool. This scheme has several advantages:

- it is formally safe, i.e., if someone trusts the logical basis and the formulation of the safety criterion (the so-called *trusted base*), then she *has no choice* but to trust the certificate;
- a certificate can be communicated at a distance, independently verified by a small, trusted *kernel*;
- perhaps more importantly, this certificate does not reveal any information on the way it was built, i.e., there is no need to reveal the source code of the program that produced it.

Yet another logical approach is to certify the procedure itself by proving that for any valid input, it will issue a safe answer, for instance in a proof assistant. This is the *certified* scheme, and is compared lucidly with the certifying one by Leroy [2006].

¶ | COMPUTING CERTIFICATES With that established, how to express our formal proofs? And how to write the piece of software (compilers for instance) that will generate these proofs? In the light of the previous chapter, the first question will be rapidly eluded by a non-answer: following Necula [1997], let us make the uncommitted choice of LF. Remember: LF is only a logical framework, a syntactical toolkit for hypothetical reasoning principle, in which we can *encode* any logic that features this kind of reasoning. Choosing LF to represent proof certificates thus means that the trusted base of the certificate system will have to provide the signature of a particular logic, agreed upon between the producer and the consumer. Now, the second question is the actual subject of the present chapter. We will present a framework for computing and verifying certificates in SLF. It presents two prominent characteristics.

First, it is difficult to write certificate-issuing programs because they are notably hard to debug. Imagine feeding a large, complex certifying program P with an input I; it issues its output O together with a possibly large proof M, supposedly proving a safety predicate $S(I,O)$. What if the proof checker fails to check that M is a proof of $S(I,O)$? This means that somewhere, program P generated an incorrect piece of proof, but there is no way in general to relate the incorrect inference with a piece of code in P. Leroy [2006] suggests that it is possible to use the compositionality of this certification scheme and split up P into several independently checkable passes. Our system turns this suggestion into a requirement by checking all certificates right where they are produced.

Secondly, certificate-issuing programs have to manipulate proofs, that are inherently higher-order objects: as we generate certificates, we will have to generate and transform terms with binders. Now, usual, functional programming language are notably not adapted to the manipulation of syntax with binders like LF objects: the implementation of α-conversion, capture-avoiding substitution, the management

of free variables are hard and error-prone to implement. Our proposition relies on the genericity of LF—since a large family of logics can be encoded adequately in LF, programming facilities to manipulate its objects should provide "the ultimate solution" to binding—and on an original construct, *function inverses*, to compute over objects with free variables.

¶ OUTLINE This chapter is organized as follows. We begin by presenting informally and discussing by means of examples the various aspects of our framework, putting the emphasis on the two novel features evoked above. We will then formally present our own DLF language: its syntax, that extends SLF with only a few new constructs, its typed evaluation algorithm, and its properties with respect to SLF. Next, we present in details two large-scale case studies for the framework: a type inference procedure and an automated prover. We finally relate it to existing work and point possible further work.

2.2 PRESENTATION

SLF is a *representation language* for proofs, in the manner of HTML for structured documents: it only provides an encoding and a verification algorithm for proofs. Although it embeds a notion of computation (hereditary substitution), it is only superficial: no "real programs" can be written as an SLF object. In other words, these objects are the *data structure* we want programs to manipulate. How to generate or more generally manipulate these proofs, in order to use them as certificates?

In this section, we present informally through examples a programming environment where the data manipulated are SLF objects. Implemented, it takes the form of an OCaml library; we describe here only the minimal core of its computational (functional) language and leave the description of the implementation to Section 3.4. Its main purpose is to rapidly prototype and test *certifying* procedures: even though a procedure can have bugs, the certificates returned are always dynamically checked with a trusted *kernel*, equivalent to the type checker for SLF described in Section 1.2. In the next section, we will formally present the corresponding language DLF and its properties.

2.2.1 SPECIFICATION AND EVALUATION

We propose to extend SLF with a few new syntactic constructs allowing to write certifying code manipulating SLF objects: functions annotated by rich SLF types serving as specifications, and their code, written using only a simple pattern matching construct on atomic objects. It is thus a *two-level system*: the representation level (SLF) and the computation level (pattern-matching). It is "only" a certifying scheme,

therefore, the computational level can be thought as an untyped language; our framework will however insert dynamic checks transparently.

A | SLF TYPES AS SPECIFICATIONS

We choose to represent the specification of these procedures by SLF type families, giving a double role to the dependent product: its usual role as a way to indicate the types of constants arguments, and as a way to describe the relationship between the inputs and output of defined functions, i.e., the usual functional programming function space, except with dynamic type checks. For instance, each occurrence of a function f of type $\Pi x : \mathsf{a}.\, \mathsf{p}\,(x)$ will be dynamically checked to be fed with an argument M of type a and to return an object of type $\mathsf{p}\,(M)$. Combined with the possibility for p (or even a itself) to be the encoding of a complex judgment as in SLF, this allows to assign rich specifications to our procedures.

Example 2.1. Let us consider a weak *call-by-name* evaluator for the λ-calculus. Its implementation can be arbitrarily complex (it can exploit e.g., closures, explicit substitutions, abstract machine, compilation ...), but in the end it must obey the two simple and well-known rules of big-step semantics, taking a closed term to a closed value (a λ-abstraction):

$$\frac{}{\lambda x.\, M \downarrow \lambda x.\, M}\ \textsc{EvLam} \qquad\qquad \frac{M \downarrow \lambda x.\, M' \qquad M'[x/N] \downarrow V}{M\ N \downarrow V}\ \textsc{EvApp}$$

We can design a procedure *eval* taking a λ-term M and producing not only a value V, but also a derivation $M \downarrow V$, enforcing the output V to be correct with respect to input M. First we declare λ-terms tm and their weak values vl using HOAS encoding:

```
tm : ∗.
app : tm → tm → tm.
lam : (tm → tm) → tm.
vl : ∗.
vlam : (tm → tm) → vl.
```

Then the judgment $M \downarrow V$ (ev) and its two rules:

```
ev : tm → vl → ∗.
EvLam : ΠM : tm → tm. ev (lam λx.M x) (vlam λx.M x).
EvApp : ΠM N : tm. ΠP : tm → tm. ΠV : vl.
  ev M (vlam λx.P x) → ev (P N) V →
  ev (app M N) V.
```

We need a way to pair together a value V and a proof that $M \downarrow V$:

$\mathsf{evt} : \mathsf{tm} \to *.$
$\mathsf{Pair} : \Pi M : \mathsf{tm}.\ \Pi V : \mathsf{vl}.\ \mathsf{ev}\ M\ V \to \mathsf{evt}\ M.$

Type evt is the encoding of a dependent pair $\Sigma x : \mathsf{vl}.\,\mathsf{ev}\,(m,x)$, for a given term $m : \mathsf{tm}$. Finally, our certifying evaluator will have type:

$\mathsf{eval} : \Pi M : \mathsf{tm}.\ \mathsf{evt}\ M = \ldots$

independently of the actual implementation details. Fed with an object $M : \mathsf{tm}$, and if it returns an object N, we can check the validity of this answer by checking whether $\Gamma \vdash N \Leftarrow \mathsf{evt}\,(M)$; if it is the case, then we know that $N = \mathsf{Pair}\,(M, M_1, M_2)$ where M_1 is our value of type vl.

B | JUST-IN-TIME TYPE CHECKING

What about just letting a complex procedure (say this time a whole compiler) issue a large certificate as a black box, generated in a completely untrusted manner, and simply check this certificate afterwards? If this certificate proves incorrect, debugging the procedure will be a delicate task: there is no way to relate the ill-typed part of the certificate to the piece of code that generated it. It could even be the case that it was produced by a *correct* piece of code which was given an ill-typed object as input.

Example 2.2. A certifying compiler consists of two passes:

$$c : \mathsf{code} \xrightarrow{\mathsf{frontEnd}} \{d : \mathsf{interm} \mid \mathsf{sameSem}'\,(c,d)\} \xrightarrow{\mathsf{backEnd}} \{e : \mathsf{asm} \mid \mathsf{sameSem}\,(c,e)\}$$

From code c, the front-end generates the pair of some intermediate code d, together with a proof that it preserves semantics[1]. From this pair, the back-end generates assembly code e and a proof that it has the same semantics as the original code c. The back-end thus possibly transforms its input proof to generate its output. If the front-end has a bug and generates an ill-typed proof, then checking the back-end's returned proof will likely fail *a posteriori*, without the user knowing where the bug was. In this case we would better also type check the intermediate proof returned by the front-end, so that erroneous proof generation is detected as early as possible and reported where it was generated.

One could note that this is easily realized by introducing dynamic checks, or *assertions*, at the beginning and end of functions *frontEnd* and *backEnd*. Actually, expressing the specifications in SLF allows to give an even more precise error report.

Example 2.3. Imagine defining the compiler erroneously as:

$$\mathit{compiler} : \Pi c : \mathsf{code}.\ \{e : \mathsf{asm} \mid \mathsf{sameSem}\,(c,e)\} = \mathbf{fun}\ \mathtt{x} \to \mathit{frontEnd}\,(\mathit{backEnd}\,(\mathtt{x}))$$

[1] It is common knowledge that this dependent pair can be coded in SLF, similarly to type evt in the previous example.

The error is in the *glue code* of *compiler* (we inverted the composition of both passes). With simple assertions in *frontEnd* and *backEnd*, this error would be reported at the beginning of *backEnd*, when passed an object of type code instead of interm. But since we have declared types for these functions, we can report the error exactly in this glue code. Let M : code; the object *compiler*(M) is well-typed and evaluates in one step to object *frontEnd*(*backEnd*(M)), which is ill-typed. An error is raised, before evaluating *backEnd*.

To conclude, type checking objects *exactly* when they are produced and consumed by the function manipulating them helps to catch errors and debug certified procedures: we can then produce sensible error messages and fail as early as possible[2]. This requirement can be reformulated as:

> Each object passed to *and* returned by a procedure is *dynamically* checked for well-typing against the declared type of that procedure. If it succeeds, its result is thus guaranteed to be correct with respect to its specification.

As we will see in the next section, this requires to interleave type checking and evaluation phases.

C | EVALUATION BY PATTERN-MATCHING

In what language can we express the code of these functions? It turns out that it can be written entirely with three construct: a "computational" λ-abstraction binding formal arguments, SLF objects possibly containing argument variables, and a pattern-matching construct on atomic terms.

Example 2.4. Let us implement the previous evaluator the simplest way possible, i.e., by using the substitution built in SLF:

```
eval : ΠM : tm. evt M = " fun m → match m with
  | « lam "p" » → « Pair (lam "p") (vlam "p") (EvLam "p") »
  | « app "m" "n" » →
    let « Pair "m" (vlam "p") "e" » = « eval "m" » in
    let « Pair "_" "v" "f" » = « eval ("p" "n") » in
    « Pair (app "m" "n") "v" (EvApp "m" "n" "p" "v" "e" "f") ».
```

We begin by binding argument M : tm to computational variable m with the **fun** m → construct, then pattern-match on it. The first pattern is for the lam case: it binds a computational variable p corresponding to the canonical object of type tm → tm (a λ-term with one free variable). On the other side, we create the pair

[2] At least during the prototyping phase, this seems to be a reasonable help. In production, we could imagine these costly checks to be switched off, keeping only the global, final type check.

of the value vlam ("p") and the "dummy" evaluation proof EvLam ("p"). The second case is application of objects bound to computational variables m and n. We create term *eval* (m) and immediately pattern-match on it[3]. This has the effect of triggering the evaluation of term m; for that, we check dynamically that m is of type tm and that its returned object has type evt (m). The **let** construct binds p to the resulting value's content, and e to the corresponding certificate. The same process is repeated for term "p" ("n"): p is bound to an object of type tm → tm, i.e., a term with a free variable, so its application to n : tm actually denotes the substitution of the free variable of p to n. This binds the final value v and the corresponding certificate f. Finally we construct the certificate for the whole term with Pair and EvApp.

Note that there is no variable case: if *eval* is applied only to closed objects at top-level, this case will never arise.

2.2.2 | ENVIRONMENT-FREE COMPUTATIONS

Soon however, we will encounter the need for these functions to manipulate *open* objects, since SLF objects possibly contain binders. For instance, our certifying evaluator could be extended to evaluate under λ-abstractions (full evaluation); function *eval* would then need to deal with open objects, since we used HOAS for their representation.

A common choice is to make explicit the environment Γ by which terms are closed. For instance, in Beluga [Pientka and Dunfield, 2010], such a function would have a type:

$$\mathit{eval} \ : \ \Pi g : \mathsf{env}.\, \Pi M : ([g]\mathsf{tm}).\, [g](\mathsf{evt}\,(M[\mathbf{id}_g]))$$

A type $[\Gamma]A$ is the type of objects closed by the environment Γ, and an object $M[\sigma]$ is the open object M closed by substitution σ. The first product binds an environment g, the second a term M open in g; the function returns an open certificate evt of the evaluation of M in the same environment (this is indicated by the identity substitution on g written $\mathbf{id}_g$.

This leads to syntactically distinguish *computational* and *representational* types in two strati: in Beluga, the dependent product in function types is not the same as the one used to declare constants. Also, the Beluga code for *eval* would pattern-match on M and include a variable case.

We propose a different and novel approach to this problem: instead of manipulating open objects, we "fool" the system into believing that they are all *locally closed*, thanks to a notion of *function inverses*, inspired by and generalized from the idea of

[3]The **let** construct is just syntactic sugar for a one-branch **match** with an *irrefutable pattern*; the same way, we use deep pattern-matching even though the formal presentation of Section 2.2 only includes shallow case analysis.

type checking without explicit environment (or *context-free*[4]) [Geuvers et al., 2010, Boespflug, 2011]. See Section 2.5 for a comparison.

A | ENVIRONMENT-FREE TYPING

The usual presentation of type checking for e.g., the simply typed λ-calculus features an environment that is threaded throughout the term; at variable nodes, we look up their types in it. Another, more recent presentation is to add a "temporary" construct $\mathit{infer}^0(x,A)$ to the syntax of terms that annotates a variable with its type. When crossing a binder, we substitute this construct for its variable:

$$\textsc{EFLam}\ \frac{\vdash M[x/\mathit{infer}^0(x,A)] : B}{\vdash \lambda x : A.\, M : A \to B} \qquad \textsc{EFApp}\ \frac{\vdash M : A \to B \quad \vdash N : A}{\vdash M\ N : B} \qquad \textsc{EFAnnot}\ \frac{}{\vdash \mathit{infer}^0(x,A) : A}$$

Note that there is no variable case, since they should all have been substituted at their binding site. This forms a type checking algorithm $\mathit{infer} : \mathsf{tm} \to \mathsf{tp}$ that is equivalent to the usual presentation on closed terms. This technique, reminiscent of the *hypothetical notation* of proofs (Section 1.1), is useful to mimic some predicate logic features in type theory [Geuvers et al., 2010] and to implement an abstract representation of typed terms in proof assistants [Boespflug, 2011]. It offers the advantage of completely removing the environment of the picture: during the process, all terms are closed, since we replaced every free variable by $\mathit{infer}^0(x,A)$, which can be considered closed (variable x is only kept for informative purpose). But it is at the cost of having to add this new, "volatile'" construct of open and typed variables to the syntax: what to think of the syntactically correct term $\lambda x : A.\, \mathit{infer}^0(x,A)$? Is it equal to $\lambda x : A.\, x$?

B | COMPUTING ON OPEN TERMS BY SUBSTITUTION

Remark that the new construct infer^0 pairs up variables x (i.e., terms, the input of function infer) to types A (i.e., its output). For reasons that will eventually become clear, we call this pair the *inverse* of function infer. Rule EFAnnot expresses the fact that inferring the type of an annotated term amounts to return the annotation: in functional notation, $\mathit{infer}(\mathit{infer}^0(x,A)) = A$. This idea can be generalized to many computations on open terms.

Example 2.5. Consider for instance the function computing the *projection* of a λ-term $\mathit{prj} : \mathsf{tm} \to \mathsf{tm}$ which erases all application arguments. It is defined on open terms by:

$$\mathit{prj}_\Gamma(\lambda x.\, M) = \lambda x.\, \mathit{prj}_{\Gamma,x}(M) \tag{2.1}$$

[4] We prefer to use the term (typing) environment, to avoid confusion with the (evaluation) contexts of Chapter 4.

$$prj_\Gamma(M\ N) = prj_\Gamma(M) \tag{2.2}$$
$$prj_\Gamma(x) = x \tag{2.3}$$

We can define an equivalent one in the environment-free style:

$$prj(\lambda x.M) = \lambda x.prj(M[x/prj^0(x,x)]) \tag{2.4}$$
$$prj(M\ N) = prj(M) \tag{2.5}$$

or in our syntax:

```
lam : (tm → tm) → tm.
app : tm → tm → tm.
prj : tm → tm = " fun x → match x with
  | « lam "m" » → « lam λx. (prj ("m" (prj⁰ (x, x)))) »
  | « app "m" "n" » → « prj "m" »".
```

The annotated term $prj^0(M,N)$ is a term such that $prj(M) = N$: it pairs up the input M and output N of function prj. We use this annotation only on variables: each bound variable will be replaced by $prj^0(x,x)$, meaning: "the call of prj on this variable should directly evaluate to x". This way, no variable case is necessary for prj: it is replaced by a built-in, generic reduction $prj(prj^0(M,N)) = N$ that we call *contraction* of a function with its inverse.

The type of inverse functions f^0 is deduced from the type of f: we had $infer$: tm → tp, then $infer^0$: tm → tp → tm; we had prj : tm → tm, so prj : tm → tm → tm. This scheme is easily generalized to functions f with n arguments (see below): then, we have a family of inverses f^n, each with its own type.

C | COMPUTING CERTIFICATES BY SUBSTITUTION

The mechanism of substitution by inverses is particularly suitable to certificate-issuing functions, where the return type depends on the arguments' values. Let us see why on an example:

Example 2.6. Consider now the *equals* function comparing two λ-terms up to α-equivalence, and returning a certificate eq for it. This certificate will be built out of this signature:

```
eq : tm → tm → ∗.
EqApp : ΠE1 : tm. ΠF1 : tm. ΠE2 : tm. ΠF2 : tm.
  eq E1 F1 → eq E2 F2 → eq (app E1 E2) (app F1 F2).
EqLam : ΠE : tm→tm. ΠF : tm→tm.
  (Πx : tm. eq x x → eq (E x) (F x)) → eq (lam λx.E x) (lam λx.F x).
```

and the comparison function reads:

```
equals : ΠM : tm. ΠN : tm. eq M N = " fun m n → match m, n with
  | « app "e1" "e2" », « app "f1" "f2" » →
    « EqApp "e1" "f1" "e2" "f2" (equals "e1" "f1") (equals "e2" "f2") »
  | « lam "t" », « lam "u" » →
    « EqLam "t" "u" λx. λh. (equals ("t" (equals⁰ x x h)) ("u" (equals¹ x x h))) ».
```

Pattern-matching is only partial: for concision, we use matching failure as a way to signal non-equality. The app case should be self-explanatory. In the lam, we bind `t` and `u` to the two subterms with one free variables. We return a certificate built out of these, with a recursive call to *equals*, where the free variables of `t` and `u` have been replaced by *resp.* $equals^0\,(x,x,h)$ and $equals^1\,(x,x,h)$. Both these inverses have the same type $\Pi T : \mathsf{tm}.\,\Pi U : \mathsf{tm}.\,\mathsf{eq}\,(T,U) \to \mathsf{tm}$, so that these substitutions are well-typed. They mean that when we will need to compare the two terms x and x (the variables bound by these two λ-abstractions), we should return h of type $\mathsf{eq}\,(x,x)$. In other words, we have the *contraction* rule $equals\ (equals^0\ x\ x\ h)\ (equals^1\ x\ x\ h) = h$.

D | PATTERN-MATCHING UNDER AN ENVIRONMENT

Note that in this last example, the recursive call to *equals* happens in the scope of variables x and h which allows to use them to provide a certificate for the variable case by contraction. But in more complex examples, this call may have to happen out of this environment, and we will then need a variation on the pattern-match construct as introduced in the following example.

Example 2.7. The "volatile" construct $eval^0$ has type $\Pi M : \mathsf{tm}.\,\mathsf{evt}\,(M) \to \mathsf{tm}$: it pairs up a term M and the proof of its correct evaluation. Let us sketch the extension of our evaluator to full evaluation: the rules of our certificates need to be changed, something in the line of:[5]

```
EvLam : ΠM V : tm → tm. (Πx : tm. ev x x → ev (M x) (V x)) → ev (lam M) (lam V).
```

This constant EvLam is the encoding of rule:

$$\frac{\begin{array}{c}[x \downarrow x]\\ \vdots \\ M \downarrow V\end{array}}{\lambda x.\,M \downarrow \lambda x.\,V}\ \textsc{EvLam}$$

The lam case in *eval* needs to be modified (among others). First, we have to evaluate the open term M under the hypothesis that its variable evaluates to itself. By pattern-matching on the result of this evaluation, we get back its value V and a certificate for this evaluation. For this evaluation however, the system needs to know the extended

[5] With the previous definition of type ev, this definition is ill-typed (ev should take two objects of type tm) but should give at least the general idea.

environment in which it will take place; otherwise, type checking intermediate results of this computation will fail. We extend the pattern-matching construct to **match** M **under** Γ **with** C (and its **let** counterpart), where Γ is the *extension* of the current environment in which M is well-typed. The construct **match** M **with** C is then syntactic sugar for **match** M **under** $\cdot$ **with** C. Let us first compute the evaluation of the open term, under the environment $x : \mathsf{tm}, h : \mathsf{ev}\,(x,x)$:

```
| « lam "p" » → let « Pair "_" "v" "e" » under «^env x : tm; h : ev x x » =
    « eval ("p" (eval^0 x (Pair x x h))) » in
```

In the scrutinee, $eval^0$ pairs together the term made up of the free variable, x, and the desired output of *eval* when we meet this variable, i.e., the pair of the value x and its certificate h. Note that the terms we get back (value `v` and certificate `e`) are still open with respect to x and h. It is the responsibility of the programmer to ensure that they are eventually closed in the return, otherwise the next dynamic type check will fail:

```
| « Pair (lam λx."p" x) (lam λx."v") (EvLam (λx."p" x) (λx."v") (λx. λh."e")) »
```

The return is the pair of the value lam λx."v" (`v` is closed with respect to x) and the certificate formed by `e` closed with respect to x and h, and wrapped into the constant EvLam.

Note that we exploited a meta-argument of the system here, namely the fact that value `v` cannot mention name h. Indeed, `v` is put in a context where h is free. Since `v` has type vl, it cannot depend on a variable h of type $\mathsf{ev}\,(x,x)$: values do not depend on certificates. On the contrary, certificate `e` does potentially depend on h, so we have to put it back in a context where both x and h are free.

In Section 2.4, we study some programs written in this style, especially a certifying type checking procedure. Now would be a good moment for the reader to peek at this example if the use of inverses is still not clear.

E | TYPE-LEVEL INVERSE ERASURE

We must be careful about one thing with this approach though: because of dependent types, values can escape in types, in particular values containing inverses, making some intermediate calls ill-typed. To illustrate it, let us follow by hand the evaluation of object *equals* (lam $\lambda x.x$) (lam $\lambda x.x$). In one evaluation step, it becomes:

```
| EqLam (λx.x) (λx.x) λx. λh. equals (equals^0 x x h) (equals^1 x x h)
```

and in another, to the awaited certificate EqLam $(\lambda x.x)$ $(\lambda x.x)$ $\lambda x.\lambda h.h$. The intermediate value however is ill-typed: the subterm at the very end, *equals* (*equals*^{0} x x h) (*equals*^{1} x x h) has type eq (*equals*^{0} x x h) (*equals*^{1} x x h) whereas it is expected by the type of EqLam to have type eq x x. This is the reason why the inverse construction must remain "volatile": it is so to say attached to a surrounding call to *equal*, waiting to

be simplified, but once it escapes from its object into a type, it must collapse to its domain: let us call this reduction *erasure*. In this case, both inverses are collapsed to just x; in general, $equals^i\ (S)$ collapses to the i-th element of spine S, and we prove further down that this reduction preserves well-typing.

2.2.3 | FULL EVALUATION STRATEGY

The algorithm we describe in the next section performs evaluation of special SLF objects that may contain function symbols, and interleaved with evaluation is the type checking of each intermediate results: it is thus a *typed evaluation* procedure.[6] Since values (SLF objects) can contain λ-abstraction, the evaluation in question must be *full*: for instance, in the previous section, we were expecting the intermediate object

$$\mathsf{EqLam}\ (\lambda x.x)\ (\lambda x.x)\ \lambda x.\lambda h.\mathit{equals}\ (\mathit{equals}^0\ x\ x\ h)\ (\mathit{equals}^1\ x\ x\ h)$$

to actually evaluate further, even if the call to *equals* was buried under two λ-abstractions. This kind of evaluation differs from the *weak evaluation* performed in most functional languages: there, no reduction is done under an abstraction until it is substituted by an actual argument.

Moreover, remember that when put in contact with its inverse, a function must not be evaluated, but the domain of the inverse directly returned: in the unary case, $f\ (f^0\ (M, N)) = M$. To detect this situation, given an object $f\ (M)$ to evaluate, we must evaluate M and test if it is the inverse f^0 before actually evaluating f. An adequate strategy for this is *call-by-value*. Once again, we must be careful with this strategy:

Example 2.8. Consider the *prj* function of example 2.5 and the (contrived) input term:[7]

$$\mathit{prj}\ (\mathsf{lam}\ \lambda x.\mathit{prj}^0\ x\ (\mathit{prj}\ x))$$

The inner call to *prj* should not be evaluated, since it does not know how to deal with variables. However, if we evaluate the outer call first, it reduces to:

$$\mathsf{lam}\ \lambda x.\mathit{prj}\ (\mathit{prj}^0\ (\mathit{prj}^0\ x\ x)\ (\mathit{prj}\ (\mathit{prj}^0\ x\ x)))$$

Now, the inner call can be simplified to x, and computation can succeed with object $\mathsf{lam}\ \lambda x.\mathit{prj}^0\ x\ x$.

In other words, we must first execute calls to functions *outside* λ-abstractions

[6] In a sense, this notion is reminiscent of *typed conversion* for Pure Type Systems found in e.g., Geuvers et al. [2010]: if we do not trust the reduction of a term to preserve typing we can check it at each step of the reduction.

[7] This example may seem contrived, however we will use extensively terms of the form $f\ (\mathtt{c}\ (\lambda x.\mathtt{C}_1[f^0\ (\mathtt{C}_2[x], \mathtt{C}_3[f\ (x)])]))$, with $\mathtt{C}_1$, $\mathtt{C}_2$ and $\mathtt{C}_3$ large contexts, to express the *memoization* of calls to f under an environment.

before executing calls to functions *under* one. The good news is that a well-known technique for full evaluation already respects this constraint. We will thus use and adapt *full reduction by iterated symbolic weak reduction and readback* as described by Grégoire and Leroy [2002]. The idea is to implement full reduction by iterating a two phases process:[8]

- a variant of weak evaluation, *symbolic weak evaluation*, allowing free, applied variables to appear in values;
- a *readback* procedure that searches in these weak values a potential closure we could fully reduce.

Note that the authors are interested in contracting β-redexes in arbitrary λ-terms; on the contrary, our objects are already β-normal (canonical), but we are looking to evaluate computational function applications.

2.3 | DLF: DYNAMICALLY COMPUTING SLF OBJECTS

2.3.1 | DEFINITION

We can now turn to the formal definition of our system, that we call DLF.

A | SYNTAX

The syntax of DLF is presented in Figure 2.1; it extends the syntax of SLF of Figure 1.2. We first extend signatures of SLF with a new kind of binding $\Sigma, f : A = \text{“}T\text{”}$ declaring a function f with type A and code T. Functions can be referred to in objects, so we extend heads with function symbols f. For each function f, we can also refer to a family of inverses f^n indexed by a natural number n (the argument number on which we “invert” f). Terms T (function code) start with a list of λ-abstraction, however of a different nature than SLF's abstractions: they bind function arguments (i.e., objects) with a new set of *computational variables* `x`, `y`...; these can be referred to in objects. Atomic terms U follow: they are either atomic objects «F», or a special pattern-matching construct on a atomic terms **match** U **under** Γ **with** C, where C is a list of branches, each carrying a pattern and an atomic term. A pattern is linear, and can have only one form: a constant applied to pairwise distinct variables. The environment Γ extends the current environment with a set of variables that can appear free in U and C. We define the set $\mathsf{FCV}(X)$ of free computational variables in X (for X in $\{H, T, U, C, Q, M, F\}$) the straightforward way.

[8] The big-step semantics of full reduction are not to be found in Grégoire and Leroy [2002]; we wrote them down at `http://syntaxexclamation.wordpress.com/2012/07/05/strong-reduction-in-big-steps/` along with several variants in call-by-name, call-by-value, with and without closures.

$$
\begin{array}{rcll}
\Sigma & ::= & \ldots \mid \Sigma, f : A = \text{“}T\text{”} & \text{Signature} \\
H & ::= & \ldots \mid \text{“x”} \mid f \mid f^n & \text{Head} \\
T & ::= & U \mid \textbf{fun}\ \mathtt{x} \to T & \text{Terms} \\
U & ::= & \text{«}F\text{»} \mid \textbf{match}\ U\ \textbf{under}\ \Gamma\ \textbf{with}\ C & \text{Atomic term} \\
C & ::= & \cdot \mid \text{«}Q\text{»} \Rightarrow U \mid C & \text{Branches} \\
Q & ::= & \mathtt{c}(\mathtt{x}, \ldots, \mathtt{x}) & \text{Pattern}
\end{array}
$$

Figure 2.1: Syntax of DLF, relatively to SLF (Figure 1.2)

To enhance readability, we distinguish syntactically the two “phases” of this language, the *computational world* of (atomic) terms, branches and computational variables, and the *representational world* of objects, types, patterns, signatures etc. by surrounding them by respectively *antiquotations* “T” and *quotations* «F».[9]

Note the unusual fact that there is no direct support for function call (application) or recursion in the computational world. Instead, a “recursive” function is one that returns or pattern-matches on an object containing its own name, that can be in turn typed and evaluated by the main loop.

B | SUBSTITUTION

With our new set of computational variable comes a new substitution operation. On Figure 2.2, we define the unary substitution of computational variables in (atomic) terms, branches, (atomic) objects and spines. Most cases are straightforward homomorphisms, save the side conditions which indicate the binding structure of terms. The only non-trivial case is Equation (2.13): when actually meeting the variable to substitute, we trigger a hereditary cut, as defined in Equation (1.17).

Talking about hereditary substitution, its previous definition (Figure 1.3) must be completed since we extended our language of objects with function symbols and computational variables. Figure 2.3 defines this easy extension: we just go over these constructs homomorphically.

C | PROJECTION

We have almost all material needed to present the typed evaluation algorithm. What we are still missing is first a way to work out the type of an inverse function given

[9] As we will see in Section 3.4, this is actually the way this language is implemented as a library on top of OCaml with a syntax extension written in CamlP4.

$$(\textbf{fun}\,\mathrm{y} \to T)[\mathrm{x}/M] = \textbf{fun}\,\mathrm{y} \to T[\mathrm{x}/M] \qquad \text{if } \mathrm{x} \neq \mathrm{y} \text{ and } \mathrm{x} \notin \mathsf{FCV}(M) \tag{2.6}$$

$$(\textbf{match}\ U\ \textbf{under}\ \Gamma\ \textbf{with}\ C)[\mathrm{x}/M] = \textbf{match}\ U[\mathrm{x}/M]\ \textbf{under}\ \Gamma[\mathrm{x}/M]\ \textbf{with}\ C[\mathrm{x}/M] \tag{2.7}$$

$$(\langle\!\langle \mathsf{c}\,(\vec{\mathrm{y}}) \rangle\!\rangle \Rightarrow U \mid C)[\mathrm{x}/M] = \langle\!\langle \mathsf{c}\,(\vec{\mathrm{y}}) \rangle\!\rangle \Rightarrow U[\mathrm{x}/M] \mid C[\mathrm{x}/M] \qquad \text{if } \mathrm{x} \notin \vec{\mathrm{y}} \tag{2.8}$$

$$\langle\!\langle F \rangle\!\rangle[\mathrm{x}/M] = \langle\!\langle F[\mathrm{x}/M] \rangle\!\rangle \tag{2.9}$$

$$(\lambda x.N)[\mathrm{x}/M] = \lambda x.N[\mathrm{x}/M] \tag{2.10}$$

$$x(S)[\mathrm{x}/M] = x(S[\mathrm{x}/M]) \tag{2.11}$$

$$\mathsf{c}(S)[\mathrm{x}/M] = \mathsf{c}(S[\mathrm{x}/M]) \tag{2.12}$$

$$\text{“}\mathrm{x}\text{”}(S)[\mathrm{x}/M] = M \star (S[\mathrm{x}/M]) \tag{2.13}$$

$$\cdot[\mathrm{x}/M] = \cdot \tag{2.14}$$

$$N, S[\mathrm{x}/M] = N[\mathrm{x}/M], S[\mathrm{x}/M] \tag{2.15}$$

Figure 2.2: Substitution of computation variables in DLF

$$\ldots \tag{2.16}$$

$$f(S)[\sigma] = f(S[\sigma]) \tag{2.17}$$

$$f^i(S)[\sigma] = f^i(S[\sigma]) \tag{2.18}$$

$$\text{“}\mathrm{x}\text{”}(S)[\sigma] = \text{“}\mathrm{x}\text{”}(S[\sigma]) \tag{2.19}$$

Figure 2.3: Hereditary substitution of DLF, relatively to SLF (Figure 1.3)

the type of its associated function: it is computed by a *projection*. To each function $f : A = T$ in a signature Σ, we associate a family of inverses f^n for $n \in \mathbb{N}$, that have type $(A)_n$. Each inverse f^n should be thought as a projection on the n-th argument of f. The "code" of this projection is morally computable from A by function $\pi_n(A)$, but it is not used while evaluating: as exposed earlier, inverses only collapse to their arguments when they are lifted in the type world.

Definition 2.1 (projection type)**.** We define type $(A)_n$ recursively by:

$$(\Pi x : A.\, B)_{n+1} = \Pi x : A.\, (B)_n \tag{2.20}$$
$$(\Pi x : A.\, B)_0 = \Pi x : A.\, (B)_0^A \tag{2.21}$$
$$(\Pi x : B.\, C)_0^A = \Pi x : B.\, (C)_0^A \tag{2.22}$$
$$(P)_0^A = \Pi x : P.\, A \tag{2.23}$$

Note that $(A)_n$ is defined only for $n < |A|$, where $|A|$ is the number of products of A seen as a telescope. It actually consists of two recursive functions: one traversing the telescope of dependent products until it finds the n-th argument, then, remembering this argument's type A, the second going to the bottom of the telescope and adding A as the return type. Similarly:

Definition 2.2 (projection object)**.** We define the canonical object $\pi_n(A)$ recursively by:

$$\pi_{n+1}(\Pi x : A.\, B) = \lambda x.\, \pi_n(B) \tag{2.24}$$
$$\pi_0(\Pi x : A.\, B) = \lambda y.\, \pi_0^{y;A}(B) \tag{2.25}$$
$$\pi_0^{y;C}(\Pi x : A.\, B) = \lambda y.\, \pi_0^{y;C}(B) \tag{2.26}$$
$$\pi_0^{y;C}(P) = \eta_y(C) \tag{2.27}$$

Note the use of η-expansion in Equation (2.27): The n-th argument of a function f is not necessarily of an atomic type, it could be itself functional. In that case, its projection must be η-expanded accordingly.

Example 2.9. Let A be the type of *eval*, i.e., $\Pi M : \mathsf{tm}.\,\mathsf{evt}(M)$. It has only one projection $(A)_0 = \Pi M : \mathsf{tm}.\,\mathsf{evt}(M) \to \mathsf{tm}$ and $\pi_0(A) = \lambda xy.x$. Let B be the type of *equals*, i.e., $\Pi MN : \mathsf{tm}.\,\mathsf{eq}(M,N)$. It has two projections of the same type $(B)_0 = (B)_1 = \Pi MN : \mathsf{tm}.\,\mathsf{eq}(M,N) \to \mathsf{tm}$ and $\pi_0(B) = \lambda xyz.x$ and $\pi_1(B) = \lambda xyz.y$. Let $C = (\mathsf{a} \to \mathsf{b}) \to \mathsf{c}$. $(C)_0 = (\mathsf{a} \to \mathsf{b}) \to \mathsf{c} \to \mathsf{a} \to \mathsf{b}$ and $\pi_0(C) = \lambda xyz.x(z)$.

Finally, we define the operation of erasure of inverses. It takes a DLF object to an SLF object where all inverses have been projected out to their respective arguments.

Definition 2.3 (Erasure of inverses)**.** Let Σ be a signature. The *erasure* $|M|_\Sigma$ of a DLF object M is defined recursively by:

$$|\lambda x.\, M|_\Sigma = \lambda x.\, |M|_\Sigma \tag{2.28}$$
$$|x\,(S)|_\Sigma = x\,(|S|_\Sigma) \tag{2.29}$$
$$|\mathsf{c}\,(S)|_\Sigma = \mathsf{c}\,(|S|_\Sigma) \tag{2.30}$$
$$|M,S|_\Sigma = |M|_\Sigma, |S|_\Sigma \tag{2.31}$$
$$|\cdot|_\Sigma = \cdot \tag{2.32}$$
$$|f^i\,(S)|_\Sigma = |\pi_i(A) \star S|_\Sigma \qquad \text{if } f : A = \text{“}T\text{”} \in \Sigma \tag{2.33}$$

Note that there is no case for functions f. It is on purpose: the erasure is defined and will be only performed on objects containing no function.

If clear from context, we accept to omit the signature Σ from the erasure operation. The only non-trivial case is Equation (2.33): meeting an inverse, we look up its type, and replace it by the corresponding projection. Since we cannot textually replace it, we perform a *cut*.

D | TYPED EVALUATION

The typed evaluation algorithm is based on the tree kind of reductions mentioned above. We will describe it in big-step semantics, but first let us recall the reductions it follows

Definition 2.4 (Small-step semantics)**.** Let f be a function of type A and code T. The following rewriting rules define the small-step semantics of DLF:

$$f\,(f^0\,(S), \ldots, f^n\,(S)) \longrightarrow \pi_n(A) \star S \tag{2.34}$$
$$f\,(S) \longrightarrow T \diamond S \tag{2.35}$$
$$(\textbf{fun}\,\mathtt{x} \to T) \diamond (M,S) \longrightarrow T[\mathtt{x}/M] \diamond S \tag{2.36}$$
$$U \diamond \cdot \longrightarrow U \tag{2.37}$$
$$(\textbf{match}\ \text{«}\mathsf{c}\,(S)\text{»}\ \textbf{with}\ \text{«}\mathsf{c}\,(\vec{\mathtt{x}})\text{»} \Rightarrow U \mid C) \longrightarrow U \tag{2.38}$$
$$(\textbf{match}\ \text{«}\mathsf{c}\,(S)\text{»}\ \textbf{with}\ \text{«}\mathsf{c}'\,(\vec{\mathtt{x}})\text{»} \Rightarrow U \mid C) \longrightarrow \textbf{match}\ \text{«}\mathsf{c}\,(S)\text{»}\ \textbf{with}\ C \tag{2.39}$$
$$f^i\,(S) \longrightarrow \pi_i(A) \star S \tag{2.40}$$

A function applied to all its inverses *contracts* to the last argument of the inverses (the announced result of the function); otherwise, it is evaluated; an inverse that did not meet its corresponding function for contraction is *erased*.

This rewrite system is clearly not confluent: for instance, a function symbol can always be evaluated or contracted. The big-step semantics below fixes a strategy: we give priority to contraction, then to evaluation, then only to erasure.

Figures 2.4 to 2.6 describe the typed evaluation algorithm. It is presented as a set of syntax-directed inference rules. Each judgment has the form $X \vdash Y \heartsuit Z$ for $\heartsuit$ ranging over $\{\downarrow, \uparrow, \Downarrow\}$. These arrows indicate respectively *weak evaluation*, *readback* and *full evaluation*. X and Y are inputs, Z is an output.

As for SLF, every judgment is implicitly parameterized by a constant signature Σ. Evaluation can either succeed with a well-typed value, or fail in two ways: either it can return an ill-typed object or it can fail to find a suitable branch in pattern-matching (matching failure). It can also loop forever.

¶ | FULL EVALUATION The entry point of this algorithm is the judgment $\Gamma \vdash F \Downarrow F' : P$ which states: "In environment Γ, the DLF atomic object F *fully* evaluates to an SLF object F' of type P". Object F' is a *value* in the sense that it does not contain any computational variable, or function or inverse symbols to evaluate anymore.

To fully evaluate an (atomic) object, it suffices to evaluate it weakly, and then read back the weak value (rules SMObj and SFAtom); the *readback* will recursively evaluate under λ-abstractions. A weak value is an object that does not contain function symbols, except under λ-abstractions. It may still contain inverse symbols, under and not under λ-abstractions.

¶ | WEAK EVALUATION OF OBJECTS AND SPINES Let us go over the rules of weak typed evaluation (Figure 2.4). Like SLF, it is a bidirectional system: from a canonical object M and a type A, we can infer its value M'; from an atomic object F, we can infer its value F' and its atomic type P. A λ-abstraction is already a weak value, so we return it as-is (rule MLam). Note that we do not require the body of the abstraction to be well-typed: this will be checked once it is "opened" by substitution resulting from evaluation, or "traversed" by *readback*. There is an implicit type comparison in rule MAtom. It is done implicitly, in the manner of SLF, only up to α-equivalence. This means that no computation (function calls) can appear in types.

Evaluating a variable or constant atom is a matter of evaluating the spine attached to it, returning the reconstituted atomic value (rules FVar and FCons). Along with the evaluation, we thread the expected type of the variable (*resp.* constant) to check it as we go. These two rules are characteristic of *symbolic weak evaluation*: open variables applied to a spine are considered to be values (the usual weak evaluation considers only λ-abstractions to be values). They are also characteristic of *call-by-value*: arguments of open variables must be values too for their application to be a value (in *call-by-name*, $x\,(S)$ is a value, even if S is not). Rule FInv is similar: during weak evaluation, an inverse f^n alone behaves just as a constant; note that in this case, the type of its arguments is checked against the n-th projection type of f as expected. There is no rule concerning the application of computational variables "x": they all

must be substituted during evaluation.[10]

Argument spines are checked/evaluated by rules SNil and SCons. Similarly to the typing of SLF (Figure 1.4) the type A of the head symbol is threaded throughout the list of arguments. Unlike it, all arguments are evaluated recursively and we reconstitute the spine of values in the end. Note the substitution in the codomain of the type in SCons: not only do we substitute the *value* M' in the type (and not the object M), such that we do not introduce computations in the resulting type, but the value *with all inverses erased* $|M'|_\Sigma$, in accordance with the discussion of Section E.

Rule FEvalInv performs the contraction of a function applied to its inverses we discussed above. If all the argument of a function f evaluate to the corresponding inverses $f^0, \dots, f^n$, and if all arguments of these inverses are pairwise syntactically equal, then the function is *not* called, and we simply return the last argument of the inverse. This is performed by the *cut* $\pi_n(A) \star S_0$. If this contraction is not possible, then we have to actually call the function (rule FEval): after evaluating the argument spine S and computing the awaited return type P, we trigger the term evaluation $\Gamma \vdash T \star S' \downarrow F : P$ (T is the code of function f). This returns a value F and a type that must be equal to P. Our initial requirement of Section B is respected here: a function f's actual arguments are checked with respect to its specification A, then it is executed, then the returned value is checked too.

¶ WEAK EVALUATION OF TERMS AND BRANCHES Figure 2.5 concerns the evaluation of terms. We recall that this evaluation is *untyped*; however, because dynamic type checks will be inserted in its course, we thread the current environment Γ. The first judgment for canonical terms, $\Gamma \vdash T \star S \downarrow F : P$ begins by "eating up" formal arguments (computational variables bound in T) and actual arguments S of the function pairwise, replacing one by the other in the body of the function (rule TLam). When it is done (rule TAtom), the remaining atomic term U is evaluated. If it is an object F (rule UAtom), this object is evaluated in turn, and its synthesized type P becomes the type of the function's return. If it is a pattern-matching **match** U **under** Γ' **with** C (rule UCase), we evaluate the scrutinee U only for its value F, which is passed to each branch in C until a match is met (rules CNoMatch and CMatch). Note that the scrutinee U is evaluated in the *enlarged* environment $\Gamma@\Gamma'$ as described informally in example 2.7. Its value F is thus potentially "more open" than the current environment allows. Since this computational language is untyped, it is the programmer's responsibility to ensure that the object eventually returned "closes" the subterms emanating from F by enough λ-abstractions; otherwise a scoping error will be issued in the final dynamic type check of rule UAtom.

[10] This is *not* enforced by this system, which does not check that every computational variable is bound in a function's definition. However, it will be enforced by our OCaml implementation in Section 3.4 since computational variables are implemented by OCaml's variables.

$\boxed{\Gamma \vdash M : A \downarrow M'}$ Canonical object

$$\textsc{MLam}\ \frac{}{\Gamma \vdash \lambda x.M : (\Pi x : A.B) \downarrow \lambda x.M} \qquad \textsc{MAtom}\ \frac{\Gamma \vdash F \downarrow F' : P}{\Gamma \vdash F : P \downarrow F'}$$

$\boxed{\Gamma \vdash F \downarrow F' : P}$ Atomic object

$$\textsc{FVar}\ \frac{x : A \in \Gamma \quad \Gamma; A \vdash S \downarrow S' : P}{\Gamma \vdash x(S) \downarrow x(S') : P} \qquad \textsc{FConst}\ \frac{\mathsf{c} : A \in \Sigma \quad \Gamma; A \vdash S \downarrow S' : P}{\Gamma \vdash \mathsf{c}(S) \downarrow \mathsf{c}(S') : P}$$

$$\textsc{FEval}\ \frac{f : A = \text{“}T\text{”} \in \Sigma \quad \Gamma; A \vdash S \downarrow S' : P \quad S' \neq f^0(S_0), \ldots, f^n(S_n) \quad \Gamma \vdash T \star S' \downarrow F : P}{\Gamma \vdash f(S) \downarrow F : P}$$

$$\textsc{FInv}\ \frac{f : A = \text{“}T\text{”} \in \Sigma \quad \Gamma; (A)_i \vdash S \downarrow S' : P}{\Gamma \vdash f^i(S) \downarrow f^i(S') : P}$$

$$\textsc{FEvalInv}\ \frac{f : A = \text{“}T\text{”} \in \Sigma \quad \Gamma; A \vdash S \downarrow f^0(S_0), \ldots, f^n(S_n) : P \quad \forall i, j,\ S_i = S_j}{\Gamma \vdash f(S) \downarrow \pi_n(A) \star S_0 : P}$$

$\boxed{\Gamma; A \vdash S \downarrow S' : P}$ Spine

$$\textsc{SNil}\ \frac{}{\Gamma; P \vdash \cdot \downarrow \cdot : P} \qquad \textsc{SCons}\ \frac{\Gamma \vdash M : A \downarrow M' \quad \Gamma; B[x/|M'|] \vdash S \downarrow S' : P}{\Gamma; \Pi x : A.B \vdash M, S \downarrow M', S' : P}$$

Figure 2.4: Call-by-value weak typed evaluation of DLF objects

$\boxed{\Gamma \vdash T \star S \downarrow F : P}$ Canonical term

$$\textsc{TLam}\ \frac{\Gamma \vdash T[\mathtt{x}/M] \star S \downarrow F : P}{\Gamma \vdash \mathbf{fun}\,\mathtt{x} \to T \star M, S \downarrow F : P} \qquad \textsc{TAtom}\ \frac{\Gamma \vdash U \downarrow F : P}{\Gamma \vdash U \star \cdot \downarrow F : P}$$

$\boxed{\Gamma \vdash U \downarrow F : P}$ Atomic term

$$\textsc{UAtom}\ \frac{\Gamma \vdash F \downarrow F' : P}{\Gamma \vdash \langle\!\langle F \rangle\!\rangle \downarrow F' : P} \qquad \textsc{UCase}\ \frac{\Gamma @ \Gamma' \vdash U \downarrow F : P \quad \Gamma \vdash F \star C \downarrow F' : P'}{\Gamma \vdash \mathbf{match}\ U\ \mathbf{under}\ \Gamma'\ \mathbf{with}\ C \downarrow F' : P'}$$

$\boxed{\Gamma \vdash F \star C \downarrow F' : P}$ Branches

$$\textsc{CNoMatch}\ \frac{\mathsf{c} \neq \mathsf{c}' \quad \Gamma \vdash \mathsf{c}(S) \star C \downarrow F : P}{\Gamma \vdash \mathsf{c}(S) \star (\langle\!\langle \mathsf{c}'(\vec{x}) \rangle\!\rangle \Rightarrow U \mid C) \downarrow F : P}$$

$$\textsc{CMatch}\ \frac{\Gamma \vdash U[x_1/M_1, \ldots, x_n/M_n] \downarrow F : P \quad \forall ij, \text{if } i \neq j \text{ then } x_i \neq x_j}{\Gamma \vdash \mathsf{c}(M_1, \ldots, M_n) \star (\langle\!\langle \mathsf{c}(x_1, \ldots, x_n) \rangle\!\rangle \Rightarrow U \mid C) \downarrow F : P}$$

Figure 2.5: Call-by-value weak typed evaluation of DLF term

¶ | READBACK Once an object has been put in weak normal form, the work is not done: there might be some functions left to evaluate under λ-abstractions. We must go over the object once more, and recursively evaluate under these abstractions. This is the role of *readback* (Figure 2.6). All rules should be straightforward, since they only traverse the term maintaining and verifying the type information, except for two rules. First RMLam: meeting an object of the form $\lambda x.M$, we have to guarantee that M is a value (even a weak one), so we fully evaluate it. Secondly RFInv: if a weak value does not contain function calls anymore, it might still contain inverses (because of rule FInv) that were left there for possible contractions. We are now sure that these contractions cannot happen anymore, since there is no function symbols in the spine S.[11] Hence, now is a good time to erase these inverses. This is what the *cut* $\pi_n(A) \star S'$ performs.

¶ | TYPES, KINDS AND SIGNATURES VERIFICATION We did not include judgments for types, kinds, and signatures. This is because they are checked with the exact same algorithm defined for SLF (Figure 1.5), only with DLF syntax. In particular, any constant declarations c containing functions f or inverses f^i will be rejected. This way we can make sure that in the type world, we manipulate only values, and the comparison of two types can be made only up to α-equivalence. This might be a controversial choice. We could imagine a system where some constants' types include functions to evaluate when they are applied, like in:

$$\mathsf{c} \ : \ \Pi x : \mathsf{a}.\, \mathsf{p}\,(f\,(x))$$

We saw the following reasons not to handle this case:

- first, when checking for the equality of two types, we would need to perform evaluation, and thus thread the environment and the type information; it would have complicated greatly the system;
- secondly, it poses a question on the nature of certificates: if I am handed a proof containing constant c above, I need to know the code of f and evaluate it to verify this certificate; is it still a certificate then?
- but most importantly, we could not think of meaningful examples making this feature useful.

¶ | DISCUSSION AND CRITIQUE There are two major inefficiencies we can spot in the evaluation algorithm. First, consider the value

$$\mathsf{c}\,(M, \lambda x.N)$$

[11] There might be some under λ-abstractions, but these λ-abstractions will always be in the way between a function and its inverses.

$\boxed{\Gamma \vdash M : A \Downarrow M' \text{ and } \Gamma \vdash F \Downarrow F' : P}$ Canonical object full evaluation

$$\textsc{SMObj}\ \frac{\Gamma \vdash M : A \downarrow M_1 \qquad \Gamma \vdash M_1 : A \uparrow M_2}{\Gamma \vdash M : A \Downarrow M_2} \qquad \textsc{SFAtom}\ \frac{\Gamma \vdash F \downarrow F_1 : P \qquad \Gamma \vdash F_1 \uparrow F_2 : P}{\Gamma \vdash F \Downarrow F_2 : P}$$

$\boxed{\Gamma \vdash M : A \uparrow M'}$ Canonical object readback

$$\textsc{RMLam}\ \frac{\Gamma, x : A \vdash M : B \Downarrow M'}{\Gamma \vdash \lambda x.\, M : (\Pi x : A.\, B) \uparrow \lambda x.\, M'} \qquad \textsc{RMAtom}\ \frac{\Gamma \vdash F \uparrow P : F'}{\Gamma \vdash F : P \uparrow F'}$$

$\boxed{\Gamma \vdash F \uparrow F' : P}$ Atomic object readback

$$\textsc{RFVar}\ \frac{x : A \in \Gamma \qquad \Gamma; A \vdash S \uparrow S' : P}{\Gamma \vdash x\,(S) \uparrow x\,(S') : P} \qquad \textsc{RFConst}\ \frac{\mathsf{c} : A \in \Sigma \qquad \Gamma; A \vdash S \uparrow S' : P}{\Gamma \vdash \mathsf{c}\,(S) \uparrow \mathsf{c}\,(S') : P}$$

$$\textsc{RFInv}\ \frac{f : A = \text{“}T\text{”} \in \Sigma \qquad \Gamma; (A)_n \vdash S \uparrow S' : P}{\Gamma \vdash f^n\,(S) \uparrow \pi_n(A) \star S' : P}$$

$\boxed{\Gamma; A \vdash S \uparrow S' : P}$ Spine readback

$$\textsc{RSNil}\ \frac{}{\Gamma; P \vdash \cdot \uparrow \cdot : P} \qquad \textsc{RSCons}\ \frac{\Gamma \vdash M : A \uparrow M' \qquad \Gamma; B[x/|M'|] \vdash S \uparrow S' : P}{\Gamma; \Pi x : A.\, B \vdash M, S \uparrow M', S' : P}$$

Figure 2.6: Call-by-value full typed evaluation of DLF

where M is a very large, purely applicative object (containing no abstraction). To evaluate it, we first weakly evaluate it: M is traversed and type checked, and the same term $\mathsf{c}(M, \lambda x.N)$ is returned since M did not contain any function to evaluate. On the other hand, N is not traversed since it is under the abstraction. Then, we must read it back: M is traversed *again* entirely, and N is finally fully evaluated. The fact that M is traversed twice is inherent to full call-by-value: in call-by-name, the weak evaluation step would stop at the constant c without traversing M. This remark is absent of e.g., Grégoire and Leroy [2002] and to the best of our knowledge, it would require further study.

The second inefficiency is that most objects, when passed or returned by a function, will be evaluated/checked several times. Consider for instance the function:

$$\mathit{not} : \mathsf{bool} \to \mathsf{bool} = \mathbf{fun}\ \mathtt{x} \to \mathbf{match}\ \mathtt{x}\ \mathbf{with}\ \text{«}\mathsf{true}\text{»} \Rightarrow \text{«}\mathsf{false}\text{»} \mid \mathsf{false} \Rightarrow \text{«}\mathsf{true}\text{»}$$

The application $\mathit{not}(\mathsf{true})$ will trigger the evaluation of true a first time (rule FEval, second premise), then the pattern-matching will again (rule UCase, first premise). This is inevitable in the general case, since the scrutinee can be any term, not only a computational variable. One solution would be to introduce a new construct in objects, e.g., $\{F : P\}$, tagging the sub-object F as already evaluated and typed of type P. Another, close solution is described in Chapter 3 and amounts to *memoize* manually this procedure; we then gain not only efficiency, but a model for incremental computations on higher-order term structure.

2.3.2 | METATHEORY

But before delving into this, let us expose the basic properties of this language, with its inefficiencies.

A | PROJECTIONS

First, we prove that the "code" of an inverse—the projection object—agrees with its type—the projection type. In an object M, we can substitute an inverse with its projection without altering the type of the object M. This is established by this easy lemma on SLF derivations:

Lemma 2.1. *For all A such that $\Gamma \vdash A$ type and $\pi_n(A)$ is defined for $n \in \mathbb{N}$, we have $\Gamma \vdash (A)_n$ type and $\Gamma \vdash \pi_n(A) \Leftarrow (A)_n$.*

Proof. By easy functional induction on the computation of $\pi_n(A)$, relying on corollary 1.2 for Equation (2.27). □

With that in mind, we can now prove that the *erasure* reduction i.e., collapsing a inverse $f^n(S)$ to its n-th argument, preserves typing. The collapse is computed with a *cut* between the n-th projection $\pi_A(n)$ and S:

Lemma 2.2. *For all A such that $\Gamma \vdash A$ type and S, if $\Gamma; (A)_n \vdash S \Rightarrow F$ then $\Gamma \vdash \pi_n(A) \star S \Rightarrow F$.*

Proof. By induction on A, using the previous lemma and Figure 1.3 □

Similarily, we want to prove that the *contraction* reduction preserves typing, that is if $f\,(f^0\,(S), \ldots, f^n\,(S))$ has type P, then so has the last projection of S, which represent the given result of the computation, $\pi_n(A) \star S$. This translates a little differently in SLF, since there, we do not have functions or inverses. Let us call A the declared type of f; inverses have types $(A)_i$, for $0 \leq i \leq n$; we use variables to represent these inverses:

Lemma 2.3. *For all A such that $\Gamma \vdash A$ type and S, if $\Gamma @ x_0 : (A)_0, \ldots, x_n : (A)_n; A \vdash x_0\,(S), \ldots, x_n\,(S) \Rightarrow P$ then $\Gamma \vdash \pi_n(A) \star S \Rightarrow P$.*

Proof. By lexicographic induction, first on A and then on the given derivation, using the two previous lemmas. □

B | SOUNDNESS

The next property we want to establish is the soundness of the system: if evaluation returns an object, then this object is a value and this value is a well-typed SLF object.

Definition 2.5 (Value)**.** An object M (*resp.* F, S, type A) is a *value* if it injects in the syntax of SLF objects M (*resp.* F, S, type A), i.e., if it does not contain function and inverse names.

Theorem 2.1. *In a signature Σ such that $\vdash \Sigma$ sig, and for all environment Γ such that $\vdash \Gamma$ env:*

1. *If $\Gamma \vdash M : A \Downarrow M'$ and A is a value, then M' is a value, and $\Gamma \vdash M' \Leftarrow A$;*
2. *If $\Gamma \vdash M : A \uparrow M'$ and A is a value, then M' is a value and $\Gamma \vdash M' \Leftarrow A$;*
3. *If $\Gamma \vdash F \uparrow F' : P$, then F' and P are values and $\Gamma \vdash F' \Rightarrow P$;*
4. *If $\Gamma; A \vdash S \uparrow S' : P$, and A value, then S' and P are values and $\Gamma; A \vdash S' \Rightarrow P$.*

Proof. By mutual induction over the provided derivations. In cases RFVar, RFConst and RFInv, we rely on the fact that Σ and Γ contain only value types. In RFInv, we rely on lemma 2.2. □

Corollary 2.1. *If $\Gamma \vdash F \Downarrow F' : P$ then F' and P are values, and $\Gamma \vdash F' \Rightarrow P$.*

Proof. By rule SFAtom and theorem 2.1, clause *3*. □

Note that this property holds without appealing to the definition of weak evaluation. Whatever weak evaluation does to an object, the result is rechecked afterwards, which corresponds to our first requirement in Section B.

Next, we prove that weak evaluation is sound too, i.e., that it always returns a well-typed object. First, we define the codomain of weak evaluation: weak values.

Definition 2.6 (Weak value)**.** A DLF object M (*resp.* F, S) is a *weak value* if it injects in the following grammar:

$$\begin{aligned} M^\circ &::= \lambda x.\, M \mid F^\circ \\ F^\circ &::= H^\circ\,(S^\circ) \\ S^\circ &::= \cdot \mid M^\circ, S^\circ \\ H^\circ &::= \mathsf{c} \mid x \mid f^n \end{aligned}$$

That is, if it contains no function symbols, except under λ-abstractions.

Now, what does it mean for a weak value to be well-typed? It means that the "applicative tip" of the term, that is, everything outside λ-abstractions, is well typed. Since weak values are not *exactly* SLF objects, we cannot use SLF typing directly. We appeal to a slight variation of SLF:

Definition 2.7 (Weak SLF)**.** We define the typing rules of *weak* SLF to be the typing rules of SLF (Figures 1.4 and 1.5) where rule MLam is replaced by:

$$\frac{}{\Gamma \vdash \lambda x.\, M^\circ \Leftarrow \Pi x : A.\, B}\ \textsc{MLam}$$

and the following rule is added for inverse functions:

$$\frac{f : A = \text{“}T\text{”} \in \Sigma \qquad \Gamma; (A)_i \vdash S^\circ \Rightarrow P}{\Gamma \vdash f^i\,(S^\circ) \Rightarrow P}\ \textsc{FInv}$$

In the following theorem, the judgments in the conclusions of the four clauses are, as expected, *weak* SLF judgments:

Theorem 2.2. *In a signature Σ such that $\vdash \Sigma$ sig, and for all environment Γ such that $\vdash \Gamma$ env:*

1. *If $\Gamma \vdash M : A \downarrow M'$ and A is a value, then M' is a weak value, and $\Gamma \vdash M' \Leftarrow A$;*
2. *If $\Gamma \vdash F \downarrow F' : P$, then F' and P are weak values and $\Gamma \vdash F' \Rightarrow P$;*
3. *If $\Gamma; A \vdash S \downarrow S' : P$, and A value, then S' and P are weak values and $\Gamma; A \vdash S' \Rightarrow P$.*
4. *If $\Gamma \vdash U \downarrow F : P$ then F and P are weak values and $\Gamma \vdash F \Rightarrow P$*

Proof. By mutual induction over the provided derivations. In case FEvalInv, we use lemma 2.3 together with Figure 1.3. In case FEval, we use clause *(4)*. Again, cases FVar, FConst, FEval and FInv, rely on the fact that Σ and Γ contain only value types. □

2.4 | CASE STUDIES

We now turn to the study of two full-scale examples of DLF signatures, defining two programs. The first one is a simple but safe automated theorem prover that will demonstrate the versatility of programming with *higher-order abstract syntax* in the production of proof certificates. Like any theorem prover, it needs to manage a database of hypotheses; the resulting certificate is however produced in a format where hypotheses are represented by SLF variables. This is implemented by maintaining at run time a database of *pairs* of hypotheses and their proofs, which can be a simple SLF variable bound in the resulting certificate.

The second one is a safe type checker for a language, System $\mathsf{T}_{<:}$ which type system is well-expressed in a declarative manner, but which type inference algorithm is complex. This use case will demonstrate that letting the type checker emit a certificate in the form of a typing derivation is feasible, and provides safety with respect to the specification—the declarative rules. It will also demonstrate once more the usage of function inverses to implement the traversal of a higher-order term structure, the terms of our language.

Whereas the computational part of DLF was presented above as a minimalist functional programming language, we will make here a rather big leap forward, and assume more than this mere pattern-matching construct; notably, we use OCaml-like exceptions for the first example. When discussing the implementation of our framework in Section 3.4, called Gasp, we will see that this is actually realized: the logical level is (a conservative extension of) SLF, but the computational language *is* OCaml itself. By anticipation, we present our examples implemented in Gasp; the reader must only convince himself that it would be theoretically possible to add the missing features to DLF.

2.4.1 | CERTIFICATE-PRODUCING PROOF SEARCH

The first illustration of our certifying computation framework is the design of a small, certifying automated theorem prover for propositional intuitionistic natural deduction. We suppose an infinite sequence of comparable atomic propositions p, $\mathsf{q}\ldots$ Propositions are built out of the grammar:

$$A, B, C, D, G \ ::= \ \mathsf{p} \mid A \vee B \mid A \wedge B \mid A \supset B \mid \top \mid \bot$$

We will propose a natural way of programming a proof-search algorithm in Gasp which, given a proposition, searches for a proof and, if found, returns a certificate in the form of an LF object representing a natural deduction of the proposition.

A | MOTIVATIONS

Automated theorem provers are programs used to automatically search for the proof of a statement in a particular logic and theory. There are numerous calculi to search for proofs,[12] combined with many different implementation techniques (e.g., *term indexing* [Ramakrishnan et al., 2001] to optimize one-to-many unification), move the actual implementation of theorem provers further away from the actual logic they are searching proofs of.

Many of these tools issue a simple ternary information: they either run forever, or return a boolean indicating if the given goal is provable or not. These tools can have a large codebase, that cannot be exempt from bugs: how can then a user trust their answers? One answer to this legitimate fear is to formally verify a prover e.g., in a proof assistant such as Coq (for instance Lescuyer [2011] proved a SMT prover); this approach has the benefit of ensuring trustability of all answers of the prover, at the expense of the tedious effort of proving the whole program. Another, more lightweight approach is to design the prover so that it generates proof certificates in case of success, in a format that can be transmitted and reverified, either by hand or by a piece of software, usually much smaller than the prover itself (a *kernel*). Many complex pieces of a prover (e.g., the term indexing data structures) do not have to be proved correct then: generally, only the main loop is modified to issue a *trace* of its (successful) run. The risk of a false negative (a non-theorem recognized as one by the prover) is suppressed: soundness is ensured by the independent checker. The possible returns of the package prover + independent proof verifier are the same as before — loop forever, fail (because of the prover or the verifier) or succeed — only the trusted code base, in the case of success, is reduced to just the independent prover.

In the remainder of this section, we follow this last approach and take the (first) pretext of proof search to demonstrate the use of our certificate generation library Gasp. We use a calculus close to the sequent calculus, namely Dyckhoff [1992]'s LJT to implement a simple tautology prover for intuitionistic propositional logic which issues certificates in natural deduction.

B | THE LJT + F CALCULUS

Natural deduction NJ [Gentzen, 1935] is a formulation of intuitionistic propositional logic (we present it on Figure 4.1 in hypothetical style). It is universally accepted, so it should form a good format to issue certificates in. However, it is not adequate

[12] To cite only the most popular, *resolution* [Bachmair and Ganzinger, 2001] or *tableaux* [Hähnle, 2001] looks for proofs in classical logic, *paramodulation* [Nieuwenhuis and Rubio, 2001] for equational theories, *SMT solving* for first-order theories [Nieuwenhuis et al., 2006] or *focusing* [Andreoli, 1992], a proof strategy devised for linear logic [Girard, 1987] but which turns out to be adaptable to intuitionistic, classical and many other propositional logics.

as-is for proof search: reading the rules bottom-up, they do not form an algorithm (not even a non-deterministic one, with need for backtracking): their application obliges to "invent" formulae in the premises that was not present in the conclusion (e.g., the A in ImpE): this system does not have the strict *subformula property*. The sequent calculus LJ (Figure 4.2), also introduced in Gentzen [1935], partially solves this problem: it forms a non-deterministic *semi-algorithm* for proof search. The main source of non-termination comes from the Permut and Contract rules, since they do not *strictly* make the size of the sequent decrease. Permut is not too much trouble, it only indicates that a left rule can select *any* hypothesis in Γ, and not only the first one in the list: environments are *multisets* and not only lists. On the contrary, Contract poses more problems: it makes the multisets into *sets*, and allows to use a hypothesis more than once.

Dyckhoff [1992] shows that this duplication can actually be eliminated by changing the ConjL rule to an admissible, additive version, and refining rule ImpL into four different rules, depending on the syntactic nature of the implication's antecedent. In this modified system LJT, contraction is useless, provided we see environments Γ as multisets (that is, taking Permut as implicit and building permutation into the notation). It thus forms a full-blown, complete decision procedure.

We refine the idea a slight bit further to get our own LJT + F, presented declaratively on Figure 2.7, by introducing a limited form of *focusing*: in LJT, each application of a rule is a choice among all other valid rule applications, and each choice made is subject to backtracking ("don't know" non-determinism). In our system, we restrict this choice by restricting rules that can be applied on the judgment: once we engage in decomposing the goal, there is no rule to switch to decomposing another formula in the list of hypotheses, unless we meet a disjunction or an atomic formula. The system consists in two mutually recursive judgments with an explicit environment Γ, viewed as a *multiset* of formulae:

Judgment $\Gamma \vdash A$: From the multiset of hypothesis Γ, we can derive A. The derivation must start by decomposing A into atomic formulae or disjunctions (i.e., *positive* formulae), then chooses non-deterministically a hypothesis to focus on (rules FocusRi/2). Note that is these two rules, the focused formula is actually taken out from Γ.

Judgment $\Gamma \mid A \vdash C$: From the multiset of hypothesis $\Gamma \cup \{A\}$, we can derive C. The derivation must start by decomposing A.

This system restricts the use of contraction by concentrating on its only problematic occurrence in LJ, i.e., in conjunction with rule ImpL: in this case, we analyze the formula on the left of the implication, and specialize its treatment in each case (rules ImpLi... ImpL4). This way, derivability is exactly preserved, as witnessed by the two following theorems:

$\boxed{\Gamma \vdash A}$ Right rules

$$\textsc{ImpR}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B} \qquad \textsc{DisjR1}\ \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \textsc{DisjR2}\ \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \qquad \textsc{ConjR}\ \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

$$\textsc{TopR}\ \frac{}{\Gamma \vdash \top} \qquad \textsc{FocusR1}\ \frac{\Gamma \mid A \vdash \mathsf{p}}{\Gamma, A \vdash \mathsf{p}} \qquad \textsc{FocusR2}\ \frac{\Gamma \mid C \vdash A \vee B}{\Gamma, C \vdash A \vee B}$$

$\boxed{\Gamma \mid A \vdash A}$ Left rules

$$\textsc{InitL}\ \frac{}{\Gamma \mid \mathsf{p} \vdash \mathsf{p}} \qquad \textsc{DisjL}\ \frac{\Gamma, A \vdash G \quad \Gamma, B \vdash G}{\Gamma \mid A \vee B \vdash C} \qquad \textsc{ConjL}\ \frac{\Gamma, A, B \vdash G}{\Gamma \mid A \wedge B \vdash G}$$

$$\textsc{ImpL1}\ \frac{\Gamma, \mathsf{p}, A \vdash G}{\Gamma, \mathsf{p} \mid \mathsf{p} \supset A \vdash G} \qquad \textsc{ImpL2}\ \frac{\Gamma, A \supset (B \supset C) \vdash G}{\Gamma \mid (A \wedge B) \supset C \vdash G} \qquad \textsc{ImpL3}\ \frac{\Gamma, A \supset C, B \supset C \vdash G}{\Gamma \mid (A \vee B) \supset C \vdash G}$$

$$\textsc{ImpL4}\ \frac{\Gamma, B \supset C \vdash A \supset B \quad \Gamma, C \vdash G}{\Gamma \mid (A \supset B) \supset C \vdash G} \qquad \textsc{BotL}\ \frac{}{\Gamma \mid \bot \vdash G}$$

Figure 2.7: The LJT + F proof search algorithm, declaratively

Theorem 2.3 (Soundness). *If* $\Gamma \vdash A$ *then* $\begin{array}{c}[\Gamma]\\ \mathscr{D}\\ \vdash A\end{array}$ *; if* $\Gamma \mid A \vdash C$ *then* $\begin{array}{c}[\Gamma]\quad[\vdash A]\\ \mathscr{D}\\ \vdash C\end{array}$.

Proof. By mutual induction on the derivations of $\Gamma \vdash A$ and $\Gamma \mid A \vdash C$. We treat only the few non-trivial cases:

Case ConjL: By induction, $\begin{array}{c}\vdash A\quad\vdash B\\ \mathscr{D}\\ \vdash C\end{array}$. We build $\dfrac{\dfrac{\vdash A \wedge B}{\vdash A}\,\textsc{ConjEi}\qquad \dfrac{\vdash A \wedge B}{\vdash B}\,\textsc{ConjE2}}{\begin{array}{c}\mathscr{D}\\ \vdash A\end{array}}$.

Case ImplI1: By induction, $\begin{array}{c}\vdash \mathsf{p}\quad\vdash A\\ \mathscr{D}\\ \vdash G\end{array}$. We build $\dfrac{\vdash \mathsf{p} \qquad \dfrac{\vdash \mathsf{p} \quad \vdash \mathsf{p} \supset A}{\vdash A}\,\textsc{ImpE}}{\begin{array}{c}\mathscr{D}\\ \vdash G\end{array}}$

Case ImplI2: By induction, $\begin{array}{c}\vdash A \supset B \supset C\\ \mathscr{D}\\ \vdash G\end{array}$. We build $\dfrac{\dfrac{\dfrac{\dfrac{[\vdash A]\quad[\vdash B]}{\vdash A \wedge B}\,\textsc{ConjI} \qquad \vdash A \wedge B \supset C}{\vdash C}\,\textsc{ImpE}}{\vdash B \supset C}\,\textsc{ImpI}}{\vdash A \supset B \supset C}\,\textsc{ImpI}}{\begin{array}{c}\mathscr{D}\\ \vdash G\end{array}}$

Case ImplL3: By induction, $\begin{array}{c}\vdash A \supset C\quad\vdash B \supset C\\ \mathscr{D}\\ \vdash G\end{array}$. We build:

$$\dfrac{\vdash A \vee B \supset C \qquad \dfrac{[\vdash A]}{\vdash A \vee B}\,\textsc{DisjI1}}{\dfrac{\vdash C}{\vdash A \supset C}\,\textsc{ImpI}}\,\textsc{ImpE} \qquad \dfrac{\vdash A \vee B \supset C \qquad \dfrac{[\vdash B]}{\vdash A \vee B}\,\textsc{DisjI2}}{\dfrac{\vdash C}{\vdash B \supset C}\,\textsc{ImpI}}\,\textsc{ImpE}$$

Case ImplL4: By induction, $\begin{array}{c}\vdash B \supset C\\ \mathscr{D}_1\\ \vdash A \supset B\end{array}$ and $\begin{array}{c}\vdash C\\ \mathscr{D}_2\\ \vdash G\end{array}$. We build:

$$\dfrac{\dfrac{[\vdash C]}{\begin{array}{c}\mathscr{D}_2\\ \vdash G\end{array}}}{\vdash C \supset G}\,\textsc{ImpI} \qquad \dfrac{\vdash (A \supset B) \supset C \qquad \dfrac{\dfrac{[\vdash (B \supset C)]}{\begin{array}{c}\mathscr{D}_1\\ \vdash A \supset B\end{array}}}{\vdash (B \supset C) \supset A \supset B}\,\textsc{ImpI} \qquad \dfrac{\vdash A \supset B \supset C \qquad \dfrac{\dfrac{[\vdash B]}{\vdash A \supset B}\,\textsc{ImpI}}{\vdash B \supset C}\,\textsc{ImpI}}{\vdash A \supset B}\,\textsc{ImpE}}{\vdash C}\,\textsc{ImpE}}{\vdash G}\,\textsc{ImpE}$$

□

Theorem 2.4 (Completeness). *If* $\begin{array}{c}[\Gamma]\\ \mathscr{D}\\ \vdash A\end{array}$ *then* $\Gamma \vdash A$.

We skip the proof of this last theorem since it is of little use for our purpose: our framework will not ensure it anyway. On the contrary, soundness plays a crucial part in the construction of the certifying prover, as we will see now.

C | THE LJT + F PROVER IN Gasp

We now present the code for our prover, written in Gasp. It is presented as a DLF signature with a few syntactic additions, since our actual library is based on OCaml and inherits from it much more than just pattern-matching as a computational language. We will use exceptions extensively here for backtracking. First, we declare a few useful types:

```
bool : ∗.
tt : bool.
ff : bool.
nat : ∗.
z : nat.
s : nat → nat.
atom : ∗.
top : atom.
bot : atom.
n : nat → atom.
```

Atomic propositions are indexed by natural numbers nat, and for commodity we include them together with $\top$ and $\bot$ in the type atom.[13] We will need later on to decide their equality, so we define the two functions:

```
eqnat : nat → nat → bool = " fun x y → match x, y with
  | « z », « z » → « tt »
  | « s "x" », « s "y" » → « eqnat "x" "y" »
  | _ → « ff »".
eq : atom → atom → bool = " fun x y → match x, y with
  | « top », « top » | « bot », « bot » → « tt »
  | « n "x" », « n "y" » → « eqnat "x" "y" »
  | _ → « ff »".
```

The declaration of propositions o and natural deductions pf (A) has already been seen in Section 1.1:

```
o : ∗.
at : atom → o.
imp : o → o → o.
conj : o → o → o.
disj : o → o → o.

pf : o → ∗.
TopI : pf (at top).
```

[13] This is acceptable in the context of intuitionistic logic, although generally incorrect.

```
BotE : ΠA : o. pf (at bot) → pf A.
ImpI : ΠA : o. ΠB : o. (pf A → pf B) → pf (imp A B).
ImpE : ΠA : o. ΠB : o. pf (imp A B) → pf A → pf B.
ConjI : ΠA : o. ΠB : o. pf A → pf B → pf (conj A B).
ConjE1 : ΠA : o. ΠB : o. pf (conj A B) → pf A.
ConjE2 : ΠA : o. ΠB : o. pf (conj A B) → pf B.
DisjI1 : ΠA : o. ΠB : o. pf A → pf (disj A B).
DisjI2 : ΠA : o. ΠB : o. pf B → pf (disj A B).
DisjE : ΠA : o. ΠB : o. ΠC : o.
  pf (disj A B) → (pf A → pf C) → (pf B → pf C) → pf C.
```

Our algorithm works by constructing NJ proofs both "in hypotheses" and "in the goal" of the LJT + F judgments. Type hyps encodes a list of hypotheses together with their proofs, and function *rev_append* appends a hyps at the end of another in reverse order:

```
hyps : ∗.
nil : hyps.
cons : ΠA : o. pf A → hyps → hyps.
append : hyps → hyps → hyps = " fun xs ys → match xs with
  | « nil » → ys
  | « cons "x" "a" "xs" » → « append "xs" (cons "x" "a" "ys") ».
```

All the following functions are partial: they can either return their result, or fail with an exception. Function *assumption* searches for the proof of an atom in the current list of hypotheses (it is used by rule ImpLi). We use a predefined OCaml exception Failure to signal that it was not found, and an **if** construct on our own booleans as syntactic sugar for pattern-matching.

```
assumption : hyps → ΠP : atom. pf (at P) = " fun hs p → match hs with
  | « nil » → raise Failure
  | « cons (at "q") "m" "hs" » → if « eq "p" "q" » then m else « assumption "hs" "p" »
  | _ → « assumption "hs" "p" ».
```

Finally, our main loop consists of three mutually recursive functions: *search* corresponding to right rules, *focus* corresponding to left rules and *select* which role is to try successively to *focus* on every hypotheses of the current list, backtracking to the next one if it failed. Function *search* is the entry point of our algorithm. It takes a hyps and a formula, and potentially returns a proof of it:

```
search : hyps → ΠA : o. pf A = " fun hs a → match a with
```

The first four case should be self-explanatory, since right rules of LJT + F follow NJ introductions: for each connective, we return the corresponding introduction rule, where sub-proofs are replaced by recursive calls to *search*. In the implication case $A \supset B$, this call takes a list hyps extended by A together with its proof, which is given

by the hypothesis of NJ's rule ImpI:

```
| « conj "a" "b" » ⟶ « ConjI "a" "b" (search "hs" "a") (search "hs" "b") »
| « at top » ⟶ « TopI »
| « at "p" » ⟶ « select "hs" nil (at "p") »
| « imp "a" "b" » ⟶ « ImpI "a" "b" λx.search (cons "a" x "hs") "b" »
```

The case $A \vee B$ is a little bit trickier, since it is a ternary point of backtrack: we can apply either FocusR2 (this rule is transparent, so we return directly the certificate returned by select), DisjL1 (justified by rule DisjI1 provided the recursive call to *search* succeeds) or DisjL2 (*resp.* DisjI2):

```
| « disj "a" "b" » ⟶
    try let x = « select "hs" nil (disj "a" "b") » in x with Failure ⟶
    try let x = « search "hs" "a" » in « DisjI1 "a" "b" "x" » with Failure ⟶
    « DisjI2 "a" "b" (search "hs" "b") »".
```

Function *select* takes two hyps: the "active" list of hypotheses still worth to focus on, and the "passive" one that have already failed to prove the goal. Each time we focus on a formula, it is discarded from the "active" ∪ "passive" list, since LJT + F is *contraction-free*.

```
select : hyps ⟶ hyps ⟶ ΠC : o. pf C = " fun xs ys c ⟶ match xs with
  | « nil » ⟶ raise Failure
  | « cons "a" "m" "xs" » ⟶
    try let x = « focus (append "ys" "xs") "a" "m" "c" » in x with Failure ⟶
    « select "xs" (cons "a" "m" "ys") "c" »".
```

Now comes the most interesting function *focus*. Its type is close to the left rule's judgment, except that each formula is annotated by a natural deduction: it takes a hyps as usual, the active formula A and its proof, and a formula G to prove. If $A = \bot$, we conclude right away with BotE; if $A = \mathsf{p}$, then G must be p, in which case we conclude with the proof attached to the hypothesis we have at hand:

```
focus : hyps ⟶ ΠA : o. pf A ⟶ ΠG : o. pf G = " fun hs a m g ⟶ match a with
  | « at bot » ⟶ « BotE "g" "m" »
  | « at "q" » ⟶ (match g with
    | « at "p" » when « eq "p" "q" » ⟶ m
    | _ ⟶ raise Failure)
```

For disjunction $A \vee B$, we go back to "left mode" (function *search*) proving G twice each time putting back respectively A and B in the hypothesis pool. We conclude by rule DisjE, which provides the hypotheses $\vdash A$ and $\vdash B$ as variables.

```
| « disj "a" "b" » ⟶
  « DisjE "a" "b" "g" "m" (λx.search (cons "a" x "hs") "g")
```

```
        (λx. search (cons "b" x "hs") "g") »
```

Conjunction $A \wedge B$ follows the same pattern: we go back to "left mode" with two new hypotheses A and B. This time, a non-trivial justification for these hypotheses must be constructed, with rules ConjE1 and ConjE2.

```
    | « conj "a" "b" » ⟶
      « search (cons "a" (ConjE1 "a" "b" "m") (cons "b" (ConjE2 "a" "b" "m") "hs")) "g" »
```

If the active formula is an implication $A \supset B$, then we must discriminate on A. If $A = \mathsf{p}$, we must *first* make sure that it is in the hypotheses, [14] before *search*ing for B with the new hypothesis p. The justification for it is constructed in the hypothesis with ImpE. The other cases are similar: they build justifications for hypotheses following closely the proof of theorem 2.3:

```
    | « imp "a" "b" » ⟶ match a with
      | « at "p" » ⟶ let n = « assumption "hs" "p" » in
        « search (cons "b" (ImpE "a" "b" "m" "n") "hs") "g" »
      | « conj "c" "d" » ⟶
        « search (cons (imp "c" (imp "d" "b"))
            (Impl "c" (imp "d" "b") λpc. Impl "d" "b" λpd.
              ImpE (conj "c" "d") "b" "m" (ConjI "c" "d" pc pd)) "hs") "g" »
      | « disj "c" "d" » ⟶
        « search
          (cons (imp "c" "b") (Impl "c" "b" λpc. ImpE (disj "c" "d") "b" "m" (DisjI1 "c" "d" pc))
          (cons (imp "d" "b") (Impl "d" "b" λpd. ImpE (disj "c" "d") "b" "m" (DisjI2 "c" "d" pd))
           "hs")) "g" »
      | « imp "c" "d" » ⟶
        « ImpE "b" "g"
          (Impl "b" "g" λpb. (search (cons "b" pb "hs") "g"))
          (ImpE (imp "c" "d") "b" "m"
            (Impl "c" "d" λpc.
              (ImpE "c" "d"
                (ImpE (imp "d" "b") (imp "c" "d")
                  (Impl (imp "d" "b") (imp "c" "d") λpdb.
                    (search (cons (imp "d" "b") pdb "hs") (imp "c" "d")))
                  (Impl "d" "b" λpd.
                    ImpE (imp "c" "d") "b" "m"
                      (Impl "c" "d" λpc. pd))) pc))) ».
```

The last case $((C \supset D) \supset B)$ is the trickiest: it requires to search for two additional proofs of G and $C \supset D$, corresponding to the induction hypotheses in the proof.

[14] The **let** binding is not required, but will ensure that we fail early in *assumption* when it is not found, before continuing searching for B.

The implementation of our prover is over. We can for instance ask the system to evaluate the DLF object:

```
search nil (imp (imp (disj (at (n z)) (imp (at (n z)) (at bot))) (at bot)) (at bot))
```

and get the object coding for the NJ proof of the double negation of the excluded middle $\vdash ((p \vee (p \supset \bot)) \supset \bot) \supset \bot$, with guarantee that it is well-typed, and even that each intermediate objects, at each step of the computation is too. Thus, if there is a bug in the code above, a dynamic check will stop its execution with an error exactly where the ill-typed certificate will have been generated. As we saw, the structure of the code follows very closely the proof of soundness of the system.

2.4.2 | SAFE TYPING

Our second illustration is the design of a certifying type checker for a small programming language based of system F. If it is already a full-blown application in itself, it will actually take even more sense in Chapter 3, where we show how to turn it "for free" into an incremental type checker.

A | MOTIVATIONS

Type checkers, as found in front-ends of compilers, have grown to be increasingly complex pieces of software, to the point that they become almost their central module for some strongly-typed languages. For instance, about 30% of the code of the OCaml compiler is dedicated to typing.[15] On them depends critically the safety of the compiled code: a type checker answering positively in presence of an ill-typed term is subject to produce unsafe code. How can I trust the answer given by my type checker?

A lightweight way to design a trusted type checker is to let it return a *certificate of well-typing*, and let a small, trusted base of code check this certificate a posteriori. The proof assistant Agda [Norell, 2007] and the Haskell compiler GHC [Peyton Jones et al., 1993] adopt this architecture internally. GHC for instance takes Haskell programs which are very scarce in type annotations; from the source program P, it runs a complex *type inference* procedure *infer* which produces a term M in an explicity typed variant of System F called F_C (see Vytiniotis [2008, chapter 3] and Sulzmann et al. [2007]). It is then *type checked*: provided M is no more than P "decorated" with type annotations, and that it is well-typed, then we know that the type inference procedure was correct. Note that the type checker for System F is a very simple program, surely orders of magnitudes smaller and trustworthier than the type inference subsystem. In that, M acts as a certificate, and *infer* as a *certifying*

[15] The compiler itself is 93 KLOC (counting out external and build tools and the standard library), whereas the `typing/` directory is 26.

type checker.

Let us present another view on the same problem. Type systems are usually presented in a declarative manner as a set of inference rules, together with proofs that all programs accepting a derivation in this system respect a certain dependability claim, e.g., the famous "progress and preservation" pair. Yet, actual algorithms of verification often differ very much from their declarative description. This is of course true when programs are provided with little type annotations as in the previous example ("type inference"), in which case the missing information has to be algorithmically synthesized, but is also true in many cases for mere "type checking", or explicit "church-style" calculi. Take for example dependent type systems, relying on definitional equality on types, or type systems with subtype polymorphism: both feature a rule of the form

$$\frac{\vdash M : A' \qquad \vdash A' \leq A}{\vdash M : A}\ \textsc{Sub}$$

(for a given notion of $\leq$) which is not syntax-directed: we could apply it to any given term M, thus it does not make this set of rule an algorithm. The implementer has to turn the declarative system into an algorithm, and convince himself that both are equivalent. For instance, he can show that it is sufficient to change the usual application rule to the "subtyping-aware" application rule where:

$$\frac{\vdash M : A \to B \qquad \vdash N : A' \qquad \vdash A' \leq A}{\vdash M\ N : B}\ \textsc{AppSub}$$

This task is highly non-trivial and non-compositional. For instance with subtyping, every eliminator (e.g., pattern-matching) must be adapted to get a syntax-directed presentation; if we later on add another non-syntax-directed rule, we will have to adapt the whole system again. Often, the algorithm obtained is much larger and more complicated than the declarative starting point. We end up with the same trust problem as in the type inference case, and with a similar observation: it is difficult to turn the declarative specification of a type system into an algorithm, but it is easy to check whether a typing derivation in this declarative system is correct.

In the following, we show how to design in Gasp a certifying type checker for such a system with subtyping. Given a term M, it will try to compute a certificate of well-typing. Contrarily to the GHC example, this certificate will be issued as an SLF term, coding for *a typing derivation in the declarative system*: its specification will be something along the line of $\Pi M : \mathsf{tm}.(\vdash M : A)$. There are three clear advantages to use Gasp to generate SLF certificates:

- For one, GHC's type inference has a type Haskell $\to \mathsf{F_C}$ so a verification has to be made *a posteriori* whether its input and output were actually related (represented

the same program, differing only by annotations). This connection is expressed in our type, thanks to the dependent product.

- Besides, GHC's trusted base is the checker for F_C, an intermediate language used only in GHC; our trusted base, the SLF checker, is generic and can be used for any typed object language.
- Finally, "just-in-time" certificate check (Section 2.2) allows to detect ill-typed certificates exactly when there are produced (and not a posteriori when checking the whole certificate), which helps to develop the program.

The comparison stops here of course, as the span of this example is much less ambitious than to infer types for full Haskell.

B | SYSTEM $\mathsf{T}_{<:}$

How to infer the type of a System T recursor in presence of subtyping? System T is a simply-typed λ-calculus with built-in naturals and a recursor, whose typing rule is (in *local notation*):

$$\frac{\Gamma \vdash M_0 : \mathsf{nat} \qquad \Gamma \vdash M_1 : A \qquad \Gamma, x : \mathsf{nat}, y : A \vdash M_2 : A}{\vdash \mathbf{rec}(M_0, M_1, xy.\, M_2) : A}\ \textsc{Rec}$$

This rule is syntax-directed, considering environment Γ and term M as input and type A as output: the type A of M_1 is inferred, and only then added back to the environment to infer the type of M_2. How must this rule be adapted if we add rule Sub, with a decidable subtyping relation $\leq$? Inferring the type A_0 of M_0, we could end up with a type less general than nat, which is fine: in that case we apply Sub. The problem is that we have to guess a type A more general than that of M_1 and M_2 , while giving that very type to variable y *when typing* M_2! There seem to be a breach of causality there...

There is a well-known algorithm[16] if the subtyping relation forms a complete lattice (if there is a most and least general type): iterate the typing of M_2, starting with $y : A$ and assigning to y each time the *join* of the previously computed type and the new type for M_2 until a fixed point is reached, i.e., until $\Gamma, x : \mathsf{nat}, y : A' \vdash M_2 : A'$. Then A' is the type of the whole recursor. This argument is hard to justify, but it is easy to verify a given derivation for it in the declarative system.

In Figure 2.8, we expose System $\mathsf{T}_{<:}$, this time in *hypothetical notation* (since we are going to represent them in SLF). It is an extension of System T with three base types nat, even and odd, the last two being subtypes of the first.[17] Naturals can be

[16] This trick is adapted from work on type inference in presence of polymorphism [Henglein, 1993], which is undecidable; its adaptation to our subtyping makes it finitely computable.

[17] Type nat is the most general type. For concision, we do not include a least general type $\bot$, and prefer to fail when computing the least general type of e.g., even and odd.

constructed with **o** and **s**(M), and will be assigned type even or odd. Function application and the recursor will "do their best" to preserve this precise type information, but will fall back on type nat if they can't (this is "just" subtyping, not intersection types): for instance, $\vdash$ **rec**(**o**, **o**, $xy.$ **s**(**s**(x))) : even, but $\vdash$ **rec**(**o**, **o**, $xy.$ **s**(x)) : nat. As usual, the subtyping relation is extended to functional types by the SubArr rule, contravariant on the domain of functions and covariant on their codomain. Just for fun, we add a **let** construct.

C | THE $T_{<:}$ TYPE CHECKER IN Gasp

Let us now turn to the implementation of the certificate-generating type checker for $T_{<:}$ in Gasp.

¶ | SIGNATURE We begin by declaring the syntax of terms tm and types tp:

```
tp : ∗.
nat : tp.
even : tp.
odd : tp.
arr : tp → tp → tp.

tm : ∗.
lam : tp → (tm → tm) → tm.
app : tm → tm → tm.
o : tm.
s : tm → tm.
letb : tm → (tm → tm) → tm.
recb : tm → tm → (tm → tm → tm) → tm.
```

For the **rec** construct, we actually have two choices for its third term argument: either making it hypothetical in two terms (x and y) as we did, or suppose it is a simple tm of type (arr nat (arr A) A). We chose the former because it allows less typing annotations. Now come the encoded typing rules, separated in two judgments encoded as type families sub(A, B) (subtyping) and is(M, A) (typing):

```
sub : tp → tp → ∗.
SubRefl: ΠA : tp. sub A A.
SubEven : sub even nat.
SubOdd : sub odd nat.
SubArr: ΠA B C D : tp. sub C A → sub B D → sub (arr A B) (arr C D).

is : tm → tp → ∗.
App : ΠM N : tm. ΠA B : tp. is M (arr A B) → is N A → is (app M N) B.
```

$$A, B ::= A \to B \mid \mathsf{nat} \mid \mathsf{even} \mid \mathsf{odd} \qquad \text{Types}$$
$$M ::= \lambda x : A.\, M \mid M\ M \mid x \mid \mathbf{o} \mid \mathbf{s}(M) \mid \mathbf{rec}(M, M, xy.\, M) \mid \mathbf{let}\ x = M\ \mathbf{in}\ M \qquad \text{Terms}$$

$\boxed{\vdash M : A}$ Term typing

$$\frac{[\vdash x : A] \quad \vdots \quad \vdash M : B}{\vdash \lambda x : A.\, M : A \to B}\ \textsc{Lam} \qquad \textsc{App}\ \frac{\vdash M_1 : A \to B \quad \vdash M_2 : A}{\vdash M_1\ M_2 : B} \qquad \textsc{O}\ \frac{}{\vdash \mathbf{o} : \mathsf{even}}$$

$$\textsc{Sn}\ \frac{\vdash M : \mathsf{nat}}{\vdash \mathbf{s}(M) : \mathsf{nat}} \qquad \textsc{So}\ \frac{\vdash M : \mathsf{even}}{\vdash \mathbf{s}(M) : \mathsf{odd}} \qquad \textsc{Se}\ \frac{\vdash M : \mathsf{odd}}{\vdash \mathbf{s}(M) : \mathsf{even}}$$

$$\textsc{Sub}\ \frac{\vdash M : A \quad \vdash A \leq B}{\vdash M : B} \qquad \frac{\vdash M_0 : \mathsf{nat} \quad \vdash M_1 : A \quad \begin{matrix}[\vdash x : \mathsf{nat}] \quad [\vdash y : A] \\ \vdots \\ \vdash M_2 : A\end{matrix}}{\vdash \mathbf{rec}(M_0, M_1, xy.\, M_2) : A}\ \textsc{Rec}$$

$$\frac{\vdash M_0 : A \quad \begin{matrix}[\vdash x : A] \\ \vdots \\ \vdash M_1 : C\end{matrix}}{\vdash \mathbf{let}\ x = M_0\ \mathbf{in}\ M_1 : C}\ \textsc{Let}$$

$\boxed{\vdash A \leq B}$ Subtyping

$$\textsc{SubEven}\ \frac{}{\vdash \mathsf{even} \leq \mathsf{nat}} \qquad \textsc{SubOdd}\ \frac{}{\vdash \mathsf{odd} \leq \mathsf{nat}} \qquad \textsc{SubRefl}\ \frac{}{\vdash A \leq A} \qquad \textsc{SubArr}\ \frac{\vdash A' \leq A \quad \vdash B \leq B'}{\vdash A \to B \leq A' \to B'}$$

Figure 2.8: Definition of $\mathsf{T}_{<:}$

```
Lam : ΠM : tm → tm. ΠA B : tp. ΠB : tp.
  (Πx : tm. is x A → is (M x) B) → is (lam A λu.M u) (arr A B).
O : is o even.
Se : ΠM : tm. is M odd → is (s M) even.
So : ΠM : tm. is M even → is (s M) odd.
Sn : ΠM : tm. is M nat → is (s M) nat.
Let : ΠM : tm. ΠN : tm → tm. ΠA B : tp.
  is M A → (Πx : tm. is x A → is (N x) B) → is (letb M λx.N x) B.
Rec : ΠM N : tm. ΠP : tm → tm → tm. ΠA : tp.
  is M nat → is N A → (Πx y : tm. is x nat → is y A → is (P x y) A) →
  is (recb M N λx. λy.P x y) A.
Sub : ΠM : tm. ΠA B : tp. sub A B → is M A → is M B.
```

Our type checker will take a term M and possibly return the pair of a type A and a derivation $\vdash M : A$; as in Section 2.4.1 we need to pair up these two results into an encoded Σ-type inf: it is the role of constant ex.

```
inf : tm → ∗.
ex : ΠM : tm. ΠA : tp. ΠH : is M A. inf M.
```

¶ SUBTYPING FUNCTIONS Two helper functions follow: *subt* computes the subtyping derivation $\vdash A \leq B$ of two types A and B, or fails if $A \not\leq B$; *subm* tries to coerce a derivation $\vdash M : A$ into a derivation $\vdash M : B$, or fails if $A \not\leq B$; *join* (*resp.* *meet*) computes the *join* $A \sqcap B$ (resp *meet* $A \sqcup B$) of two types A and B. For concision, we use pattern-matching failure to signal errors:

```
subt : ΠA B : tp. sub A B = " fun a b → match a, b with
  | « even », « even » | « nat », « nat » | « odd », « odd » → « SubRefl "a" »
  | « odd », « nat » → « SubOdd »
  | « even », « nat » → « SubEven »
  | « arr "a1" "b1" », « arr "a2" "b2" » →
    « SubArr "a1" "b1" "a2" "b2" (subt "a2" "a1") (subt "b1" "b2") ».

subm : ΠM : tm. ΠA B : tp. is M A → is M B = " fun m a b d → match « subt "a" "b" » with
  | « SubRefl "_" » → d
  | s → « Sub "m" "a" "b" "s" "d" ».

join : tp → tp → tp = " fun a b → match a, b with
  | « nat », _ → « nat »
  | « even », « even » → « even » | « even », _ → « nat »
  | « odd », « odd » → « odd » | « odd », _ → « nat »
  | « arr "a1" "b1" », « arr "a2" "b2" » → « arr (meet "a1" "a2") (join "b1" "b2") ».
```

```
meet : tp → tp → tp = " fun a b → match a, b with
  | « nat », _ → b
  | « even », (« even » | « nat ») → « even »
  | « odd », (« odd » | « nat ») → « odd »
  | « arr "a1" "b1" », « arr "a2" "b2" » → « arr (join "a1" "a2") (meet "b1" "b2") ».
```

Note that function *meet* is incomplete: even ⊔ odd is undefined since there is no bottom type to our lattice of subtypes.

¶ | THE TYPE CHECKER The type checker *infer* takes a term M to a pair of a type A and a derivation $\vdash M : A$. We pattern-match on input term M. The first two cases should be easily grasped: **o** has type even which is justified by rule O; **s**(M) has a type that depends on the type of M so we match on the recursive call to *infer*(M). The app case is very similar:

```
infer : ΠM : tm. inf M = " fun m → match m with
  | « o » → « ex o even O »
  | « s "m" » →
    let « ex "_" "a" "d" » = « infer "m" » in
    (match a with
      | « nat » → « ex (s "m") nat (Sn "m" "d") »
      | « even » → « ex (s "m") odd (So "m" "d") »
      | « odd » → « ex (s "m") even (Se "m" "d") »)
  | « app "m" "n" » →
    let « ex "_" (arr "a" "b") "d1" » = « infer "m" » in
    let « ex "_" "a'" "d2" » = « infer "n" » in
    « ex (app "m" "n") "b" (App "m" "n" "a" "b" "d1" (subm "n" "a'" "a" "d2")) »
```

Note that in this last case (application $M\ N$), we retrieve types $A \to B$ of M, A' of N, and the respective derivations, and then use function *subm* to construct the argument's derivation: it will evaluate either to $\mathscr{D}_2$ if $A = A'$, to a Sub rule if $A \leq A'$ or raise an error otherwise.

The $\lambda x : A.M$ case is the most emblematic of our type checker, since lam is a constant with a higher-order type, using HOAS to represent open terms. Type checking the open term M demonstrates the first usage of inverses:

```
  | « lam "a" "m" » →
    let « ex "_" "b" "d" » under «^env x : tm; h : is x "a" » =
      « infer ("m" (infer^0 x (ex x "a" h))) » in
    « ex (lam "a" "m") (arr "a" "b") (Lam "m" "a" "b" (λx. λh. "d")) »
```

First, we *infer* the type B and derivation $\mathscr{D}$ for M, but replacing the free variable of M by "what *infer* should return when it meets this variable". Remember: there is no variable case. Actually, even if there was, we would be well distraught to tell

what type a variable has without an environment! What should *infer* return when it meets this variable? the pair of A and the derivation h for it. This is the sense of the argument to M: (*infer*^{0}) x (ex x "a" h). Now these x and h should be bound locally for this call to *infer* to be well-typed. We use the **match** M **under** Γ **with** C construct (actually written **let** because its pattern is irrefutable), with $\Gamma = x : \mathsf{tm}, h : \mathsf{is}(x, A)$. The result of this pattern-matching, B and $\mathscr{D}$, potentially contains free variables x and h; on the contrary, we know that B does not since it is a type (this is a meta-argument on the system). In the returned object, we use B as the return type, and "close" $\mathscr{D}$, using the hypotheses x and h provided by rule Lam.

The **let** case works almost identically:

```
| « letb "m" "n" » →
  let « ex "_" "a" "d1" » = « infer "m" » in
  let « ex "_" "b" "d2" » under «env x:tm; h:is x "a" » =
    « infer ("n" (infer⁰ x (ex x "a" h))) » in
  « ex (letb "m" "n") "b" (Let "m" "n" "a" "b" "d1" (λx. λh. "d2")) »
```

Finally, the **rec**$(M_0, M_1, xy.\, M_2)$ case should not be a surprise either: we infer types A_0 and A_1 for M_0 and M_1, then use A_1 to infer the type A_2 of M_2 with a **let** "under context" with four bindings (two variables and two derivations for them). The two types A_1 and A_2 potentially differ, so we call *infer* once more on M_2, to be sure it reached a fixed point (nat). We know that it did with only two iterations since our lattice is of size two. We then return the pair of this computed type and the derivation formed with rule Rec and calls to *subm* to ensure that types are compatible.

```
| « recb "m" "n" "p" » →
  let « ex "_" "tm" "dm" » = « infer "m" » in
  let « ex "_" "tn" "dn" » = « infer "n" » in
  let « ex "_" "tp" "_" » under «env x : tm; hx : is x nat; y : tm; hy : is y "tn" » =
    « infer ("p" (infer⁰ x (ex x nat hx)) (infer⁰ y (ex y "tn" hy))) » in
  let « "a" » = « join "tn" "tp" » in
  let « ex "_" "tp" "dp" » under «env x:tm; hx:is x nat; y:tm; hy:is y "a" » =
    « infer ("p" (infer⁰ x (ex x nat hx)) (infer⁰ y (ex y "a" hy))) » in
  « ex (recb "m" "n" "p") "a"
    (Rec "m" "n" "p" "a" (subm "m" "tm" nat "dm") (subm "n" "tn" "a" "dn")
      λx. λy. λhx. λhy. (subm ("p" x y) "tp" "a" "dp")) ».
```

¶ | USAGE The type checker is over. We can now ask the system to infer the type of the Ackermann function:

```
infer (lam nat λm. recb m (lam nat λx. s x) λ_. λf.
         lam nat λn. recb n (app f (s o)) λ_. λg. app f g)
```

to what it will answer: (skipping the SLF objects for this term and its derivation)

```
ex (...) (arr nat (arr nat nat)) (...)
```

or of the operation 2^3:

```
infer (letb (lam nat λx. lam nat λy. recb x y λz. λ_. s z) λadd.
       letb (lam nat λx. lam nat λy. recb o y λz. λ_. app (app add x) z) λmult.
       letb (lam nat λx. lam nat λy. recb (s o) y λz. λ_. app (app mult x) z) λexp.
       letb (lam nat λx. recb o x λ_. λw. w) λpred.
       app (app exp (s o)) (s (s o)))
```

which results in:

```
ex (...) nat (...)
```

The computed type is guaranteed to be right with respect to the rules of system $\mathsf{T}_{<:}$, the accompanying typing derivation being the certificate of this fact.

Two kinds of errors can happen. If the user gives a term to type that is not typabe (say `lam nat λx. app x x`), then an exception will be raised *at computation time*, i.e., in the code of a function (here a match failure, but more appropriate error messages could be easily added). If, on the other hand, there is a typo in the code of the type checker (e.g., it returns an invalid certificate), then an exception will be raised *at typing time*. Since this typing is performed right after the certificate is returned, we can easily pinpoint the place in the code where it was generated.

We saw in this case study one usage of the inverse function infer^0 to indicate *at bind time* what should be the output of *infer* on the variable, allowing the type checker not to carry an explicit environment.

2.5 | RELATED AND FURTHER WORK

The DLF language is a core functional language specialized in the generation and manipulation of proof certificates. It is atypical by several aspects: whereas it manipulates proofs, it is not logic programming; while its values—SLF objects—obey a strong typing discipline, it itself checks them dynamically; although it is based on higher-order abstract syntax and thus manipulates open objects, it avoids having to deal with an environment of free variables thanks to *function inverses*. Yet, it rests on several orthogonal lines of work, of which here are a several.

¶ | PROGRAMMING WITH BINDERS Manipulating syntax with binders with the traditional algebraic data types introduced by Burstall et al. [1980], is notably hard; many "design patterns" have been proposed to squeeze their higher-order nature into purely first-order data structures [de Bruijn, 1972, McBride and McKinna, 2004, Aydemir et al., 2008]. Pouillard and Pottier [2010] designed a library which, thanks to the abstraction provided by a module system, helps to enforce invariants on the representation. Another approach is to enrich the language itself; for instance,

Shinwell et al. [2003] propose FreshML, an extension of ML based on nominal logic in which new constructs ensure the freshness of variable names. Similarly, Cαml [Pottier, 2006] compiles a *binding specification* to the corresponding first-order OCaml type. All these approaches are mainly concerned with forbidding to explicitly manipulate bound names as strings, namely to avoid variable capture. Because DLF uses the SLF apparatus as its values and relies on it for substitution, there is no risk of variable capture; however, nothing statically prevents the programmer from introducing free variables, but dynamic checks will then fail.

¶ FUNCTIONAL LANGUAGES WITH HIGHER-ORDER DATA The closest relatives to DLF are the functional languages that extend algebraic data types with values that can represent proofs. Specifically, it should be categorized in the family of so-called "*two-level*" specification languages. They distinguish syntactically two strati in programs: the *representational* part, which is the specification of types and constants, which can be higher-order, and the *computational* part itself, which consist of functions manipulating these data. Beluga [Pientka, 2008, Pientka and Dunfield, 2008, 2010], already been mentioned in Section 2.2, takes LF as its representational layer. It approaches the problem of manipulating open terms by *contextual* types, which is based on the theoretical work of Nanevski et al. [2008]: the type of an open term is parameterized by the environment that closes it. Contextual types are a generalization of the *necessity* modality [Pfenning and Davies, 2001]. The result is a statically typed programming language where functions are ensured to return a well-typed LF object. VeriML [Stampoulis and Shao, 2010, 2012] is a direct descendant of Beluga, that specializes in programming safely tactics for proof assistants; their main difference is that its logical layer is not a logical framework but a dependently typed programming language close to Coq. Delphin [Poswolsky and Schürmann, 2008] uses LF as its logical engine, but distinguishes two kinds of variables in its computational language: those subject to substitution and those intended to remain uninstantiated. It is based on the use of the ∇ type quantifier of Miller and Tiu [2005] that guarantees freshness of term variables.

We could maybe more adequately compare DLF with the way proof assistants are generally implemented. Coq, Matita, Agda, HOL and many others are implemented according to the so-called *De Bruijn principle* [Wiedijk, 2006]: the vast majority of their code is dedicated to help the user build proof objects (by refinement, tactics etc.), but ultimately all of them will be rechecked by a small, trusted part of the code, the *kernel* [Asperti et al., 2009]. DLF can be considered as a refinement of this idea in two ways: first, the production and the verification of proofs are intertwined, such that ill-typed proofs can be caught earlier; secondly, its simple computational language allows to hide most of the details concerning binder representation and present only a named interface. Of course, there remains to implement a full-blown

tactic-based proof assistant in DLF to judge if the approach scales up.

Of course, the principal difference with the systems mentioned above remains that DLF is dynamically typed: there is no static check enforcing that the result of a computation will be well-typed with respect to SLF, not even that function inverses are well-utilized: it is only a "programming style". Typing statically the computational language is a clear next step in our further work.

¶ | ENVIRONMENT-FREE PROGRAMMING Although the two levels are well-distinguished in DLF, our framework takes another approach to managing open data: we prefer to think that input terms are given closed by the user, and that they are open only in the recursive case; during the computation, we consider an open term as being *closed by a special construct representing all computations remaining to do on variables*; this is the essence of *function inverses*, a concept that is a generalization to any computation of how Geuvers et al. [2010] and Boespflug [2011] program type checkers without carrying an explicit environment. The reader familiar with *normalization by evaluation* [Berger and Schwichtenberg, 1991] will have recognized a known pattern. *NbE* is a full evaluation algorithm consisting of two passes: first, it projects the syntax of terms into a *model* (usually the function space of the meta-language); then it *reifies* back this meta-objects into the syntax of full values. The parallel with DLF is double. First, our full evaluation algorithm performs exactly these two steps, called here *resp.* weak evaluation and readback, even though we do not disinguish syntactically "syntactic" objects, objects "in the model" and values: they all share the same syntactic class of objects. This was already remarked by Grégoire and Leroy [2002]. Secondly, and more questionably, the use of function inverses recalls strongly the implementation of function *reify* of *NbE*: instead of descending "syntactically" in the body of a λ-abstraction, we use a built-in substitution (that of SLF in our case, that of the metalanguage in *NbE*) to substitute the variable by the result of the evaluation on variables (an inverse in our case, a function *reflect* for *NbE*, that has a type inverse of *reify*). This certain connection remains however to be explored beyond the remark of this mere resemblance.

¶ | HIGHER-ORDER LOGIC PROGRAMMING Although further away from our objective, we should mention the large body of work in logic programming languages manipulating sorted or typed higher-order data. There, computation is not the evaluation of a functional language, but proof search, starting from given axioms that constitute the program. λProlog [Miller and Nadathur, 1986, 2012] is an extension of Prolog that departs from first-order terms, and is based instead on an intuitionistic fragment of Church [1940]'s simple theory of types. Therefore, it manipulates a language of terms with λ-abstraction. One of its direct descendent is Twelf [Pfenning and Schürmann, 1999], which emerged as the canonical implementation

of LF. It features a logic programming engine that allows to see LF signatures as logic programs, and a totality checker that checks coverage and termination of these programs.

¶ | COMPUTING LF CERTIFICATES The practical purpose of DLF is to enhance the trust of a particular piece of (initially untrusted) software by letting it issue a *proof certificate*. A certificate is the syntactic representation of a mathematical proof that the output respects a certain specification. It can be communicated from the producer to a consumer, which can recheck its validity independently thanks to a small proof checker. This approach, known as *Proof-Carrying Code* (PCC) [Necula, 1997], was initially thought as a mean to execute untrusted code by designing certificate-issuing compilers and assembly code verification. It refers nowadays to any form of certificate-issuing computation [Miller, 2011]. Leroy [2006] calls this scheme *certifying*, as opposed to the *certified* scheme where the piece of software itself is proved correct, e.g., in a proof assistant, and can thus only issue correct code; he compares the two approaches and shows that they are observationally equivalent in terms of trust. Certifying software is more lightweight (formal verification can be avoided for inner parts of the software, they just have to issue a certificate) but more subject to poor implementation (the certificate eventually issued can be ill-typed).

The LF logical framework [Harper et al., 1993] is a common choice of representation for certificates, due to its universal nature: the choice of a particular logics is the matter of representing it as a signature. As studied by Appel and Felten [1999] the *trusted base* of a PCC system based on LF includes its type checker, the signature of the particular logic, and the formal description of the safety policy required by the consumer. A similar goal is sought by Dedukti [Boespflug, 2011], a proof checker for an extension of LF where types are seen equal *modulo* a set of user-provided rewriting rule: *Deduction modulo* [Dowek et al., 2003]. It differs however from DLF both by its goal (provide a compact universal evidence format by eliding computation steps from proofs) and by its realization (rewriting rules where we use functional programs, computation entirely in types where ours happen only in objects).

A common weakness found for proofs in LF are their sizes: they usually contain a lot of redundancy, and communicating them as-is does not usually scale to large proof objects. Several different variants have been proposed as the actual data structure to transmit, most of them compressing proofs using sharing [Appel and Felty, 1999, Stump and Dill, 2002, Appel et al., 2003] or reconstruction by unification [Necula and Lee, 1998, Reed, 2004, Sarkar et al., 2005]. A recent, novel choice proposed by Miller [2011] is to represent certificates using higher-order logic compressed by the use of *focusing* [Andreoli, 1992]. All these techniques of proof reduction have to make a compromise between the size of the proof (the amount of omitted information) and the size of the proof checker (the complexity of the code necessary

to rebuild the omitted information). Reducing the size of certificates produced by DLF will be a major challenge if the system is used in real-world condition; a side gain of the extension described in Chapter 3 is to implement a type safe sharing mechanism, that still remains to be tested and evaluated in real conditions.

¶ | DYNAMIC TYPING Finally, an important aspect of our framework is that typing is not checked statically, but at run-time. Type checking statically function code would guarantee that they *always* return SLF objects of the announced type, i.e., these functions would be *certified*. This would certainly impose a much stricter discipline on how to write their code.[18] Our proposition is more lightweight, at the cost of having to do run-time checks. In this sense, it is close to *dynamically typed* languages: the type information is conveyed at run time.

Practically, our OCaml implementation is a *shallow embedding* of DLF [Wildmoser and Nipkow, 2004]: it uses OCaml itself as the computational language. For OCaml, all SLF objects have the same type obj, but run-time checks are inserted at the right places (see Section 3.4). In that, it is related to *hybrid type systems* [Findler and Felleisen, 2002, Wadler and Findler, 2009]. Hybrid type systems separate two phases in the language: the typed one and the *unityped* one [Harper, 2012], where all types collapse to a single one, usually called Dyn. The comparison stops there since the two phases of DLF are not mirrors of each other: the representational level is not a programming language.

[18] For instance, on several occasions in Section 2.4, we used informally meta-arguments on the signature to justify our code; this would need to be formally verifiable if statically checked.

3 | INCREMENTAL TYPE CHECKING

¶ | ABSTRACT What is the impact of a small, confined modification on the correctness of a large program? Will it break the invariants and abstractions I carefully enforced by typing? This chapter presents a safe method for incrementally judging of the well-typing of a program, that is without having to batch recompile the whole program after each change. We begin this chapter by an analysis and critique of available tools to ease the management of changes in typed programming and proof languages. We then present an algorithm for incrementally type checking SLF objects based on sharing. It will be the basis of cDLF, a framework taking any certifying type checker written in Chapter 2's DLF programming language and turning it into an incremental type checker. We finish by presenting some highlights of its implementation as an OCaml library.

3.1 | MOTIVATIONS

The interaction between a user and a compiler has not changed much since the early days of compiled programming languages: a program is written in a text file thanks to a (more-or-less) specialized text editor. When he is done, the user calls the compiler which parses, checks and compiles the program. In case of error in any of these passes, the compiler reports it, possibly with its location in the source file. If there is none, the user is then free to launch the produced executable to test it for possible bugs. Yet, the huge size and complexity of the programs we develop today is such that a developer spends more time *editing* its already-written program to correct bugs or add a feature than actually *writing* completely new code, and our compilation toolchain is not adapted to this inherently non-linear workflow.

Richly typed programming languages offer the advantage of giving programs a strict, *static semantics*, that helps greatly to detect bugs statically, i.e., without the need for testing. In this context, the *interaction* with the compiler, more specifically with the type checker, becomes the central part of the development process. As type systems become richer, it becomes increasingly difficult for the programmer to write a well-typed program in one try, compile it and run it: writing a program becomes

more and more a tight and non-linear interaction between the human and the type checker, involving constant experiments, fixes, etc.

We believe this to be true for strongly typed programming languages, in particular functional languages exploiting higher-order functions and algebraic data types like OCaml. There, a programmer will often even *rely* on the type checker to indicate conflicting uses after a small change: for instance, he would add a constructor to an existing data type, and launch compilation again for the type checker to detect non-exhaustive pattern-matching on this type, and add a new case to each of them. We believe this to be *a fortiori* true for proof languages, where the construction of a valid proof is simply not conceivable without guidance from the system: most proof assistants provide editing environments which show e.g., the goal to be solved at a certain point.

¶ INCREMENTAL TYPE CHECKING If a batch interaction, where we feed the whole program to the type checker at every run, is possible for short programs and fast checkers, it becomes increasingly hindered by the latency of complete rechecks when programs get longer and the time taken to check then becomes non-negligible: even if the modifications made to the program between two interactions are small, the whole program is rechecked. In particular for languages with formal verification aspects like proof assistants, the time taken to infer easy parts of the proofs (proof search) not present in the source script can often be long. What if we could compute *incrementally* the effect of the edition of the source by numerous small changes on the well-typing of whole edifice (a program of thousands lines of code, a complex proof requiring hundreds of lemmas)?

The following examples should illustrate the benefits of such an interaction, and also the difficulties it poses:

Example 3.1. For instance, we could want to change a small subterm M in a large, well-typed OCaml program $C[M]$ by e.g., inserting the call to a function `f` resulting in $C[\texttt{f}\ M]$. It suffices to know that M has type A—an information available from the type checking of the original program—and `f` has type $A \to A$ in context C to conclude that $C[\texttt{f}\ M]$ is well-typed: there is no need to recheck it entirely. We could want to change completely the definition of a symbol, from the well-typed $C[\textbf{let}\ \texttt{x} = M\ \textbf{in}\ N]$ to $C[\textbf{let}\ \texttt{x} = M'\ \textbf{in}\ N]$. If M and M' have the same type, then the new term is well-typed without the need to recheck it entirely; if their types are different, it depends only on the use sites of `x` in N, except if this changes the type of N, in which case we need to analyze its impact on context C...

On a higher level, we could add a constructor to a data type, and be instantly pointed by the system at all problematic pattern matches, possibly very far away from the type's declaration, *without having to wait for the recompilation of the whole program*. In the context of formal proof, we could realize that one of the hypotheses

of a lemma is actually not needed in its proof, and suppress it from its statement; we would be pointed by the system to all uses of this lemma in the whole development needing to be changed, or if there is none, be instantly informed that the whole development is still valid without needing to recheck it completely.

Incremental type checking thus encompasses the verification of well-typing of a large program (or proof following the *Curry-Howard isomorphism*) when performing arbitrary small modifications, and this in less time than rechecking the whole program. An informal definition of the problem could be:

Definition 3.1 (Incremental type checking)**.** Suppose a typed language of terms M, and a representation of changes, or *deltas* δ, on M such that a function $apply$ takes a term and a delta into another term. Given a well-typed term M and a term delta δ, decide whether $apply(M, \delta)$ is well-typed in a time less than $O(|apply(M, \delta)|)$, where $|M|$ is the number of nodes in term M.

There remains to devise a particular type system, a notion of deltas, and the incremental type checking algorithm. Note that in many interesting cases we cannot hope to achieve incremental type checking in time $O(|\delta|)$ if the notion of delta chosen follows the syntax of the language: as we saw on the previous examples, the impact of a change on the type of a program can reside far outside the location of the change itself.

D | INCREMENTALITY IN PROGRAMMING AND PROVING

This remark on the usefulness of incrementality is not new. Actually, several tools have become standard nowadays to cope with the fact that compilers process source in batch.

¶ | SEPARATE COMPILATION AND DEPENDENCY MANAGEMENT The most important of these tools is separate compilation. A large development can be split into several files or *modules*, and compiled separately (we are here interested only in the type checking aspect of compilation); Each file comes with an *interface*, either given or inferred, setting out the types of all values defined in the module. To check the type of a module supposing that all its dependencies have been checked already, we just look up in the interfaces the type of all external values contained in that module; there is no need to recheck their implementation. Given a module dependency graph, tools like make automatically recheck (recompile) a changed module and all its reverse dependencies.

Separate compilation is a manual and coarse way of handling incrementality. The programmer has to manually split its development into modules, making sure by himself that the dependency relation is not circular. Besides, the atomic unit

of compilation is the module, which can be arbitrarily large; small changes within a module have to be handled in a batch fashion. Moreover, as remarked already by Cardelli [1997], modules usually serve another purpose: that of organizing the development into logical units and namespaces; yet, there is no reason for these two purposes to coincide.

¶ TOPLEVEL WITH GLOBAL ROLLBACK Incidentally, in the context of proof assistants, separate compilation is not fine-grained enough of a tool, and the standard approach is to split modules further into *commands*. Coq [Coq development team, 2012] or Matita [Asperti et al., 2007] are simple *toplevels*[1] (or *read-eval-print loops*) where one enters commands that modify the internal state of the prover and print information about this state. On top of this mechanism, a global undo mechanism is provided to roll back from erroneous commands. Their user interfaces, e.g., Aspinall [2000], follow this interaction model (Figure 3.1). A read-only, colored zone delimited by the *cursor* indicates the commands that are taken into account by the system and the user can move the cursor by commands forward or backward; the zone after the cursor is editable. This provides a form of incremental proof checking: one can change a command after the cursor without having to recheck all commands before the cursor.

It is nonetheless very limited: there is no way to alter a command without having to recheck all commands after it, even if they are not affected by the change at all. For instance, if we modify the proof of a lemma (but not its statement), the theorems depending on it will need to be rechecked. Note that there is a significant ongoing effort to reform this model and implement parallel processing in proof assistants [Wenzel, 2012], with possible gains on incrementality of checking.

¶ BOTTOM-UP PROOF CONSTRUCTION The third, and maybe more complex tool for incremental construction of proofs and programs is implemented in proof assistants of the LCF family [Gordon et al., 1979] (of which Coq and Matita are members along with many others), and is also adopted in a simpler form in Agda. The idea is to construct proof terms sequentially from the root to the leaves, at each step leaving "holes", or *metavariables*, in the proof to be filled up later. For instance, to prove the *goal* A, the user might provide a first term $C[\mathtt{X},\mathtt{Y}]$ with two holes $\mathtt{X}$ and $\mathtt{Y}$; he might then provide a second term $C'[\mathtt{Z}]$ to fill up $\mathtt{X}$, resulting in the term $C[C'[\mathtt{Z}],\mathtt{Y}]$ with two goals etc. until there is no more holes in the proof. The gain is twofold: at each step, the system can print the goals remaining to be solved (the type of each hole), providing helpful feedback to the user. Secondly, this provides a limited form of incrementality: $C[\mathtt{X},\mathtt{Y}]$ is checked to be of type A, and $\mathtt{X}$ is annotated to be of type B; then $C'[\mathtt{Z}]$ is checked to be of type B. If it is the case, we have the

[1] Actually, a system of separate compilation is also provided in these tools.

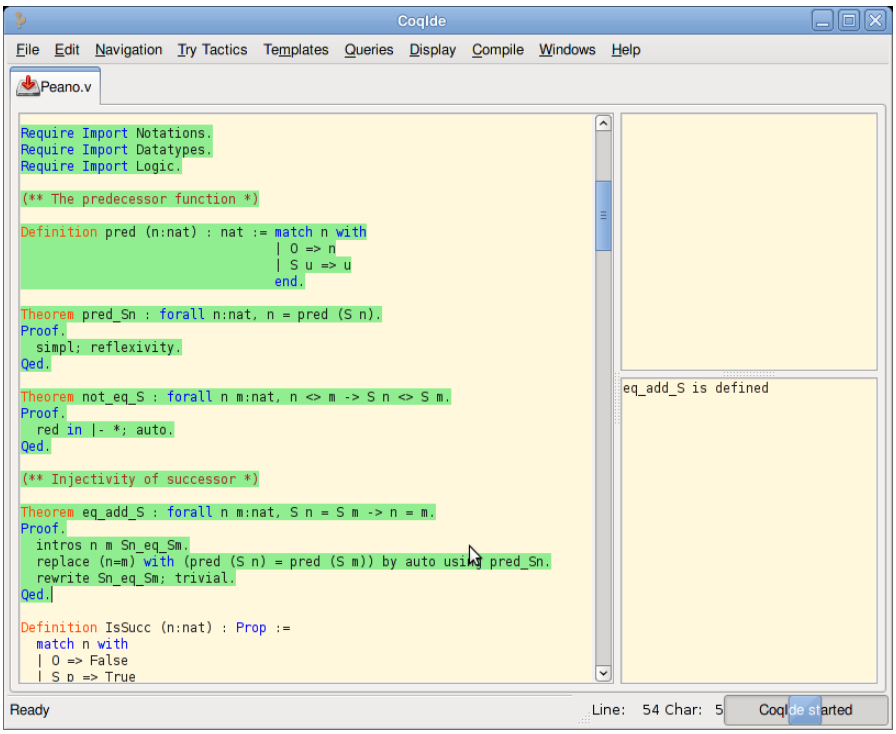

Figure 3.1: The linear interaction loop in a proof assistant: Coqide

guarantee that the whole reconstructed proof will be of type A without ever having to recheck it.[2] This exact facility is provided in Agda [Norell, 2007]: the user can put holes in its terms to query the type of the expression expected at that point, and later fill them up without the system rechecking the whole file. In LCF and its descendants, proofs are constructed bottom-up by the repeated application of *tactics*, which are (untrusted) programs generating terms with holes.

But this system is limited in the kind of modifications we can incrementally make on terms: for instance in Agda's user interface, we can fill up a hole with a term, but the modification of any term that is not a hole requires to recheck the entire file. Similarly in e.g., Coq, no tactic can modify parts of an already-built proof: they can only fill in holes.

This is why we believe that all these tools solve only instances of the more general problem of incremental type checking. The goal of the rest of this chapter is to show how the context developed in Chapter 2 can be adapted to help solve this problem.

E | INCREMENTALITY THROUGH SHARING OF SUBDERIVATIONS

Consider again definition 3.1. To save us from recomputing the type information for already-typed terms, we somehow need to store the type information between runs of the type checker. This way, if some annotated subterm changes, typing can resume at that position, without needing to check unchanged subterms. This is what separate compilation does when generating (or checking) the *interface* of a module, or bottom-up proof construction systems when computing and storing the awaited type of a hole. In all generality, this type information must be stored *for all subterms* of the program, not only at special nodes, like module definition or holes. This way, *any* subterm can be changed and rechecked incrementally.

We propose to actually store the *typing derivation* of the program, in hypothetical notation. Since typing derivations contain explicitly the type information at each node of the abstract syntax tree,[3] it will allow to reuse any subderivations for any unchanged parts of the program.

Example 3.2. Suppose that we computed the typing derivation of the simply-typed λ-term $(\lambda f\, x.f\, x)\ (\lambda x.\, \mathbf{s}(x))\ (\mathbf{s}(\mathbf{o}))$:

[2] We will see in the rest of this chapter that the situation is actually more complicated since these holes can appear under an environment of hypotheses.

[3] We should add: *at least* at every node, since there can be silent typing rules, for instance a subtyping rule.

$$\dfrac{\dfrac{\dfrac{[\vdash f : \mathsf{nat} \to \mathsf{nat}] \quad [\vdash x : \mathsf{nat}]}{\vdash f\ x : \mathsf{nat}}}{\vdash \lambda x.f\ x : \mathsf{nat} \to \mathsf{nat}}}{\vdash \lambda f\,x.f\ x : (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat} \to \mathsf{nat}} \quad \dfrac{\dfrac{[\vdash x : \mathsf{nat}]}{\vdash \mathbf{s}(x) : \mathsf{nat}}}{\vdash \lambda x.\,\mathbf{s}(x) : \mathsf{nat} \to \mathsf{nat}} \quad \dfrac{\dfrac{}{\vdash \mathbf{o} : \mathsf{nat}}}{\vdash \mathbf{s}(\mathbf{o}) : \mathsf{nat}}}{\dfrac{\vdash (\lambda f\,x.f\ x)\ (\lambda x.\,\mathbf{s}(x)) : \mathsf{nat}}{\vdash (\lambda f\,x.f\ x)\ (\lambda x.\,\mathbf{s}(x))\ (\mathbf{s}(\mathbf{o})) : \mathsf{nat}}}$$

and that we are given the new term to type check $(\lambda f.\,\lambda x.f\ x)\ (\lambda x.\,\mathbf{s}(\mathbf{s}(x)))\ (\mathbf{s}(\mathbf{o}))$. Only the body of the first argument is changed, so there is no need to recompute the typing derivation $\mathscr{D}$ of the functional $\lambda f\,x.f\ x$, neither the second argument $\mathbf{s}(\mathbf{o})$ (let us call it $\mathscr{D}_2$); subterm $\mathbf{s}(x)$'s derivation $\mathscr{D}_1$ did not change either. We can reuse them by building only the partial derivation:

$$\dfrac{\dfrac{\mathscr{D}}{\vdash \lambda f\,x.f\ x : (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat} \to \mathsf{nat}} \quad \dfrac{\dfrac{\dfrac{[\vdash x : \mathsf{nat}]}{\mathscr{D}_1}}{\vdash \mathbf{s}(x) : \mathsf{nat}}}{\vdash \mathbf{s}(\mathbf{s}(x)) : \mathsf{nat}}}{\vdash \lambda x.\,\mathbf{s}(\mathbf{s}(x)) : \mathsf{nat} \to \mathsf{nat}} \quad \dfrac{\mathscr{D}_2}{\vdash \mathbf{s}(\mathbf{o}) : \mathsf{nat}}}{\dfrac{\vdash (\lambda f\,x.f\ x)\ (\lambda x.\,\mathbf{s}(\mathbf{s}(x))) : \mathsf{nat}}{\vdash (\lambda f\,x.f\ x)\ (\lambda x.\,\mathbf{s}(\mathbf{s}(x)))\ (\mathbf{s}(\mathbf{o})) : \mathsf{nat}}}$$

Checking this second term amounts to only generate the nodes of this second derivation: all unchanged subderivations are shared, and not recomputed. Note that the three nodes at the bottom (the two applications and the lambda) are regenerated, even though they are identical to their equivalent in the original derivation. Note also that thanks to the hypothetical notation of proofs, we reuse derivation $\mathscr{D}_1$, even though it depends on hypothesis $\vdash x : \mathsf{nat}$.

¶ | MEMOIZATION Reusing the result of a sub-computation when its input is equal to a previous run is a well-known technique in functional programming called *memoization* [Michie, 1968, Norvig, 1991, Acar et al., 2003]. For each successful call to a function f, it amounts to record in a global map the pair of its input and output. When the function is called, a lookup in the table is made with the arguments of the function: if a binding is found, we return the previously computed result; if not, we actually compute the result and store it in the table. This supposes that the arguments to function f are first-order, comparable data. We could simply think of the previous process of derivation sharing as memoizing a type checker:

$$\mathit{check} \ : \ \mathsf{env} \to \mathsf{term} \to \mathsf{type}$$

The approach of explicitly computing the derivation presents two main differences. First, memoization can be optimized in language-dependent ways: for instance, the comparison of argument during the lookup can be made up to "compatible" environments and terms (weakening, strengthening, substitution, type instantiation...) that are hard to justify metatheoretically. Generating the derivation allows to check it afterwards, thus providing a *justification* for all changes. Secondly, reusing *higher-order* derivations allows for more "intelligent" reuse of results than purely first-order memoization, as witnessed in this example:

Example 3.3. Suppose now that we are given the term $(\lambda f\, x.f\ x)\ (\lambda x.\,\mathbf{s}(\mathbf{s}(\mathbf{o})))\ (\mathbf{s}(\mathbf{o}))$ to type check, knowing the first derivation of example 3.2. We can recognize this term as the original term where we replaced x in the first argument by $\mathbf{s}(\mathbf{o})$, a term for which we already have a derivation $\mathscr{D}_2$. We can generate the following derivation:

$$\dfrac{\dfrac{\vdash \lambda f\, x.f\ x : (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat} \to \mathsf{nat}\ (\mathscr{D}) \qquad \dfrac{\dfrac{\vdash \mathbf{s}(\mathbf{o}) : \mathsf{nat}\ (\mathscr{D}_2)}{\vdash \mathbf{s}(\mathbf{s}(\mathbf{o})) : \mathsf{nat}}\ (\mathscr{D}_1)}{\vdash \lambda x.\,\mathbf{s}(\mathbf{s}(\mathbf{o})) : \mathsf{nat} \to \mathsf{nat}}}{\vdash (\lambda f\, x.f\ x)\ (\lambda x.\,\mathbf{s}(\mathbf{s}(\mathbf{o}))) : \mathsf{nat}} \qquad \vdash \mathbf{s}(\mathbf{o}) : \mathsf{nat}\ (\mathscr{D}_2)}{\vdash (\lambda f\, x.f\ x)\ (\lambda x.\,\mathbf{s}(\mathbf{s}(\mathbf{o})))\ (\mathbf{s}(\mathbf{o})) : \mathsf{nat}}$$

To type $\mathbf{s}(\mathbf{s}(\mathbf{o}))$, we first use $\mathscr{D}_1$, but instead of discharging its hypothesis $\vdash x : \mathsf{nat}$, we instantiate it by derivation $\mathscr{D}_2$. The amount of recheck to do is minimal here: no new successor node is needed. Memoization would here fail to achieve as much sharing, because it would only detect already-seen subterms. Specifically, it would generate a successor node for the outer $\mathbf{s}()$ of $\mathbf{s}(\mathbf{s}(\mathbf{o}))$.

Our key to incremental type checking is therefore having a type checker return a typing derivation in hypothetical proof notation in a form where we can address and reuse any subderivation. By grafting already-checked subderivation to a derivation being built, we avoid type checking again subterms that were already type checked.

¶ | CERTIFICATES FOR INCREMENTAL TYPE CHECKING How should we represent and compute these typing derivations? It turns out that we can reuse the apparatus constructed in the two previous chapters: SLF (Section 1.2) as a higher-order data structure for proofs that can favorably represent typing derivations, and DLF (Section 2.3) as a certifying computation language on them. In Section 2.4.2, we gave a practical tutorial on how to program a type checker in this framework. To implement our notion of incremental type checking, we are lacking only a way to

store pre-generated derivations, and to refer to and instantiate pieces of derivations marked as already checked.

In this chapter, we develop gradually a version of DLF that generates large derivations, *slices* them into small, reusable pieces, and allows the user to refer to these slices. For this we introduce successively systems cSLF (Section 3.2), a *contextual* logical framework that constructs a context of derivation slices, and cDLF (Section 3.3), a computation language close to DLF, but relying on cSLF instead of SLF as a representation language. This language will allow a user to enter successively program *deltas*, which derivations will be built and checked against a *context* of already-constructed pieces of derivations. This will be demonstrated in Section 3.3.3. Finally, in Section 3.4 we present an overview and some of the important ideas of our implementation of cDLF, an OCaml library named Gasp.

3.2 | cSLF: SHARING AND REUSING SLF PROOFS

In our quest to incrementally check programs, we are first interested in verifying the validity of typing derivations *piece by piece*. In this section, we construct a language and algorithm cSLF that will allow to answer the question: given a set of already-checked derivation pieces (we will call them *slices*), is the new slice $\mathscr{D}$ valid? Not only will it answer it, but in case of success, it will add to the set of checked slices all newly checked slices.

3.2.1 | PRESENTATION

Let us first introduce informally the different features that allow us to extend SLF to form the contextual type theory of cSLF.

¶ | LOCAL VERIFICATION What is a *slice* of derivation? And can it be checked independently of the context in which it appears? Two preliminary remarks will help to answer these questions. We noted from the previous section that we need to be able to address any subderivation or syntactical construct, that is any rule application. Now, a constant spine $\mathsf{c}(S)$ in SLF is the encoding of the application of a *rule*, be it syntactical or logical.

Example 3.4. Let M and N be the encodings of two λ-terms t and u, i.e., objects of type tm in the signature of example 2.1. Then the constant spine $\mathsf{app}(M,N)$ is the encoding of λ-term "$t\ u$". Let $\mathscr{D}_1$ and $\mathscr{D}_2$ be the encoding of proofs of *resp.* $\vdash \mathsf{p} \supset \mathsf{q}$ and $\vdash \mathsf{p}$, i.e., objects of type $\mathsf{is}(\mathsf{p})$ and $\mathsf{is}(\mathsf{imp}(\mathsf{p},\mathsf{q}))$ in the signature of NJ proofs

(Figure 1.1). Then constant spine $\mathsf{ImpE}(\mathsf{p},\mathsf{q},\mathscr{D}_1,\mathscr{D}_2)$ is the encoding of proof:

$$\dfrac{\begin{array}{cc}\mathscr{D}_1 & \mathscr{D}_2\\ \vdash \mathsf{p}\supset\mathsf{q} & \vdash \mathsf{p}\end{array}}{\vdash \mathsf{q}}$$

Our language must thus give a *name* to all applications of constants, and the ability to refer to these names.

Secondly, SLF is a *local encoding* of proofs: the encoding of a proof is a term that is annotated enough to decide *locally* if two consecutive rule applications agree on the judgment at their interface. By locally, we mean without having to look at the subproofs of these two rules; by agreement, we mean that if the subproofs are valid, then the whole proof is valid. This is due to the fact that each logical rule application explicitly mentions the *substitution* by which it is instantiated, as introduced in Section 1.1.1.

Example 3.5. For any objects M_1 and M_2, if object $M = \mathsf{ImpE}(\mathsf{p},\mathsf{q},M_1,M_2)$ is the SLF encoding of a valid proof (this is all we know), and M_0 is the encoding of a valid proof of $\vdash \mathsf{q}$, then we know that $\mathsf{ImpE}(\mathsf{q},\mathsf{r},M_0,M)$ is a valid proof of $\vdash \mathsf{r}$, and contrarily that $\mathsf{ImpE}(\mathsf{q},\mathsf{r},M,M_0)$ is not.

This is in opposition to a *global encoding*, where checking the validity of two consecutive rules requires to know the encoding of the whole proof.

Example 3.6. Consider the encoding of NJ proofs by proof terms (rule ImpE is encoded by an application, ImpI by a λ-abstraction etc.). For any proof term t_1 and t_2, if proof term $t = t_1\ t_2$ is well-typed (it is all we know) and t_0 is the encoding of a valid proof of $\vdash \mathsf{r}$, what to say of term $t_0\ (t_1\ t_2)$? To know if it is valid, we must compute the types of t_1 and t_2, which implies to know t_1 and t_2.

In other words, the SLF encoding of a subproof contains enough information to check it once and for all, and later graft it in another proof without having to inspect it again. Whether the grafted proof is valid then amounts only to a *local* check, at the boundary of the graft. This is exactly what cSLF will do.

¶ CONTEXT AND METAVARIABLE To refer to already checked constant spines, we introduce in objects a new set of special variables, called *metavariables* and noted $\mathtt{X}$, $\mathtt{Y}$, $\mathtt{Z}\ldots$ Each metavariable stands for a defined atomic object, which definition can be found in a large store, the *context* Δ. We call each element of this context a *slice*, and you can think of it as a named, already checked object, itself possibly referring to other metavariables defined somewhere else in this context.

Because SLF objects are higher-order terms, possibly containing λ-abstractions, a slice named $\mathtt{X}$ can potentially contain free variables. When we decide to reuse $\mathtt{X}$ when

building another object, we will need to ensure that all free variables of that slice are either bound or defined. This is why every slice is parameterized by an *environment* of variables that are free in the slice, and all occurrences of a metavariable comes with a *substitution* $\mathtt{X}[\sigma]$ that instantiates these variables. Then, we will need to compare the *awaited type* at this point with the *actual type* of the metavariable; when a slice has been checked and its type computed, we put it in the current context *with its actual synthesized type*. This annotation behaves like a *cache*: it could be reconstructed, but remembering it will allow to quickly compare the types when grafting a metavariable. A context binding will thus be written $\mathtt{X}[\Gamma] : P = \mathsf{c}(S)$.

Example 3.7. The object $\mathsf{lam}(\lambda x.\, \mathtt{X}[x/x])$ in a context where $\mathtt{X}[x : \mathsf{tm}] : \mathsf{tm} = \mathsf{lam}(\lambda y.\, \mathtt{Y}[x/x, y/\mathsf{app}(x,y)])$ and $\mathtt{Y}[x : \mathsf{tm}, y : \mathsf{tm}] : \mathsf{tm} = \mathsf{app}(x,y)$ is a sliced representation of the well-typed object $\mathsf{lam}(\lambda x.\, \mathsf{lam}(\lambda y.\, \mathsf{app}(x, \mathsf{app}(x,y))))$.

Note that Γ binds here not only in S but also in A: this type can contain objects with free variables; besides, A is also free to refer to metavariables defined elsewhere in the context. An invariant that we *must* preserve is for a valid context Δ to be *acyclic*, i.e., a metavariable must refer only to metavariables not referring to itself.

The introduction of metavariables is a common tool when implementing programs that manipulate higher-order syntax: in Coq and other LCF-style proof assistants, they are used as the holes to implement bottom-up proof search, but also unification and disambiguation [Barras, 1999, Pfenning, 2001, Spiwack, 2010]. Although earlier propositions [Dowek et al., 2000], were based on the use of (usual) variables of function types to denote open objects, it is now standard to distinguish between variables and metavariables. Nanevski et al. [2008] expose a type theory with metavariables called *Contextual Modal Type Theory* (CMTT) that denote holes in an object. Our language can be thought as a restriction of their work: contrarily to CMTT, all of our metavariables actually have a definition, and can be expanded.

¶ | TYPING AND SLICING A context is thus a set of well-typed objects, and we can already imagine that it is easy to check the well-typedness of a new object referring to metavariables in this context. But how do we construct such a context?

At first sight, it might seem easy to design an algorithm traversing a well-typed object M and returning the pair of a context Δ and an object M' where M' is the "sliced" version of M, which metavariables are defined in Δ. However, to record an object as a slice, we need to know its type, and the types of each of its free variables, so this type information has to be propagated throughout the traversal. All in all, this algorithm will do type checking as well: both the *typing* phase and the *slicing* phase will be interleaved. We present a simple algorithm, whose specification is:

$$\mathit{slice} : \mathsf{ctx} \times \mathsf{env} \times \mathsf{atom} \to \mathsf{option}\,(\mathsf{ctx} \times \mathsf{atom} \times \mathsf{type})$$

such that $\mathit{slice}(\Delta, \Gamma, F) = \mathsf{Some}(\Delta', F', P)$ only when F was well typed in Γ, with metavariables defined in Δ. Its post-condition is that F' is well-typed of type P in Γ, with its metavariables defined in Δ', and that F' "represents" F with all its constant spines sliced in Δ' (F' is *sliced*).

3.2.2 | DEFINITION

We now extend the SLF representation language to include *metavariables*, giving rise to cSLF, a subset of Nanevski et al. [2008]'s CMTT to represent proofs that are cut up in small, open object *slices*. We present its bidirectional typing algorithm. It takes place in a *context* Δ of slices, and one can refer to these in objects; the typing algorithm not only checks or infers the type of the object, but also returns the pair of a new context Δ', enlarged with the new slices resulting from slicing the object in question, and the sliced object in question.

A | SYNTAX AND SUBSTITUTION

The syntax of cSLF is presented on Figure 3.2. We suppose an infinite set of *metavariables* $\mathtt{X}$, $\mathtt{Y}$, $\mathtt{Z}\ldots$ This grammar differs from that of SLF (Figure 1.2) by only two additions: atomic objects are extended with metavariables attached to a substitution $\mathtt{X}[\sigma]$; also, we introduce *contexts* which are lists of metavariable definitions. You can think of each metavariable definition $\mathtt{X}[\Gamma] : P = \mathsf{c}(S)$ in a context Δ as the definition of an object $\mathsf{c}(S)$ that has already been checked to be of type P in environment Γ. In context Δ, it can then be referred to by other objects with the $\mathtt{X}[\sigma]$ construct, where substitution σ should instantiate all variables of Γ. In this sense, cSLF objects are still canonical with respect to β-reduction, as in SLF, but are not canonical with respect to unfolding of metavariables. Any cSLF term without metavariable is also an SLF term; in the following, we implicitly use this coercion.

Substituting cSLF objects is defined by extending SLF's substitution (Figure 1.3), and is showed on Figure 3.3. Only three new equations are added: when substituting σ' into a metavariable $\mathtt{X}[\sigma]$ (Equation (3.5)), we apply σ' to σ. This amounts to apply σ' to each binding x/M of σ (Equations (3.13) and (3.14)).

B | TYPED SLICING

The typing and slicing algorithm of cSLF is presented on Figures 3.4 to 3.6. We follow a syntactical convention to indicate the mode of its judgments: they are either of the form $X \vdash Y \Rightarrow Z$ or $X \vdash Y$ where X and Y are inputs, and Z, if present, is an output.

There are nine different judgments, corresponding to the nine judgments of SLF,

$$
\begin{array}{rcll}
K & ::= & \Pi x : A.\, K \mid * & \text{Kind} \\
A, B & ::= & \Pi x : A.\, B \mid P & \text{Type family} \\
P & ::= & \mathsf{a}(S) & \text{Atomic type} \\
\\
M, N & ::= & \lambda x.\, M \mid F & \text{Canonical object} \\
F & ::= & H(S) \mid \mathtt{X}[\sigma] & \text{Atomic object} \\
H & ::= & x \mid \mathsf{c} & \text{Head} \\
S & ::= & \cdot \mid M, S & \text{Spine} \\
\\
\sigma & ::= & \cdot \mid \sigma, x/M & \text{Parallel Substitution} \\
\Gamma & ::= & \cdot \mid \Gamma, x : A & \text{Environment} \\
\Sigma & ::= & \cdot \mid \Sigma, \mathsf{c} : A \mid \Sigma, \mathsf{a} : K & \text{Signature} \\
\\
\Delta & ::= & \cdot \mid \Delta, \mathtt{X}[\Gamma] : P = \mathsf{c}(S) & \text{Contexts}
\end{array}
$$

Figure 3.2: Syntax of cSLF: contextual SLF

$$
\begin{array}{rll}
(\lambda x.\, M)[\sigma] = \lambda x.\, M[\sigma] & \text{if } x \notin \mathsf{FV}(\sigma) \text{ and } x \notin \mathrm{dom}(\sigma) & (3.1) \\
(\mathsf{c}(S))[\sigma] = \mathsf{c}(S[\sigma]) & & (3.2) \\
(x(S))[\sigma] = x(S[\sigma]) & \text{if } x \notin \mathsf{FV}(\sigma) \text{ and } x \notin \mathrm{dom}(\sigma) & (3.3) \\
(x(S))[\sigma] = M \star S[\sigma] & \text{if } x/M \in \sigma & (3.4) \\
(\mathtt{X}[\sigma])[\sigma'] = \mathtt{X}[\sigma[\sigma']] & & (3.5) \\
\cdot[\sigma] = \cdot & & (3.6) \\
(M, S)[\sigma] = M[\sigma], S[\sigma] & & (3.7) \\
*[\sigma] = * & & (3.8) \\
(\Pi x : A.\, B)[\sigma] = \Pi x : A[\sigma].\, B[\sigma] & \text{if } x \notin \mathsf{FV}(\sigma) \text{ and } x \notin \mathrm{dom}(\sigma) & (3.9) \\
(\mathsf{a}(S))[\sigma] = \mathsf{a}(S[\sigma]) & & (3.10) \\
\lambda x.\, M \star N, S = M[x/N] \star S & & (3.11) \\
F \star \cdot = F & & (3.12) \\
\cdot[\sigma] = \cdot & & (3.13) \\
(\sigma', x/M)[\sigma] = (\sigma'[\sigma]), x/(M[\sigma]) & \text{if } x \notin \mathsf{FV}(\sigma) \text{ and } x \notin \mathrm{dom}(\sigma) & (3.14)
\end{array}
$$

Figure 3.3: Hereditary substitution of cSLF

$\boxed{\Delta;\Gamma \vdash M : A \Rightarrow \Delta';M'}$ Canonical object

$$\textsc{MLam}\ \frac{\Delta;\Gamma,x:A \vdash M : B \Rightarrow \Delta';M'}{\Delta;\Gamma \vdash \lambda x.M : \Pi x:A.B \Rightarrow \Delta';\lambda x.M'}$$

$$\textsc{MAtom}\ \frac{\Delta;\Gamma \vdash F \Rightarrow \Delta';F' : P' \qquad \downarrow_{\Delta} P = \downarrow_{\Delta'} P'}{\Delta;\Gamma \vdash F : P \Rightarrow \Delta';F'}$$

$\boxed{\Delta;\Gamma \vdash F \Rightarrow \Delta';F' : P}$ Atomic object

$$\textsc{FVar}\ \frac{x:A \in \Gamma \qquad \Delta;\Gamma;A \vdash S \Rightarrow \Delta';S' : P}{\Delta;\Gamma \vdash x(S) \Rightarrow \Delta';x(S') : P}$$

$$\textsc{FConst}\ \frac{\mathsf{c}:A \in \Sigma \qquad \Delta;\Gamma;A \vdash S \Rightarrow \Delta';S' : P \qquad \mathrm{stren}_{\Gamma}(\mathsf{c}(S')) = \Gamma' \qquad \mathtt{X} \notin \mathrm{dom}(\Delta')}{\Delta;\Gamma \vdash \mathsf{c}(S) \Rightarrow (\Delta',\mathtt{X}[\Gamma'] : P = \mathsf{c}(S'));\mathtt{X}[\mathrm{id}_{\Gamma'}] : P}$$

$$\textsc{FMeta}\ \frac{\mathtt{X}[\Gamma'] : P = \mathsf{c}(S) \in \Delta \qquad \Delta;\Gamma \vdash \sigma : \Gamma' \Rightarrow \Delta';\sigma'}{\Delta;\Gamma \vdash \mathtt{X}[\sigma] \Rightarrow \Delta';\mathtt{X}[\sigma'] : P[\sigma']}$$

$\boxed{\Delta;\Gamma;A \vdash S \Rightarrow \Delta';S' : P}$ Object spine

$$\textsc{SCons}\ \frac{\Delta;\Gamma \vdash M : A \Rightarrow \Delta_1;M' \qquad \Delta_1;\Gamma;B[x/M'] \vdash S \Rightarrow \Delta_2;S' : P}{\Delta;\Gamma;\Pi x:A.B \vdash M,S \Rightarrow \Delta_2;(M',S') : P}$$

$$\textsc{SNil}\ \frac{}{\Delta;\Gamma;P \vdash \cdot \Rightarrow \Delta;\cdot : P}$$

$\boxed{\Delta;\Gamma \vdash \sigma : \Gamma' \Rightarrow \Delta';\sigma'}$ Substitution

$$\sigma\textsc{Cons}\ \frac{\Delta;\Gamma \vdash \sigma : \Gamma' \Rightarrow \Delta_1;\sigma' \qquad \Delta_1;\Gamma \vdash M : A[\sigma'] \Rightarrow \Delta_2;M'}{\Delta;\Gamma \vdash (\sigma,x/M) : (\Gamma',x:A) \Rightarrow \Delta_2;(\sigma',x/M')} \qquad \sigma\textsc{Nil}\ \frac{}{\Delta;\Gamma \vdash \cdot : \cdot \Rightarrow \Delta;\cdot}$$

Figure 3.4: Bidirectional typing algorithm for cSLF (1/3)

plus an additional one for contexts. The main judgment is $\Delta;\Gamma \vdash F \Rightarrow \Delta';F' : P'$ and reads: "in the context of slices Δ and environment Γ, slicing the well-typed atomic object F (possibly with deep applications of constants) results in the atomic object F', which is well-typed of type P' in the extended context Δ'". We call F' the *residual* of slicing: it represents the "tip" of F that has not been sliced. Binding in this judgment is as follows: types in Γ have metavariables defined in Δ, object F has its free variables bound in Γ and its metavariables bound in Δ, F' and P' have their free variables bound in Γ and their metavariables bound in Δ'.

Example 3.8. The residual F' of the slicing of object $F = x(\mathsf{c}(y))$ in environment $\Gamma = x : \mathsf{a} \to \mathsf{b}, y : \mathsf{b}$ is $x(\mathtt{X}[y/y])$ in context $\mathtt{X}[y : \mathsf{b}] : \mathsf{a} = \mathsf{c}(y)$ if $\mathsf{c} : \mathsf{b} \to \mathsf{a} \in \Sigma$, or:

$$\cdot;(x : \mathsf{a} \to \mathsf{b}, y : \mathsf{b}) \vdash x(\mathsf{c}(y)) \Rightarrow (\mathtt{X}[y : \mathsf{b}] : \mathsf{a} = \mathsf{c}(y)); x(\mathtt{X}[y/y]) : \mathsf{b}$$

As is customary, all judgments except the one for signatures are implicitly parameterized by a signature Σ that is constant in all rules, and that declares the types and kinds of constants c and a. The difference with SLF signatures is that cSLF signatures themselves can contain metavariables $\mathtt{X}[\sigma]$ standing for open atomic objects and defined in the input (left-hand side) context Δ.

In the following, we will need a notion of *checkout*, that relates sliced cSLF objects to their unfolded counterpart of SLF.

Definition 3.2 (Checkout)**.** The *checkout* of an object M with respect to a context Δ is the operation of "stripping out" all metavariables in M, replacing them by their definition in Δ. The result is an SLF object $\downarrow_\Delta M$.

$$\downarrow_\Delta \lambda x.\, M = \lambda x.\, \downarrow_\Delta M \tag{3.15}$$
$$\downarrow_\Delta H(S) = H(\downarrow_\Delta S) \tag{3.16}$$
$$\downarrow_\Delta M, S = (\downarrow_\Delta M), (\downarrow_\Delta S) \tag{3.17}$$
$$\downarrow_\Delta \mathtt{X}[\sigma] = \downarrow_\Delta (\mathsf{c}(S)[\sigma]) \qquad \text{if } \mathtt{X}[\Gamma] : P = \mathsf{c}(S) \in \Delta \tag{3.18}$$

This operation is extended homomorphically to kinds K, types A, P, environment Γ and signatures Σ. In Equation (3.18), to compute the checkout of a metavariable, we look up its definition in Δ, and substitute all its free variables with σ. Note that this operation is partial, since substitution is partial.

Let us go over some of the important typing rules of Figure 3.4. Rule MLam is similar to its SLF equivalent, except that its premise returns a context Δ' and a residual body M'; in its conclusion, we reconstruct the sliced term $\lambda x.\, M'$ that forms our residual. In SLF's MAtom, the checked and the inferred types were compared up to α-equivalence only; in presence of metavariables, this comparison must be done up to unfolding of those. We thus require for the *checkout* of the given type $\downarrow_\Delta P$ to be equal to the *checkout* of the inferred type $\downarrow_{\Delta'} P'$.

$\boxed{\Delta;\Gamma \vdash A \text{ type} \Rightarrow \Delta';A'}$ Type

$$\textsc{APrd}\ \frac{\Delta;\Gamma \vdash A \text{ type} \Rightarrow \Delta_1;A' \qquad \Delta_1;\Gamma,x:A \vdash B \text{ type} \Rightarrow \Delta_2;B'}{\Delta;\Gamma \vdash \Pi x:A.B \text{ type} \Rightarrow \Delta_2;\Pi x:A'.B'}$$

$$\textsc{AAtm}\ \frac{\mathsf{a}:K \in \Sigma \qquad \Delta;\Gamma;K \vdash S \Rightarrow \Delta';S':*}{\Delta;\Gamma \vdash \mathsf{a}\,(S) \text{ type} \Rightarrow \Delta';\mathsf{a}\,(S')}$$

$\boxed{\Delta;\Gamma;K \vdash S \Rightarrow \Delta';S':*}$ Type spine

$$\textsc{ASCons}\ \frac{\Delta;\Gamma \vdash M:A \Rightarrow \Delta_1;M' \qquad \Delta_1;\Gamma;K[x/M'] \vdash S \Rightarrow \Delta_2;S':*}{\Delta;\Gamma;\Pi x:A.K \vdash M,S \Rightarrow \Delta_2;M',S':*}$$

$$\textsc{ASNil}\ \frac{}{\Delta;\Gamma;* \vdash \cdot \Rightarrow \Delta;\cdot:*}$$

$\boxed{\Delta;\Gamma \vdash K \text{ kind} \Rightarrow \Delta';K'}$ Kind

$$\textsc{KPrd}\ \frac{\Delta;\Gamma \vdash A \text{ type} \Rightarrow \Delta_1;A' \qquad \Delta_1;\Gamma,x:A \vdash K \text{ kind} \Rightarrow \Delta_2;K'}{\Delta;\Gamma \vdash \Pi x:A.K \text{ kind} \Rightarrow \Delta_2;\Pi x:A'.K'}$$

$$\textsc{KType}\ \frac{}{\Delta;\Gamma \vdash * \text{ kind} \Rightarrow \Delta;*}$$

Figure 3.5: Bidirectional typing algorithm for cSLF (2/3)

When meeting an applied constant $\mathsf{c}(S)$, we must slice it by giving it a metavariable name and recording it in the context. This is what rule FConst does: the spine S is recursively sliced into S', and we add a new binding $\mathtt{X}[\Gamma'] : P = \mathsf{c}(S')$ (with $\mathtt{X}$ fresh) to the output context Δ'. This new binding represents the fact that $\mathsf{c}(S')$ has type P and can, from now on, be referred to as $\mathtt{X}$. All its free variables are in Γ, however, we strengthen this environment to the strict minimum, by taking $\Gamma' = \mathrm{stren}_{\mathsf{c}(S')}(\Gamma)$. While we could have saved the whole environment Γ, taking its minimum subset typing F allows to shorten greatly the substitutions attached to metavariables. The residual object of this slicing is the metavariable $\mathtt{X}$ applied to the identity substitution $\mathrm{id}_{\Gamma'}$, that takes the recorded term from Γ' back to Γ.

Rule FMeta is the essence of the proof reuse system: typing a metavariable $\mathtt{X}[\sigma]$ does not require to type its definition $\mathsf{c}(S)$, since its type P has already been computed and stored in Δ. We just check that the provided substitution σ agrees with the local environment Γ' of $\mathtt{X}$.

In rule SCons slicing a spine M, S, the context is threaded, i.e., a context Δ_1 is returned for M, it is then passed as input to the slicing of S, which returns the final context Δ_2 (the same applies to several of the remaining rules). Note that in the second premise, we substitute the *residual* M' for x in B, and not the input term M. This will guarantee that the type P eventually returned will contain only sliced object i.e., it will be sliced itself (see theorem 3.4). The same precaution of substituting residuals is taken in rules FMeta, ASCons and σCons. Rules for substitutions, types, kind should be straightforward as they are identical to their SLF equivalent, save the threading of the context and the recomposition of the residual subterms in the returned residual.

One could be surprised to see that the signature judgment carries and returns a context too (after all, it serves only to verify that a signature is well-typed). This is because we slice once and for all the objects contained in the declared constants' types; this way we do not need to slice their type each time we use them. Given an SLF signature Σ_0, one uses it in cSLF by first slicing it in the empty context, to get the pair of a Δ and Σ ($\cdot \vdash \Sigma_0$ sig $\Rightarrow \Delta; \Sigma$); then this pair can be used to slice e.g., object M ($\Delta; \Gamma \vdash_\Sigma M : A \Rightarrow \Delta'; M'$). This gives rise to a rather non-standard binding structure between signatures and context: Δ contains constants c declared in Σ, and Σ contains metavariables $\mathtt{X}$ defined in Δ.

Finally, environment and context judgments are declared only for metatheoretical reasons, and are not part of the implementation of cSLF. In particular, they do not need to slice their argument; however they both need to carry a context Δ defining the metavariables they may contain.

$\boxed{\Delta \vdash \Sigma \text{ sig} \Rightarrow \Delta'; \Sigma'}$ Signature

$$\textsc{SigType}\ \frac{\Delta \vdash \Sigma \text{ sig} \Rightarrow \Delta_1; \Sigma' \qquad \Delta_1; \cdot \vdash_{\Sigma'} A \text{ type} \Rightarrow \Delta_2; A' \qquad \mathsf{c} \notin \text{dom}(\Sigma')}{\Delta \vdash \Sigma, \mathsf{c} : A \text{ sig} \Rightarrow \Delta_2; (\Sigma', \mathsf{c} : A')}$$

$$\textsc{SigKind}\ \frac{\Delta \vdash \Sigma \text{ sig} \Rightarrow \Delta_1; \Sigma' \qquad \Delta_1; \cdot \vdash_{\Sigma'} K \text{ kind} \Rightarrow \Delta_2; K' \qquad \mathsf{a} \notin \text{dom}(\Sigma')}{\Delta \vdash \Sigma, \mathsf{a} : K \text{ sig} \Rightarrow \Delta_2; (\Sigma', \mathsf{a} : K)}$$

$$\textsc{SigNil}\ \frac{}{\Delta \vdash \cdot \text{ sig} \Rightarrow \Delta; \cdot}$$

$\boxed{\Delta \vdash \Gamma \text{ env}}$ Environment

$$\textsc{EnvCons}\ \frac{\Delta \vdash \Gamma \text{ env} \qquad \Delta; \Gamma \vdash A \text{ type} \Rightarrow \Delta'; A'}{\Delta \vdash \Gamma, x : A \text{ env}} \qquad \textsc{EnvNil}\ \frac{}{\Delta \vdash \cdot \text{ env}}$$

$\boxed{\Delta \vdash \Delta' \text{ ctx}}$ Context

$$\textsc{CtxCons}\ \frac{\Delta_0 \vdash \Delta_1 \text{ ctx} \qquad \mathsf{c} : A \in \Sigma \qquad \Delta_1; \Gamma; A \vdash S \Rightarrow \Delta_2; S' : P'}{\Delta_0 \vdash \Delta_2, \mathtt{X}[\Gamma] : P = \mathsf{c}(S) \text{ ctx}} \qquad \textsc{CtxId}\ \frac{}{\Delta \vdash \Delta \text{ ctx}}$$

Figure 3.6: Bidirectional typing algorithm for cSLF (3/3)

3.2.3 | CASE STUDY

Let us consider again examples 3.2 and 3.3 and implement in cSLF these incremental type checks. For the purpose of this example, we will use an *intrinsic* or *Church-style* encoding of the simply-typed λ-calculus [Benton et al., 2012]: instead of first defining the untyped λ-terms, and then superimposing them derivations, a relation tying a term to its type (the *extrinsic* encoding), we directly define the type of well-typed terms $\mathsf{tp}\ A$ indexed by their types A. Under the *Curry-Howard isomorphism*, we are defining the types of proofs of a proposition A. Here is our initial signature Σ:

$$
\begin{array}{l}
\mathsf{tp} : *. \\
\mathsf{nat} : \mathsf{tp}. \\
\mathsf{arr} : \mathsf{tp} \to \mathsf{tp} \to \mathsf{tp}. \\
\mathsf{tm} : \mathsf{tp} \to *. \\
\mathsf{o} : \mathsf{tm}\ \mathsf{nat}. \\
\mathsf{s} : \mathsf{tm}\ \mathsf{nat} \to \mathsf{tm}\ \mathsf{nat}. \\
\mathsf{lam} : \Pi A : \mathsf{tp}.\ \Pi B : \mathsf{tp}.\ (\mathsf{tm}\ A \to \mathsf{tm}\ B) \to \mathsf{tm}\ (\mathsf{arr}\ A\ B). \\
\mathsf{app} : \Pi A : \mathsf{tp}.\ \Pi B : \mathsf{tp}.\ \mathsf{tm}\ (\mathsf{arr}\ A\ B) \to \mathsf{tm}\ A \to \mathsf{tm}\ B.
\end{array}
$$

We can feed it to the system for checking and slicing, and get a sliced signature Σ_0 and a context Δ_0. It corresponds to judgment $\cdot \vdash \Sigma\ \mathsf{sig} \Rightarrow \Delta_0; \Sigma_0$. Here is Σ_0:

$$
\begin{array}{l}
\mathsf{tp} : *. \\
\mathsf{nat} : \mathsf{tp}. \\
\mathsf{arr} : \mathsf{tp} \to \mathsf{tp} \to \mathsf{tp}. \\
\mathsf{tm} : \mathsf{tp} \to *. \\
\mathsf{o} : \mathsf{tm}\ \mathsf{E}. \\
\mathsf{s} : \mathsf{tm}\ \mathsf{E} \to \mathsf{tm}\ \mathsf{E}. \\
\mathsf{lam} : \Pi A : \mathsf{tp}.\ \Pi B : \mathsf{tp}.\ (\mathsf{tm}\ A \to \mathsf{tm}\ B) \to \mathsf{tm}\ \mathsf{F}[A/A; B/B]. \\
\mathsf{app} : \Pi A : \mathsf{tp}.\ \Pi B : \mathsf{tp}.\ \mathsf{tm}\ \mathsf{F}[A/A; B/B] \to \mathsf{tm}\ A \to \mathsf{tm}\ B.
\end{array}
$$

Each applied object constant in types has been replaced by a metavariable: nat has been replaced by E (remember that nat alone is a constant applied to an empty spine of arguments), and arr A B by F (since this expression has free variables A and B, each occurrence of F must carry a substitution). The produced context Δ_0 defining these metavariables is:

$$
\begin{array}{l}
\mathsf{E}[] : \mathsf{tp} = \mathsf{nat}; \\
\mathsf{F}[A : \mathsf{tp}, B : \mathsf{tp}] : \mathsf{tp} = \mathsf{arr}\ A\ B
\end{array}
$$

For concision, we took the liberty to collapse metavariables that have the same definition.[4]

[4] In the described system, we would get as many metavariables with definition nat as there are occurrences of nat in the signature. The implementation employs the same trick: names of metavariables reflect their content.

Now that we have a sliced signature, we can check and slice objects. The initial λ-term of example 3.2, $(\lambda f\, x.f\ x)\ (\lambda x.\, \mathbf{s}(x))\ (\mathbf{s}(\mathbf{o}))$, is encoded by the atomic object F_0:

```
app nat nat
  (app (arr nat nat) (arr nat nat)
    (lam (arr nat nat) (arr nat nat) λf. lam nat nat λx. app nat nat f x)
    (lam nat nat λx. s x))
  (s o)
```

Note how its intrinsic encoding obliges to annotate it largely with types, not only at λ-abstractions, but also at applications; this is due to the fact that SLF is an encoding with *local verification* (see Section 3.2.1).

We feed the system with F_0, in signature Σ_0 and context Δ_0. The corresponding judgment is $\Delta_0; \cdot \vdash_{\Sigma_0} F_0 \Rightarrow \Delta_1; F'_0 : P'_0$. We get back the *residual* $F'_0 = \mathtt{X}$, of type $P'_0 = \mathtt{E}$, in the new context Δ_1 which is Δ_0 extended with:

```
G[] : tp = arr E E;
H[x : tm E] : tm E = s x;
I[] : tm E = s J;
J[] : tm E = o;
K[] : tm G = app G G L M;
L[] : tm F[G; G] = lam G G λf. N[f/f];
M[] : tm F[E; E] = lam E E λx. H[x/x];
N[f : tm G] : tm F[E; E] = lam E E λx. O[f/f; x/x];
O[f : tm G, x : tm E] : tm E = app E E f x;
X[] : tm E = app E E K I
```

Our initial term is well-typed and sliced: we can check that $\downarrow_{\Delta_1} F'_0 = F_0$. We can now reuse the metavariables in Δ_1 to build incrementally the modified term $(\lambda f.\, \lambda x.f\ x)\ (\lambda x.\, \mathbf{s}(\mathbf{s}(x)))\ (\mathbf{s}(\mathbf{o}))$. We call F_1 its shared encoding as an atomic object:

```
app E E (app G G L (lam E E λx. s H[x/x])) I
```

Note how we reused type annotations `E` and `G`, the functional term `L` that has not changed, the last argument `I` and the inner successor `H`. We had however to reconstruct the whole *path* from the tip of the term to the location of the change, i.e., the two `app` and the `lam` nodes. Note that $\downarrow_{\Delta_1} F_1$ is indeed the encoding of our modified term $(\lambda f.\, \lambda x.f\ x)\ (\lambda x.\, \mathbf{s}(\mathbf{s}(x)))\ (\mathbf{s}(\mathbf{o}))$. Again, we feed it to the system in the context of the new acquired typing information Δ_1, an operation corresponding to judgment $\Delta_1; \cdot \vdash F_1 \Rightarrow \Delta_2; F'_1 : P'_1$. We get back $F'_1 = \mathtt{Q}$, $P'_1 = \mathtt{E}$ and Δ_2 which extends Δ_1 with:

```
P[x : tm E] : tm E = s H[x/x];
Q[] : tm E = app E E R I;
```

```
R[] : tm G = app G G L S;
S[] : tm F[E; E] = lam E E λ_x. P[x/x]
```

Note that $\downarrow_{\Delta_2} F_1'$ is again the encoding of our modified term. Note also that the slicing of F_1 takes a time proportional to the size of F_1, not of $\downarrow_{\Delta_1} F_1$, since the system rules are directed by the form of the shared term.

Finally, we can continue and build another *diff*, starting from the typing information we acquired in Δ_2. We encode λ-term $(\lambda f\, x.f\, x)\ (\lambda x.\mathbf{s}(\mathbf{s}(\mathbf{o})))\ (\mathbf{s}(\mathbf{o}))$ by the object F_2:

```
app E E (app G G L (lam E E (λ_x. P[x/I]))) I
```

Instead of, as before, reconstructing the whole path from the root of the term to the location of the change (the inner **o**), this time we use the fact that we have already type-checked the slice `P` where this inner term was a free variable (it is a *context*: a term with an instantiable hole). Up to now, all substitutions attached to metavariables of input terms were the identity; this is the first use of substituting by an already typed piece of term `I`. Again, judgment $\Delta_2; \cdot \vdash F_2 \Rightarrow \Delta_3; F_2' : P_2'$ allows to feed F_2 to the system for slicing, and get back $F_2' = W$, $P_2' =$ `E` in context Δ_3 which extends Δ_2 with:

```
T[] : tm G = app G G L U;
U[] : tm F[E; E] = lam E E λx. V;
V[] : tm E = s I;
W[] : tm E = app E E T I
```

Remark that an incorrect reuse would lead to a typing error. For instance, there is no derivation of judgment $\Delta_3; \cdot \vdash$ `app E E U U` $\Rightarrow \Delta'; F' : P'$.

3.2.4 | METATHEORY

How can we be sure that an incrementally checked piece of term is indeed well-typed, i.e., that the incremental type checking of a cSLF object succeeds exactly when the corresponding batch checking in SLF does? In the following, we prove the main properties of cSLF, which shows the two main ideas explained earlier:

- cSLF objects are only SLF objects, but with the ability to refer to previously typed pieces of objects;
- the typing and slicing process is such that the residual of an object is a sliced object, and is also well-typed.

For this, we prove what we (maybe abusively) will call soundness and completeness of cSLF with respect to SLF. Their composition will show that type checking in cSLF is decidable.

First, soundness establishes that the residual of typing an object is always an object which, stripped off of its metavariables, is well-typed with respect to SLF;

we show at the same time that the returned context is also well-typed. This is established by mutual induction over the main judgments. These judgments hold in a signature Σ that has been generated by typing a signature Σ_0 containing no metavariables. Clause *1.* proves such a signature is sound, i.e., it is well-typed in SLF. A noticeable case is *3. (d)*: it states that a well-typed contexts necessarily contains only well-typed objects.

Theorem 3.1 (Soundness)**.**

1. *If* $\cdot \vdash \Sigma_0$ sig $\Rightarrow \Delta; \Sigma$ *then* $\vdash \Delta$ ctx *and* $\vdash \downarrow_\Delta \Sigma$ sig
2. *If* $\Delta \vdash \Gamma$ env *then* $\vdash \downarrow_\Delta \Gamma$ env
3. *Suppose* $\cdot \vdash \Sigma_0$ sig $\Rightarrow \Delta_0; \Sigma$ *and* $\Delta_0 \vdash_\Sigma \Delta$ ctx *and* $\Delta \vdash_\Sigma \Gamma$ env.
   (a) *If* $\Delta; \Gamma \vdash_\Sigma F \Rightarrow \Delta'; F' : P'$ *then* $\vdash \Delta'$ ctx *and* $\downarrow_\Delta \Gamma \vdash_{\Sigma_0} \downarrow_{\Delta'} F' \Rightarrow \downarrow_{\Delta'} P'$.
   (b) *If* $\Delta; \Gamma \vdash_\Sigma M : A \Rightarrow \Delta'; M'$ *then* $\vdash \Delta'$ ctx *and* $\downarrow_\Delta \Gamma \vdash_{\Sigma_0} \downarrow_{\Delta'} M' \Leftarrow \downarrow_\Delta A$
   (c) *If* $\Delta; \Gamma; A \vdash_\Sigma S \Rightarrow \Delta'; S' : P'$ *then* $\vdash \Delta'$ ctx *and* $\downarrow_\Delta \Gamma; \downarrow_\Delta A \vdash_{\Sigma_0} \downarrow_{\Delta'} S' \Rightarrow \downarrow_{\Delta'} P'$
   (d) *If* $\Delta \vdash_\Sigma \Delta_1 @ \Delta_2$ ctx *and* $\mathtt{X}[\Gamma] : P = \mathtt{c}(S) \in \Delta_1$ *and* $\mathtt{c} : A \in \Sigma$ *then* $\downarrow_{\Delta_1} \Gamma; \downarrow_{\Delta_1} A \vdash_{\Sigma_0} \downarrow_{\Delta_1} S \Rightarrow \downarrow_{\Delta_1} P$
   (e) *If* $\Delta; \Gamma \vdash_\Sigma A$ type $\Rightarrow \Delta'; A'$ *then* $\vdash \Delta'$ ctx *and* $\downarrow_\Delta \Gamma \vdash_{\Sigma_0} \downarrow_{\Delta'} A'$ type
   (f) *If* $\Delta; \Gamma \vdash_\Sigma K$ kind $\Rightarrow \Delta'; K'$ *then* $\vdash \Delta'$ ctx *and* $\downarrow_\Delta \Gamma \vdash_{\Sigma_0} \downarrow_{\Delta'} K'$ kind

Proof. We first prove clauses *1*, *2* and *3. (a)-(d)* by mutual induction on the given derivations, and on Δ_1 for *(d)*. Clauses *(e)* and *(f)* are proved by single induction over the given derivation. Most cases are straightforward: they amount to unfold the definition of checkout and apply the appropriate rule of SLF to the goal. Only a few are more involved; let us go over them:

Case FConst Let us call Δ_0 the enlarged context $\Delta', \mathtt{X}[\Gamma'] : P' = \mathtt{c}(S')$. We must show $\downarrow_\Delta \Gamma \vdash \downarrow_{\Delta_0} \mathtt{X}[\mathrm{id}_{\Gamma'}] \Rightarrow \downarrow_{\Delta_0} P'$, knowing by induction $\downarrow_\Delta \Gamma'; \downarrow_\Delta A \vdash \downarrow_{\Delta'} S' \Rightarrow \downarrow_{\Delta'} P'$. By lemma 1.12, we have $\downarrow_\Delta \Gamma; \downarrow_\Delta A \vdash \downarrow_{\Delta'} S' \Rightarrow \downarrow_{\Delta'} P'$ since $\mathrm{stren}_{\mathtt{c}(S')}(\Gamma) = \Gamma'$. By the definition of checkout, it suffices to prove $\downarrow_\Delta \Gamma \vdash \downarrow_{\Delta_0} \mathtt{c}(S')[\mathrm{id}_{\Gamma'}] \Rightarrow \downarrow_{\Delta_0} P'$, which reduces to $\downarrow_\Delta \Gamma \vdash \downarrow_{\Delta_0} \mathtt{c}(S') \Rightarrow \downarrow_{\Delta_0} P'$ by lemma 1.11, which in turn reduces to our hypothesis by SLF's FConst.

Case FMeta We must show $\downarrow_\Delta \Gamma \vdash \downarrow_{\Delta'} \mathtt{X}[\sigma'] \Rightarrow \downarrow_{\Delta'} P'[\sigma']$, which reduces to $\downarrow_\Delta \Gamma \vdash \downarrow_{\Delta'} (\mathtt{c}(S))[\sigma'] \Rightarrow \downarrow_{\Delta'} P'[\sigma']$ seen that we know $\mathtt{X}[\Gamma'] : P = \mathtt{c}(S) \in \Delta'$ (a premise of FMeta). By induction, we know that $\downarrow_\Delta \Gamma \vdash \downarrow_{\Delta'} \sigma' \Leftarrow \downarrow_{\Delta'} \Gamma'$, so by theorem 1.4, we only need to show $\downarrow_\Delta \Gamma' \vdash \downarrow_{\Delta'} \mathtt{c}(S) \Rightarrow \downarrow_{\Delta'} P'$, which is induced by FConst and clause 4.

Case $\Delta = \cdot$ By induction on the derivation of $\mathtt{X}[\Gamma] : P = \mathtt{c}(S) \in \Delta$, we derive a contradiction.

Case $\Delta = \Delta', \mathtt{Y}[\Gamma] : P = \mathtt{c}(S)$ We conclude easily by induction on the derivation of $\mathtt{X}[\Gamma] : P = \mathtt{c}(S) \in \Delta$.

□

The residual M' of an object M differs from M only by the recursive replacement of subterms by metavariables standing for these subterms. In other words, checking out an object or its residual gives the same result. This is established by this simple proposition:

Lemma 3.1 (Residual). *Suppose* $\cdot \vdash \Sigma_0$ sig $\Rightarrow \Delta_0; \Sigma$ *and* $\Delta_0 \vdash \Delta$ ctx *and* $\Delta \vdash \Gamma$ env. *Then:*

1. *If* $\Delta; \Gamma \vdash F \Rightarrow \Delta'; F' : P'$ *then* $\downarrow_\Delta F = \downarrow_{\Delta'} F'$.
2. *If* $\Delta; \Gamma \vdash M : A \Rightarrow \Delta'; M'$ *then* $\downarrow_\Delta M = \downarrow_{\Delta'} M'$.
3. *If* $\Delta; \Gamma; A \vdash S \Rightarrow \Delta'; S' : P'$ *then* $\downarrow_\Delta S = \downarrow_{\Delta'} S'$.
4. *If* $\Delta; \Gamma \vdash \sigma : \Gamma' \Rightarrow \Delta'; \sigma'$ *then* $\downarrow_\Delta \sigma = \downarrow_{\Delta'} \sigma'$.
5. *If* $\Delta; \Gamma \vdash A$ type $\Rightarrow \Delta'; A'$ *then* $\downarrow_\Delta A = \downarrow_{\Delta'} A'$.
6. *If* $\Delta; \Gamma \vdash K$ kind $\Rightarrow \Delta'; K'$ *then* $\downarrow_\Delta K = \downarrow_{\Delta'} K'$.

Proof. By induction on the derivation. The only non-trivial case is FConst:

$$\begin{aligned} \downarrow_{\Delta', \mathtt{X}[\Gamma']:P=\mathsf{c}(S')} \mathtt{X}[\Gamma'] &= \downarrow_{\Delta', \mathtt{X}[\Gamma']:P=\mathsf{c}(S')} (\mathsf{c}(S')[\Gamma']) && (3.19)\\ &= \downarrow_{\Delta', \mathtt{X}[\Gamma']:P=\mathsf{c}(S')} (\mathsf{c}(S')) && (3.20)\\ &= \downarrow_{\Delta'} (\mathsf{c}(S')) && (3.21) \end{aligned}$$

Equation (3.20) is by lemma 1.11 and corollary 1.3 and theorem 3.1; Equation (3.21) holds since $\mathtt{X} \notin \mathrm{dom}(\Delta')$. □

Corollary 3.1 (Constant lookup). *Suppose* $\cdot \vdash \Sigma_0$ sig $\Rightarrow \Delta_0; \Sigma$ *and* $\Delta_0 \vdash \Delta$ ctx. *Then:*

1. *If* $\mathsf{c} : A \in \Sigma_0$ *then* $\mathsf{c} : A' \in \Sigma$ *and* $\downarrow_\Delta A' = A$
2. *If* $\mathsf{a} : K \in \Sigma_0$ *then* $\mathsf{a} : K' \in \Sigma$ *and* $\downarrow_\Delta K' = K$

Proof. By induction over the derivation of $\cdot \vdash \Sigma_0$ sig $\Rightarrow \Delta_0; \Sigma$, using lemma 3.1. □

Next, completeness establishes a dual (but not inverse) property: if a well-typed object (with respect to SLF) is given as input to the slicer, then it *will* return a sliced version of this object. More precisely, suppose an object with metavariables, which checkout is well-typed in SLF. Then there exists a corresponding derivation in cSLF.

Theorem 3.2 (Completeness).

1. *If* $\vdash \downarrow_\Delta \Sigma$ sig *then* $\vdash \Sigma$ sig $\Rightarrow \Delta'; \Sigma'$.
2. *If* $\vdash \downarrow_\Delta \Gamma$ env *then* $\Delta \vdash \Gamma$ env.
3. *Suppose* $\cdot \vdash \Sigma_0$ sig $\Rightarrow \Delta_0; \Sigma$ *and* $\Delta_0 \vdash \Delta$ ctx *and* $\Delta \vdash \Gamma$ env. *Then:*
   (a) *If* $\downarrow_\Delta \Gamma \vdash \downarrow_\Delta F \Rightarrow P$ *then there is* Δ', F', P' *such that* $\Delta; \Gamma \vdash F \Rightarrow \Delta'; F' : P'$ *and* $\downarrow_{\Delta'} P' = P$.
   (b) *If* $\downarrow_\Delta \Gamma \vdash \downarrow_\Delta M \Leftarrow \downarrow_\Delta A$ *then there is* Δ', M' *such that* $\Delta; \Gamma \vdash M : A \Rightarrow \Delta'; M'$.

(c) *If* $\downarrow_\Delta \Gamma; \downarrow_\Delta A \vdash \downarrow_\Delta S \Rightarrow P$ *then there is* Δ', S', P' *such that* $\Delta; \Gamma; A \vdash S \Rightarrow \Delta'; S' : P'$ *and* $\downarrow_{\Delta'} P' = P$.
(d) *If* $\downarrow_\Delta \Gamma \vdash \downarrow_\Delta \sigma \Leftarrow \downarrow_\Delta \Gamma'$ *then there is* Δ', σ' *such that* $\Delta; \Gamma \vdash \sigma : \Gamma' \Rightarrow \Delta'; \sigma'$
(e) *If* $\downarrow_\Delta \Gamma \vdash \downarrow_\Delta A$ type *then there is* Δ', A' *such that* $\Delta; \Gamma \vdash A$ type $\Rightarrow \Delta'; A'$.
(f) *If* $\downarrow_\Delta \Gamma \vdash \downarrow_\Delta K$ kind *then there is* Δ', K' *such that* $\Delta; \Gamma \vdash K$ kind $\Rightarrow \Delta'; K'$.

Proof. By mutual, functional induction on the computation of the checkout of the judged syntactic category: $\downarrow_\Delta \Sigma$, $\downarrow_\Delta \Gamma$, $\downarrow_\Delta F$, $\downarrow_\Delta M$ etc. Most cases are straightforward, and amount to inverse the given derivation and apply the induction hypothesis, relying on corollary 3.1 for cases involving constants and theorem 3.1 clause *3. (d)* for the case involving a metavariable. □

Thanks to these two results, we conclude that typing/slicing cSLF objects is decidable. As usual, this proof is carried in an informal intuitionistic logic and thus serves as a decidability result.

Theorem 3.3 (Decidability of typing)**.** *Suppose* $\cdot \vdash \Sigma_0$ sig $\Rightarrow \Delta_0; \Sigma$ *and* $\Delta_0 \vdash \Delta$ ctx *and* $\Delta \vdash \Gamma$ env. *Then:*

1. *For all* M *and* A*, either there is* Δ' *and* M' *such that* $\Delta; \Gamma \vdash M : A \Rightarrow \Delta'; M'$ *or not;*
2. *For all* F *either there is* Δ', F' *and* P' *such that* $\Delta; \Gamma \vdash F \Rightarrow \Delta'; F' : P'$ *or not;*
3. *For all* A *and* S *either there is* Δ', S', P' *such that* $\Delta; \Gamma; A \vdash S \Rightarrow \Delta'; S' : P'$ *or not;*
4. *For all* σ *and* Γ_0*, either there is* Δ' *and* σ' *such that* $\Delta; \Gamma \vdash \sigma : \Gamma_0 \Rightarrow \Delta'; \sigma'$ *or not;*
5. *For all* K*, either there is* Δ' *and* K' *such that* $\Delta; \Gamma \vdash K$ kind $\Rightarrow \Delta'; K'$ *or not;*
6. *For all* A*, either there is* Δ' *and* A' *such that* $\Delta; \Gamma \vdash A$ type $\Rightarrow \Delta'; A'$ *or not;*
7. *For all* K *and* S *either there is* Δ' *and* S' *such that* $\Delta; \Gamma; K \vdash S \Rightarrow \Delta'; S' : *$ *or not;*
8. *For all* Σ*, either there is* Δ' *and* Σ' *such that* $\Delta \vdash \Sigma$ sig $\Rightarrow \Delta'; \Sigma'$ *or not;*

Proof. Take the first clause. We reason by case on whether $\downarrow_\Delta \Gamma \vdash \downarrow_\Delta M \Leftarrow \downarrow_\Delta A$ is well-typed, relying on theorem 1.3. If it is, then by theorem 3.2 there is M' and a derivation allowing to conclude. Otherwise, by the contraposition of theorem 3.1, there is no such derivation. All other clauses are proved the same way. □

Finally, the following shows the relationship between the input object F of our algorithm and its output F': while the input can be any object with metavariables, the output is in a form where all constants (and their arguments) have been sliced, i.e., given a unique metavariable name, lifted from their environment and stored in the context:

Definition 3.3 (Sliced form)**.** An atomic cSLF object F (*resp.* M, S, P, A, K, σ, Σ, Γ) is in *sliced form* if it contains no object constant $\mathtt{c}$. A context Δ is sliced if it is empty, or written $\Delta, \mathtt{X}[\Gamma] : P = \mathtt{c}(S)$ where Δ, S and P are sliced.

Finally, this proposition states that the slicer does its job, i.e., always returns a sliced term:

Theorem 3.4 (Sliced form)**.** *Let Δ and Γ be sliced. Then:*

1. *If $\Delta; \Gamma \vdash F \Rightarrow \Delta'; F' : P'$, then Δ', F' and P' are sliced.*
2. *If $\Delta; \Gamma \vdash M : A \Rightarrow \Delta'; M'$ and A sliced then Δ' and M' are sliced.*
3. *If $\Delta; \Gamma; A \vdash S \Rightarrow \Delta'; S' : P$ and A sliced then Δ' and S' are sliced.*
4. *If $\Delta; \Gamma \vdash \sigma : \Gamma' \Rightarrow \Delta'; \sigma'$ and Γ' sliced then Δ' and σ' are sliced.*
5. *If $\Delta; \Gamma \vdash A$ type $\Rightarrow \Delta'; A'$ then Δ' and A' are sliced.*
6. *If $\Delta; \Gamma \vdash K$ kind $\Rightarrow \Delta'; K'$ then Δ' and K' are sliced.*
7. *If $\Delta \vdash \Sigma$ sig $\Rightarrow \Delta'; \Sigma'$ then Δ' and Σ' are sliced.*

Proof. By easy induction on the derivation. In case FConst, we put $\mathsf{c}(S')$ in the context, and Γ' is sliced. □

3.3 | cDLF: COMPUTING DLF CERTIFICATES INCREMENTALLY

In Section 3.2.3, we constructed incrementally a λ-term that was well-typed *by construction*, since the encoding of it was *intrinsic*: there was no distinction between terms and derivations. Here, we will see the need to separate a term and its typing derivation, and a way to still be able to incrementally type check them.

3.3.1 | PRESENTATION

There are two obvious reasons why we cannot really call cSLF a general-purpose incremental type checker and we would want to go further. First, the terms we had to provide in Section 3.2.3 needed to carry full type annotations, which are naturally not present in real programs. While many programming languages need some typing input from the user, notably on binders, we reasonably cannot imagine putting type information on every subterm. A possible answer to this problem is a mechanism for *type reconstruction*. As mentioned earlier in Section 2.5, a notable deficiency of LF objects is that they are very redundant, since each rule application must carry its *substitution*. For instance, in object

```
app (arr nat nat) (arr nat nat)
  (lam (arr nat nat) (arr nat nat) λx.x) (lam nat nat λx.x)
```

all occurrences of annotation arr nat nat are bound to be equal, and so are occurrences of nat, otherwise it would be ill-typed. In other words, this information could be eluded in all cases except one, for instance the first annotation on each lam constructor, and reconstructed everywhere else at typing time; when declaring constants, we would have to indicate or compute which arguments are reconstructible and which are not. Type reconstruction is a very common feature found in most systems

implementing some form of dependent type theory, among which Twelf and Beluga, but also Coq and Matita, and has been extensively studied in the literature [Pym, 1992, Pfenning and Schürmann, 1998, Dowek, 2001, Pientka, 2013, Asperti et al., 2012]. It is thus helpful not only as a way to compress LF proofs as in Section 2.5, but also to relieve the user from entering all type information. For instance, our first *delta* in the previous example could then be written simply

```
app (app L (lam E λx. s H[x/x])) I
```

Yet, we cannot expect a generic type reconstruction algorithm to be powerful enough to infer this implicit information for any typed language. In fact, another problem lies directly in the fact that *intrinsic* encodings cannot represent faithfully a type system where some rules are *silent*. For instance, we studied in Section 2.4.2 the design of a type checker for System $\mathsf{T}_{<:}$. This system comprises a subtyping rule

$$\frac{\vdash M : A' \qquad \vdash A' \leq A}{\vdash M : A}\ \textsc{Sub}$$

that is not directed by the syntax of M, even though typing remains decidable in its presence. An intrinsic encoding of it would have to have a sub syntactic construct apparent in the term. Another example of such a silent rule is the polymorphism of Curry-style System F [Girard, 1972, Reynolds, 1974]:

$$\frac{\vdash M : A}{\vdash M : \forall \alpha.A}\ \textsc{Inst} \qquad \frac{\vdash M : \forall \alpha.A}{\vdash M : A[x/B]}\ \textsc{Gen}$$

In this case, it is well-known that type checking becomes undecidable [Wells, 1994], even though an algorithmic restriction of it is used in the ML family type systems [Milner, 1978].

To conclude these two remarks, a full treatment of incremental type checking requires a full treatment of type checking by itself: the terms and the typing derivations are two separate entities, and terms must be considered untyped before being superimposed typing derivations that possibly have a much more involved structure. The program computing these derivations can then be arbitrarily complex.

In this section, we propose to extend cSLF with an untyped computational language, actually the exact same extension that made SLF into DLF in Chapter 2, only with a representational layer that allows sharing in proof objects. As we shall see, we obtain a way to turn any implementation of a certifying type checker into an incremental type checker, supporting the mechanism of incrementality by sharing of subderivations.

Maybe surprisingly, the key to express sharing will be *function inverses* and the *contraction* reduction rule described in Section 2.3. Indeed, if a type checker takes

$$
\begin{array}{rcll}
\Sigma & ::= & \ldots \mid \Sigma, f : A = \text{“}T\text{”} & \text{Signature} \\
H & ::= & \ldots \mid \text{“x”} \mid f \mid f^n & \text{Head} \\
T & ::= & U \mid \textbf{fun}\ \mathtt{x} \to T & \text{Terms} \\
U & ::= & \text{«}F\text{»} \mid \textbf{match}\ U\ \textbf{under}\ \Gamma\ \textbf{with}\ C & \text{Atomic term} \\
C & ::= & \cdot \mid \text{«}Q\text{»} \Rightarrow U \mid C & \text{Branches} \\
Q & ::= & \mathtt{c}(\mathtt{x}, \ldots, \mathtt{x}) & \text{Pattern}
\end{array}
$$

Figure 3.7: Syntax of cDLF, relatively to cSLF (Figure 3.2)

a term to a typing derivation, a term for which the derivation of some subterm is known will need to coerce that already-checked derivation back to a term, which is precisely what inverses do. Contracting the type checking function and its inverse will then take a new sense: that of avoiding computing the derivation when it has already been computed.

3.3.2 | DEFINITION

We turn to the formal description of cDLF. Since all the main ideas of this language were already present in cSLF and DLF, the presentation will be more succinct. The confident reader can skip directly to the next section which presents a usage example, and come back to this formal description afterwards if necessary.

A | SYNTAX, SUBSTITUTIONS, OPERATIONS

The syntax of cDLF, defined on Figure 3.7, is an extension of the syntax of cSLF, actually the exact same extension than what we did in Figure 2.1 to go from SLF to cSLF: we add the ability to define functions in the signature, and to refer to these functions, their inverses and special, computational variables $\mathtt{x}$ in heads. In other words, the cDLF language is the DLF language where atomic objects are extended with metavariables, defined in contexts.

Thus, if a cDLF object contains no metavariables, it is a DLF object; if it contains no function or inverse symbols nor computational variables, it is a cSLF object. we will use these implicit coercions in the following to lighten the definitions.

We first define the two kinds of substitution, for variable and computational variable:

Definition 3.4 (Variable substitution)**.** The *hereditary substitution* operation $M[\sigma]$

is extended homomorphically from that of cSLF (Figure 3.3) by the three equations:

$$\ldots \tag{3.22}$$
$$f(S)[\sigma] = f(S[\sigma]) \tag{3.23}$$
$$f^i(S)[\sigma] = f^i(S[\sigma]) \tag{3.24}$$
$$\text{“x”}(S)[\sigma] = f(S[\sigma]) \tag{3.25}$$

Definition 3.5 (Computational variable substitution)**.** The substitution of computational variables $M[\mathtt{x}/N]$ is extended homomorphically from that of DLF (Figure 2.2) on metavariables and substitutions by the equations:

$$\ldots \tag{3.26}$$
$$(\mathtt{X}[\sigma])[\mathtt{x}/M] = \mathtt{X}[\sigma[\mathtt{x}/M]] \tag{3.27}$$
$$(\cdot)[\mathtt{x}/M] = \cdot \tag{3.28}$$
$$(\sigma, x/N)[\mathtt{x}/M] = (\sigma[\mathtt{x}/N]), x/(M[\mathtt{x}/N]) \tag{3.29}$$

The projections defining the type and code of inverses are unaffected by the presence of metavariables, since they are defined only on the type structure. On the contrary, erasure must be extended trivially:

Definition 3.6 (Projection types and objects)**.** The *projection* type $(A)_n$ and object $\pi_n(A)$ are defined identically to the corresponding definitions in DLF (definitions 2.1 and 2.2).

Definition 3.7 (Erasure)**.** The *erasure* operation $|M|_\Sigma$ is extended homomorphically from that of DLF (definition 2.3) on metavariables and substitutions:

$$\ldots \tag{3.30}$$
$$|\lambda x.\, M|_\Sigma = \lambda x.\, |M|_\Sigma \tag{3.31}$$
$$|\mathtt{X}[\sigma]|_\Sigma = \mathtt{X}[|\sigma|_\Sigma] \tag{3.32}$$
$$|\cdot|_\Sigma = \cdot \tag{3.33}$$
$$|\sigma, x/M|_\Sigma = |\sigma|_\Sigma, x/|M|_\Sigma \tag{3.34}$$

Checkout in cSLF was a partial operation: it was not defined when a metavariable was carrying an ill-typed substitutions. It is “even more partial” in cDLF, since it is undefined on terms containing functions.

Definition 3.8 (Checkout)**.** The *checkout* operation $\downarrow_\Delta M$ maps a cDLF term to an SLF term. Its definition is identical to the cSLF definition (definition 3.2).

$\boxed{\Delta;\Gamma \vdash F \downarrow F' : P}$ Atomic object

$$\textsc{FVar}\ \frac{x : A \in \Gamma \qquad \Delta;\Gamma;A \vdash S \downarrow S' : P}{\Delta;\Gamma \vdash x(S) \downarrow x(S') : P} \qquad \textsc{FConst}\ \frac{\mathsf{c} : A \in \Sigma \qquad \Delta;\Gamma;A \vdash S \uparrow \Delta';S' : P}{\Delta;\Gamma \vdash \mathsf{c}(S) \downarrow \mathsf{c}(S') : P}$$

$$\textsc{FMeta}\ \frac{\mathtt{X}[\Gamma] : P = \mathsf{c}(S) \in \Delta \qquad \Delta;\Gamma \vdash \sigma : \Gamma' \downarrow \sigma'}{\Delta;\Gamma \vdash \mathtt{X}[\sigma] \downarrow \mathtt{X}[\sigma'] : P[\sigma']}$$

$$\textsc{FEval}\ \frac{\begin{array}{ll} f : A = \text{“}T\text{”} \in \Sigma & S' \neq f^0(S_0),\dots,f^n(S_n) \\ \Delta;\Gamma;A \vdash S \downarrow S' : P & \Delta;\Gamma \vdash T \star S' \downarrow F : P \end{array}}{\Delta;\Gamma \vdash f(S) \downarrow F : P}$$

$$\textsc{FInv}\ \frac{f : A = \text{“}T\text{”} \in \Sigma \qquad \Delta;\Gamma;(A)_i \vdash S \downarrow S' : P}{\Delta;\Gamma \vdash f^i(S) \downarrow f^i(S') : P}$$

$$\textsc{FEvalInv}\ \frac{f : A = \text{“}T\text{”} \in \Sigma \qquad \Delta;\Gamma;A \vdash S \downarrow f^0(S_0),\dots,f^n(S_n) : P \qquad \forall i,j,\ S_i = S_j}{\Delta;\Gamma \vdash f(S) \downarrow \pi_n(A) \star S_0 : P}$$

Figure 3.8: Call-by-value weak typed evaluation of cDLF objects (1/5)

B | TYPED EVALUATION AND SLICING

The evaluation algorithm of cDLF is presented on Figures 3.8 to 3.12. With the typed slicing algorithm of cSLF and the typing evaluation of DLF in mind, it should be fairly easy to grasp despite its size. We maintain the invariant that all slices recorded in a context Δ only contain *values*, that is objects with no function or inverse symbols. Like the algorithm of DLF, it is separated in two phases.

First, a *weak evaluation* procedure traverses and type checks an object (Figures 3.8 and 3.9) looking for applied function symbols to evaluate (rule FEval) or contractions to perform (rule FEvalInv), but only within the current scope, i.e., not traversing λ-abstractions. When a function evaluation is triggered, its code is interpreted against its arguments (Figure 3.10). The only differences with the evaluation of DLF are that:

- All rules carry a supplementary context as input, defining all metavariables. Type comparison, which was purely syntactic in DLF, must be done up to unfolding of

$\boxed{\Delta;\Gamma \vdash M : A \downarrow M'}$ Canonical object

$$\frac{}{\Delta;\Gamma \vdash \lambda x.M : (\Pi x : A.B) \downarrow \lambda x.M}\ \textsc{MLam} \qquad \frac{\Delta;\Gamma \vdash F \downarrow F' : P' \qquad \downarrow_{\Delta} P = \downarrow_{\Delta'} P'}{\Delta;\Gamma \vdash F : P \downarrow F'}\ \textsc{MAtom}$$

$\boxed{\Delta;\Gamma;A \vdash S \downarrow S' : P}$ Spine

$$\frac{}{\Delta;\Gamma;P \vdash \cdot \downarrow \cdot : P}\ \textsc{SNil} \qquad \frac{\Delta;\Gamma \vdash M : A \downarrow M' \qquad \Delta;\Gamma;B[x/|M'|] \vdash S \downarrow S' : P}{\Delta;\Gamma;\Pi x : A.B \vdash M,S \downarrow M',S' : P}\ \textsc{SCons}$$

$\boxed{\Delta;\Gamma \vdash \sigma : \Gamma' \downarrow \sigma'}$ Substitution

$$\frac{}{\Delta;\Gamma \vdash \cdot : \cdot \downarrow \cdot}\ \sigma\textsc{Nil} \qquad \frac{\Delta;\Gamma \vdash \sigma : \Gamma' \downarrow \sigma' \qquad \Delta;\Gamma \vdash M : A[|\sigma'|] \downarrow M'}{\Delta;\Gamma \vdash (\sigma, x/M) : (\Gamma', x : A) \downarrow (\sigma', x/M')}\ \sigma\textsc{Cons}$$

Figure 3.9: Call-by-value weak typed evaluation of cDLF objects (2/5)

metavariables. This is seen in the second premise of rule MAtom.[5]

- The weak evaluation of a metavariable (the new rule FMeta, similar to its cSLF homonym) amounts to evaluating only its attached substitution, but does not involve looking at its body, since it does not contain function symbols.
- However, during the evaluation of a function, a metavariable must not block a pattern matching. When a metavariable is the *scrutinee* of a **match** (the new rule CMeta), we unfold its definition once, and continue looking for a matching pattern.
- A judgment is also added to evaluate and type substitutions (rules σNil and σCons), similarly to the corresponding rules in cSLF. Note the type substitution that is performed on the *erased* type in σCons, in the manner of rule SCons and RSCons.

Secondly, the *readback* procedure takes a *weak value* F produced by weak evalua-

[5]Here, this comparison is presented very naively: we normalize completely both types P and P' and compare the result syntactically. A more efficient algorithm is not hard to find: when there is a metavariable on one side, we unfold it only once and compare recursively; this way, we fail earlier. In essence, this is the algorithm described in Harper and Pfenning [2005]. To further accelerate this conversion, metavariable could be tagged as *injective* or not; when we have the same injective metavariable on both sides, then we only need to compare their substitutions. A syntactic criterion for being injective is proposed in Pfenning and Schürmann [1998].

$\boxed{\Delta;\Gamma \vdash T \star S \downarrow F : P}$ Canonical term

$$\textsc{TLam}\ \frac{\Delta;\Gamma \vdash T[\mathtt{x}/M] \star S \downarrow F : P}{\Delta;\Gamma \vdash \mathbf{fun}\ \mathtt{x} \to T \star M, S \downarrow F : P} \qquad \textsc{TAtom}\ \frac{\Delta;\Gamma \vdash U \downarrow F : P}{\Delta;\Gamma \vdash U \star \cdot \downarrow F : P}$$

$\boxed{\Delta;\Gamma \vdash U \downarrow F : P}$ Atomic term

$$\textsc{UAtom}\ \frac{\Delta;\Gamma \vdash F \downarrow F' : P}{\Delta;\Gamma \vdash \langle\!\langle F \rangle\!\rangle \downarrow F' : P} \qquad \textsc{UCase}\ \frac{\Delta;\Gamma@\Gamma' \vdash U \downarrow F : P \qquad \Delta;\Gamma \vdash F \star C \downarrow F' : P'}{\Delta;\Gamma \vdash \mathbf{match}\ U\ \mathbf{under}\ \Gamma'\ \mathbf{with}\ C \downarrow F' : P'}$$

$\boxed{\Delta;\Gamma \vdash F \star C \downarrow F' : P}$ Branches

$$\textsc{CNoMatch}\ \frac{\mathtt{c} \neq \mathtt{c}' \qquad \Delta;\Gamma \vdash \mathtt{c}(S) \star C \downarrow F : P}{\Delta;\Gamma \vdash \mathtt{c}(S) \star (\langle\!\langle \mathtt{c}'(\vec{x}) \rangle\!\rangle \Rightarrow U \mid C) \downarrow F : P}$$

$$\textsc{CMatch}\ \frac{\Delta;\Gamma \vdash U[x_1/M_1, \ldots, x_n/M_n] \downarrow F : P \qquad \forall ij, \text{if } i \neq j \text{ then } x_i \neq x_j}{\Delta;\Gamma \vdash \mathtt{c}(M_1, \ldots, M_n) \star (\langle\!\langle \mathtt{c}(x_1, \ldots, x_n) \rangle\!\rangle \Rightarrow U \mid C) \downarrow F : P}$$

$$\textsc{CMeta}\ \frac{\mathtt{X}[\Gamma'] : P' = \mathtt{c}(S) \in \Delta \qquad \Delta;\Gamma \vdash \mathtt{c}(S[\sigma]) \star C \downarrow F' : P}{\Delta;\Gamma \vdash \mathtt{X}[\sigma] \star C \downarrow F' : P}$$

Figure 3.10: Call-by-value weak typed evaluation of cDLF term (3/5)

$\boxed{\Delta;\Gamma \vdash M : A \Downarrow \Delta';M' \text{ and } \Delta;\Gamma \vdash F \Downarrow \Delta';F' : P}$ Can. obj. full evaluation

$$\textsc{SMObj}\ \frac{\Delta;\Gamma \vdash M_0 : A \downarrow M_1 \qquad \Delta;\Gamma \vdash M_1 : A \uparrow \Delta';M_2}{\Delta;\Gamma \vdash M_0 : A \Downarrow \Delta';M_2}$$

$$\textsc{SFAtom}\ \frac{\Delta;\Gamma \vdash F_0 \downarrow F_1 : P \qquad \Delta;\Gamma \vdash F_1 \uparrow \Delta';F_2 : P}{\Delta;\Gamma \vdash F_0 \Downarrow \Delta';F_2 : P}$$

$\boxed{\Delta;\Gamma \vdash F \uparrow \Delta';F' : P}$ Atomic object readback

$$\textsc{RFVar}\ \frac{x : A \in \Gamma \qquad \Delta;\Gamma;A \vdash S \uparrow \Delta';S' : P}{\Delta;\Gamma \vdash x(S) \uparrow \Delta';x(S') : P}$$

$$\textsc{RFConst}\ \frac{\mathsf{c} : A \in \Sigma \qquad \Delta;\Gamma;A \vdash S \uparrow \Delta';S' : P \qquad \mathrm{stren}_{\Gamma}(\mathsf{c}(S')) = \Gamma' \qquad \mathtt{X} \notin \mathrm{dom}(\Delta')}{\Delta;\Gamma \vdash \mathsf{c}(S) \uparrow (\Delta', \mathtt{X}[\Gamma'] : P = \mathsf{c}(S)); \mathtt{X}[\mathrm{id}_{\Gamma'}] : P}$$

$$\textsc{RFInv}\ \frac{f : A = \text{“}T\text{”} \in \Sigma \qquad \Delta;\Gamma;(A)_n \vdash S \uparrow \Delta';S' : P}{\Delta;\Gamma \vdash f^n(S) \uparrow \Delta';\pi_n(A) \star S' : P}$$

$$\textsc{RFMeta}\ \frac{\mathtt{X}[\Gamma'] : P = \mathsf{c}(S) \in \Delta \qquad \Delta;\Gamma \vdash \sigma : \Gamma' \uparrow \Delta';\sigma'}{\Delta;\Gamma \vdash \mathtt{X}[\sigma] \uparrow \Delta';\mathtt{X}[\sigma'] : P[\sigma']}$$

Figure 3.11: Call-by-value full typed evaluation of cDLF (4/5)

$\boxed{\Delta; \Gamma \vdash M : A \uparrow \Delta'; M'}$ Canonical object readback

$$\textsc{RMLam}\ \frac{\Delta; \Gamma, x : A \vdash M : B \Downarrow \Delta'; M'}{\Delta; \Gamma \vdash \lambda x.\, M : (\Pi x : A.\, B) \uparrow \Delta'; \lambda x.\, M'} \qquad \textsc{RMAtom}\ \frac{\Delta; \Gamma \vdash F \uparrow \Delta'; P : F'}{\Delta; \Gamma \vdash F : P \uparrow \Delta'; F'}$$

$\boxed{\Delta; \Gamma; A \vdash S \uparrow \Delta'; S' : P}$ Spine readback

$$\textsc{RSNil}\ \frac{}{\Delta; \Gamma; P \vdash \cdot \uparrow \Delta; \cdot : P}$$

$$\textsc{RSCons}\ \frac{\Delta_0; \Gamma \vdash M : A \uparrow \Delta_1; M' \qquad \Delta_1; \Gamma; B[x/|M'|] \vdash S \uparrow \Delta_2; S' : P}{\Delta_0; \Gamma; \Pi x : A.\, B \vdash (M, S) \uparrow \Delta_2; (M', S') : P}$$

$\boxed{\Delta; \Gamma \vdash \sigma : \Gamma' \uparrow \Delta'; \sigma'}$ Substitution readback

$$\textsc{R}\sigma\textsc{Nil}\ \frac{}{\Delta; \Gamma \vdash \cdot : \cdot \uparrow \Delta; \cdot} \qquad \textsc{R}\sigma\textsc{Cons}\ \frac{\Delta_0; \Gamma \vdash \sigma : \Gamma' \uparrow \Delta_1; \sigma' \qquad \Delta_1; \Gamma \vdash M : A[|\sigma'|] \uparrow \Delta_2; M'}{\Delta_0; \Gamma \vdash (\sigma, x/M) : (\Gamma', x : A) \uparrow \Delta_2; (\sigma', x/M')}$$

Figure 3.12: Call-by-value full typed evaluation of cDLF (5/5)

tion, together with its context Δ and environment Γ, and traverses it a second time to find λ-abstraction bodies to evaluate (Figures 3.11 and 3.12). Once a subterm is an actual *value*, i.e., once it does not contain functions anymore, it is sliced (rule RFConst), its inverses erased (rule RFInv), and the procedure eventually returns a context Δ' and a *residual* F'. All rules are similar to their DLF equivalent, except that:

- All judgments take and return a context, and also return a residual of slicing; context are threaded sequentially in the premises of rules RSCons and RσCons; in most rules, the residual is the recomposed input term ...
- ... except for rule RFConst which slices a constant application (this rule is similar to FConst in cSLF), and rule RFInv, which projects out all inverses met. Remember that we only slice constant applications, and not variable application: this is why the residual of RFInv is $x(S')$.
- Rule RFMeta is similar to FMeta: reading back a metavariable does not involve inspecting its body, only its substitution, since its body is already a value.

Like in DLF, this walk must carry the type of the value (infer or check it), so as to pass it to weak evaluation when we meet a λ-abstraction.

Finally, *full evaluation* (Figure 3.11) is the composition of these two phases. Starting from a context Δ, an environment Γ and an atomic object F, they are passed to weak evaluation (rule SFAtom); then the resulting weak value F_1 is read back in the same context Δ and environment Γ. Readback returns a value F_2, sliced in the extended context Δ'.

We did not include judgments for types, kinds and signatures. This is because they are the same as in cSLF (Figure 3.6), and we did not repeat them here: just like DLF was relying on the signature verification of SLF, cDLF types cannot contain function or inverse symbols.

3.3.3 | CASE STUDY

Let us consider again the certifying type checker of Section 2.4.2. We will use it *as is* to implement the small term reuse examples of examples 3.2 and 3.3.

We recall that we declared a signature of terms and derivations:

```
tm : ∗.
tp : ∗.
is : tm → tp → ∗.
```

The type checker *infer* takes a term are returns the "package" of a type and a derivation. This package was declared by type inf:

```
inf : tm → ∗.
ex : Π M : tm. Π A : tp. Π H : is M A. inf M.
infer : Π M : tm. inf M = " ... ".
```

Without modifying the code or the signature of this type checker, we can already use it as an incremental type checker, thanks to the caching mechanism provided by cDLF *and* the use of inverses to represent computations that can be avoided. First, let us take the whole signature defined in Section 2.4.2 (all constants and functions) and slice it. We get back a signature Σ_0 where all applied constants have been replaced by metavariables, and a context Δ_0 defining among others:

```
A[M : tm → tm, A : tp] : tm = lam A (λu. M u).
D[A : tp, B : tp] : tp = arr A B.
J[x : tm, x' : tm] : tm = app x x'.
F[x : tm] : tm = s x.
```

¶ | INITIAL TERM Our first query is to type check the initial $\mathsf{T}_{<:}$ term $(\lambda f\, x.f\, x)\ (\lambda x.\, \mathbf{s}(x))\ (\mathbf{s}(\mathbf{o}))$. This query corresponds to the following atomic object F_0 to be fed to the system for evaluation and slicing:

```
infer (app (app
        (lam (arr nat nat) λf. lam nat λx. app f x)
        (lam nat λx. s x))
      (s o))
```

Note how the encoding of the term is significantly shorter than that of Section 3.2.3. This is because we use an *extrinsic* or *à la Curry* encoding of the language: first, bare $\mathsf{T}_{<:}$ terms are encoded, then derivations typing these terms *a posteriori* are defined as a relation between a (term) and a type. Type annotations appear nonetheless at λ-abstractions: we are not doing "full" type inference in the sense of Pierce and Turner [2000].

Wrapping this object with a call to function *infer* makes the system generate and slice a full typing derivation for it, i.e., an object of type inf X where X refers to the sliced input term. Indeed, we have $\Delta_0; \cdot \vdash F_0 \Downarrow \Delta_1; \mathtt{Y} : \mathsf{nat}$, and Δ_1 is Δ_0 extended with slides:

```
B[f : tm] : tm = lam nat (λx. J[x/f; x'/x]).
C : tp = arr nat nat.
E : is A[M/λx. B[f/x]; A/C] D[A/C; B/C] =
   Lam (λf. B[f/f]) C C (λx. λh. O[x/x; h/h]).
G[x : tm, h : is x nat] : is F[x/x] nat = Sn x h.
H : tm = s o.
I : is H nat = Sub H odd nat SubOdd K.
K : is F[x/o] odd = So o L.
```

```
L : is o even = O.
O[x : tm, h : is x C] : is A[M/λx'. J[x/x; x'/x']; nat] D[A/nat; B/nat] =
  Lam (λx'. J[x/x; x'/x']) nat nat (λx'. λh'. P[x/x'; h/h; x'/x; h'/h']).
P[x : tm, h : is x C, x' : tm, h' : is x' nat] : is J[x/x; x'/x'] nat =
  App x x' nat nat h h'.
Y : inf X = ex X nat Z.
Z : is X nat = ...
```

We eluded some of the slices for concision. We could compute $\downarrow_{\Delta_1} F_0'$ to get the full typing derivation, and verify that it is well-typed with respect to SLF of type inf F_0, but this is actually guaranteed by the metatheory of our system.

¶ FIRST DELTA Now, let us consider the second $\mathsf{T}_{<:}$ term, $(\lambda f.\, \lambda x.\, f\ x)\ (\lambda x.\, \mathbf{s}(\mathbf{s}(x)))\ (\mathbf{s}(\mathbf{o}))$, and see how we can encode the derivations reuse. First, consider subterm $\mathbf{s}(\mathbf{o})$ on the right; since it was in our original term as-is, there must be a slice in Δ_1 of a type convertible to is (s o) nat: there is, it is I. Contrarily to Section 3.2.3, we cannot directly graft it in place of $\mathbf{s}(\mathbf{o})$ in the term, because it would be ill typed: a term is not the same as a derivation anymore, *infer* precisely generates one from the other. Yet, there is a "coercion" from derivations to terms: infer^0 which "projects out" a derivation back to the term it types. In fact it *almost* does: it takes an inf, and not a derivation, to a tm (its type is ΠM : tp. inf $M \to$ tm). In a nutshell, reusing the derivation of $\mathbf{s}(\mathbf{o})$ amounts to replacing sub-object s o in the query by:

```
infer⁰ H (ex H nat I)
```

When evaluating *infer* (infer^0 H (ex H nat I)), the system will first check that the argument of *infer* is well-typed (taking constant time with respect to the size of the shared derivation stored in I) and then apply contraction (rule FEvalInv) and return simply ex H nat I. Note that $\mathbf{s}(\mathbf{o})$ is a small term here, so this reuse is a small gain, however the reuse of a much larger derivation for a hundred-lines program will take the exact same time.

The same pattern applies to the reuse of derivation for subterm $\lambda f.\, \lambda x.\, f\ x$. The computed derivation is stored in metavariable E; as indicated by its type, the associated term and type were represented with sharing by *resp.* A[M/λx. B[f/x]; A/C] and D[A/C; B/C]. In a nutshell, we reuse E by writing:

```
infer⁰ A[M/λx. Bλ_. ; A/C] (ex A[M/λx. B[f/x]; A/C] D[A/C; B/C] E)
```

¶ REUSING OPEN DERIVATIONS A harder reuse to express is the one of the internal $\mathbf{s}(x)$ subterm, because it is an open term. The corresponding derivation is G and is parameterized by both a term x and a derivation h stating that it has type nat; under these parameters, it states that term F[x/x] (i.e., s x) has type nat. How should we instantiate the parameter h? The answer is to use *infer* x: when the type

checker will traverse the λ-abstraction binding x, it will substitute it with $\mathit{infer}^0\ x\ h$' where h' is the appropriate derivation. Object $\mathit{infer}\ x$ cannot directly instantiate the parameter h however since it is a "package" of type $\mathsf{inf}\ x$ and not $\mathsf{is}\ x\ \mathsf{nat}$; we must open the package and project out the derivation part. An object of type $\mathsf{inf}\ M$ is necessarily constructed by application of the ex constant. Let us introduce an auxiliary function that destructs an inf into the derivation it contains:

```
prj : ΠM : tm. ΠA : tp. inf M → is M A = " fun _ _ i →
  let « ex "_" "_" "h" » = i in h".
```

In a nutshell, the reuse of the derivation G for $\mathbf{s}(x)$ in an environment where x is a variable is thus expressed by the object M:

```
infer⁰ F[x/x] (ex F[x/x] nat G[x/x; h/prj x nat (infer x)])
```

Note the inner call to *infer*: it is the crucial point of this example. The object $\mathit{infer}\ M$ is well-typed in an environment where x : tm. But if we evaluate as it is, it will fail: the inner call to *infer* is made on a free variable. If on the contrary we evaluate $\mathit{infer}\,(\mathsf{lam}\ \lambda x.\,M)$, the first step will be to go under the λ-abstraction and substitute x by $\mathit{infer}^0\ x\ h$ (in an environment where x : tm, h : $\mathsf{is}\ x\ \mathsf{nat}$): this way the inner call to *infer* can be contracted to just h.

Finally, our second query is object F_1:

```
infer (app (app
        (infer⁰ A[M/λx.Bλ_x.; A/C] (ex A[M/λx.B[f/x]; A/C] D[A/C; B/C] E))
        (lam nat λx.s (infer⁰ F[x/x]
                              (ex F[x/x] nat G[x/x; h/prj x nat (infer x)])))
      (infer⁰ H (ex H nat I)))
```

Note that even though this term is large, its size does not depend on the size of the derivations referred by E, H and I we avoided to recompute; it depends however on the number of free variables referred to in these metavariables, since we have to explicitly mention their substitution. Note also that many arguments are redundant here: for instance, in the frequently used subobject $\mathit{infer}^0\ M_1\ (\mathsf{ex}\ M_1'\ M_2\ M_3)$, M_1 is always equal to M_1' and M_2 can be deduced by just reading off the type of M_3. This is because our language is *explicit*: even though it would be possible, no inference of *implicit arguments* is performed. Cleared from all arguments that could be inferred, our query could be simply:

```
infer (app (app (infer⁰ E) (lam nat λx.s (infer⁰ G[h/infer x]))) (infer⁰ I))
```

which is far more readable and easier to generate.

Once evaluated, F_1 gives rise to a sliced derivation for our modified program, where some of the new derivation slices point to the first derivation we built. Because we built, among others, a derivation for $\mathbf{s}(\mathbf{s}(x))$, the returned context Δ_2 contains new slices:

```
M[x : tm] : tm = s F[x/x].
N[x : tm, h : is x nat] : is F[x/F[x/x]] nat = Sn F[x/x] G[x/x; h/h].
```

¶ SECOND DELTA Finally, we can easily type term $(\lambda f\, x.f\ x)\ (\lambda x.\ \mathbf{s}(\mathbf{s}(\mathbf{o})))\ (\mathbf{s}(\mathbf{o}))$, by instantiating term M with o and derivation N with *infer* o. There is a slight twist however: subtyping. The generated type of o will be even, which cannot directly be grafted in derivation N which awaits a nat parameter; however we can easily coerce one to the other with function *subm*, which tries to coerce a typing derivation from one type to a subtype. This gives the following query F_2:

```
infer (app (app
        (infer^0 A[M/λx. B[f/x]; A/C] (ex A[M/λx. B[f/x]; A/C] D[A/C; B/C] E))
        (lam nat λx. (infer^0 M[x/o]
            (ex M[x/o] nat N[x/o; h/subm o even nat (prj o even (infer o))])))))
      (infer^0 H (ex H nat I)))
```

Note here that the derivation nodes for the two internal $\mathbf{s}()$ constructors where not regenerated, we instantiated the parametric slice M. If we were to use a purely first-order representation of derivations and terms, as it is the case with *memoization*, this would be impossible and we would need to regenerate the derivations for $\mathbf{s}()$ nodes since their subterm went from x to $\mathbf{o}$.

Again, the skeptical reader can evaluate it in the empty environment, get a context Δ_3 and a residual F_2', and verify that the object $\downarrow_{\Delta_3} F_2'$ encodes a valid derivation for the $\mathsf{T}_{<:}$ term $(\lambda f\, x.f\ x)\ (\lambda x.\ \mathbf{s}(\mathbf{s}(\mathbf{o})))\ (\mathbf{s}(\mathbf{o}))$.

3.3.4 METATHEORY

We finish this section by a quick look at the properties of cDLF. Since the major work has been done already on its direct parents DLF and cSLF, we only outline here its relation to them.

First, we sketch that it has the same operational behaviour as DLF, in the sense that they evaluate equal objects to equal objects, modulo the substitution of metavariables by their definitions: this is *soundness*.

Lemma 3.2 (Soundness)**.** *Let* $\cdot \vdash \Sigma_0$ sig $\Rightarrow \Delta_0; \Sigma$ *and* $\Delta_0 \vdash \Delta$ ctx *and* $\Delta \vdash \Gamma$ env. *Then:*

1. *If* $\Delta; \Gamma \vdash_\Sigma M : A \downarrow M'$ *then* $\downarrow_\Delta \Gamma \vdash_{\Sigma_0} (\downarrow_\Delta M) : (\downarrow_\Delta A) \downarrow (\downarrow_\Delta M')$;
2. *If* $\Delta; \Gamma \vdash_\Sigma F \downarrow F' : P'$ *then* $\downarrow_\Delta \Gamma \vdash_{\Sigma_0} (\downarrow_\Delta F) \downarrow (\downarrow_\Delta F') : (\downarrow_\Delta P')$;
3. *If* $\Delta; \Gamma; A \vdash_\Sigma S \downarrow S' : P'$ *then* $\downarrow_\Delta \Gamma; \downarrow_\Delta A \vdash_{\Sigma_0} (\downarrow_\Delta S) \downarrow (\downarrow_\Delta S') : (\downarrow_\Delta P')$;

Theorem 3.5 (Soundness)**.** *Let* $\cdot \vdash \Sigma_0$ sig $\Rightarrow \Delta_0; \Sigma$ *and* $\Delta_0 \vdash \Delta$ ctx *and* $\Delta \vdash \Gamma$ env. *Then:*

1. *If* $\Delta; \Gamma \vdash_\Sigma M : A \Downarrow \Delta'; M'$ *then* $\downarrow_\Delta \Gamma \vdash_{\Sigma_0} (\downarrow_\Delta M) : (\downarrow_\Delta A) \Downarrow (\downarrow_{\Delta'} M')$;
2. *If* $\Delta; \Gamma \vdash_\Sigma M : A \uparrow \Delta'; M'$ *then* $\downarrow_\Delta \Gamma \vdash_{\Sigma_0} (\downarrow_\Delta M) : (\downarrow_\Delta A) \uparrow (\downarrow_{\Delta'} M')$;
3. *If* $\Delta; \Gamma \vdash_\Sigma F \uparrow \Delta'; F' : P'$ *then* $\downarrow_\Delta \Gamma \vdash_{\Sigma_0} (\downarrow_\Delta F) \uparrow (\downarrow_{\Delta'} F') : (\downarrow_{\Delta'} P')$;
4. *If* $\Delta; \Gamma; A \vdash_\Sigma S \uparrow \Delta'; S' : P'$ *then* $\downarrow_\Delta \Gamma; (\downarrow_\Delta A) \vdash_{\Sigma_0} (\downarrow_\Delta S) \uparrow (\downarrow_{\Delta'} S') : (\downarrow_{\Delta'} P')$.

Corollary 3.2. *If* $\Delta; \Gamma \vdash F \Downarrow \Delta'; F' : P'$ *then* $\downarrow_\Delta \Gamma \vdash \downarrow_\Delta F \Downarrow (\downarrow_{\Delta'} F') : (\downarrow_{\Delta'} P')$.

Proof. By rule SFAtom and the previous theorem. □

With what we know about DLF, we can conclude that all cDLF residuals are well-typed with respect to SLF.

Corollary 3.3. *If* $\Delta; \Gamma \vdash F \Downarrow \Delta'; F' : P'$ *then* $\downarrow_\Delta \Gamma \vdash \downarrow_{\Delta'} F' \Rightarrow \downarrow_{\Delta'} P'$.

Proof. By composing theorem 2.1, and the previous corollary. □

Secondly, it is easy to see that cDLF has also the same property as cSLF: whatever its input term is, its output term will be sliced, i.e., there will be a metavariable name assigned to all rule application:

Theorem 3.6 (Sliced form)**.** *Let* Δ, Σ *and* Γ *be sliced. Then:*

1. *If* $\Delta; \Gamma \vdash_\Sigma F \uparrow \Delta'; F' : P'$, *then* Δ', F' *and* P' *are sliced.*
2. *If* $\Delta; \Gamma \vdash_\Sigma M : A \uparrow \Delta'; M'$ *and* A *sliced then* Δ' *and* M' *are sliced.*
3. *If* $\Delta; \Gamma; A \vdash_\Sigma S \uparrow \Delta'; S' : P'$ *and* A *sliced then* Δ', S' *and* P' *are sliced.*
4. *If* $\Delta; \Gamma \vdash_\Sigma \sigma : \Gamma' \uparrow \Delta'; \sigma'$ *and* Γ' *sliced then* Δ' *and* σ' *are sliced.*

Proof. By easy mutual induction over the derivation. □

3.4 | IMPLEMENTATION HIGHLIGHTS

The Gasp framework is the implementation of cDLF in OCaml. We could call it a *library*, since it is not a fully-fledged program but a set of primitives that can be used in other OCaml programs to manipulate certificates. We could also call it an embedded *Domain Specific Language*, since it is only a *shallow embedding* of the cDLF language on top of OCaml. In particular, we will discover in this section that the computational layer presented above is actually OCaml itself, used in a controlled way thanks to carefully chosen abstractions.

¶ | ARCHITECTURE The implementation consists of three different modules. First, the CLF module defines the abstract syntax tree of the *surface language*, just as it is parsed. There, there is no distinction between objects, types and kinds: each of these terms is parsed and has type term. At this level, concrete names typed by the user are represented as strings; also, application is *binary*.

Secondly, the cDLF module defines the syntax of cDLF objects, types etc. but not its computational language (terms, atomic terms, patterns and branches): these will be handled by OCaml itself. At this level, variables are represented by *De Bruijn indices*, application is n-ary, and metavariables are represented by strings which are hash keys to the pair of their environment and their definition. This way, we ensure that there will never be two distinct metavariables for the same slice. In this module is also defined a partial function from CLF terms to either objects, types or kinds, and its inverses, mapping back a cDLF term to a CLF one. We call this operation *stratification* (or *unstratificaton* for its inverse) since it determines the "level" of a given term.

The *kernel* module defines the typed, slicing evaluation algorithm described above. It defines the type of a *repository* repo to be the triplet of a signature sign, a context ctx and a residual atomic object atom: it packs together all input and output of the evaluation process. The module exports only four functions:

$$\mathit{init} : \mathsf{sign} \to \mathsf{repo} \tag{3.35}$$

$$\mathit{commit} : \mathsf{repo} \to \mathsf{env} \to \mathsf{atom} \to \mathsf{repo} \tag{3.36}$$

$$\mathit{checkout} : \mathsf{repo} \to \mathsf{atom} \to \mathsf{atom} \tag{3.37}$$

$$\mathit{eval} : \mathsf{repo} \to \mathsf{env} \to \mathsf{atom} \to \mathsf{atom} \tag{3.38}$$

Function *init* takes a signature, checks and slices it into an initial repository formed of the sliced signature, the context of these slices and a dummy residual (this initial repository contains no object yet). The *commit* function takes an atomic object, possibly containing function and inverse symbols and metavariables, and evaluates it. The returned repository has the same signature as the input one, only a context enlarged with new slices and a residual that is replaced by the residual resulting from slicing. The *checkout* function implement the checkout operation of cSLF: it takes an atomic object containing no functions to evaluate, and strips it off of all its metavariables. If the atomic object is a residual (i.e., if it was well-typed with respect to the repository's context and signature) then this operation returns an object that is well-typed with respect to SLF. Finally, *eval* implements weak cDLF evaluation. It is used in the code of defined functions to evaluate recursively objects before pattern-matching.

¶ QUOTATIONS AND ANTIQUOTATIONS As is common when implementing a DSL, our implementation relies heavily on the use of *quotations* and *antiquotations*. We use the CamlP4[6] quotation facility to define the concrete syntax of the embedded language of terms and to parse. CamlP4 is a multi-purpose tool that is shipped with OCaml, and that can be viewed as a set of independent components, among which:

[6] `http://brion.inria.fr/gallium/index.php/Camlp4`

- an LL(1) parser, taking a string and the representation of an α-grammar (i.e., a grammar which semantic actions have type α), and returning an α;
- the abstract syntax e of OCaml itself, and an e-grammar to parse OCaml programs...
- ... extended with one parametric construct for *quotations* «^{L} X » where L is a special label string identifying the particular language in which the string X is written;
- a table associating an e-grammar to each label L, so that each quotation «^{L} X » is parsed by the associated grammar, and turned into an OCaml expression which inserted in place of the quotation;

The OCaml compiler itself has an option to use CamlP4 instead of its own internal parser. Note that with this simple mechanism, one is free to declare quotations featuring a particular syntactic rule (written here "e") which semantic action is to parse string e with e-grammar, i.e., OCaml code itself; these are *antiquotations*. In turn, antiquotations, like any OCaml expression, can themselves contain quotations, which can contain antiquotations...

Gasp uses three different quotations: signatures «^{sign} X », environments «^{env} X » and terms « X » (without label). Each of these quotations is replaced by a value of OCaml type respectively sign, env and term which represent the abstract syntax of CLF.

Example 3.9. The OCaml code

```
fun t → « λx. λy. "t" x y »
```

has type term → term and is preprocessed into

```
fun t → Lam ("x", Lam ("y", App (App (t, Var "x"), Var "y")))
```

¶ THE COMPUTATIONAL LEVEL The abstract syntax tree of CLF defines notably signatures the following way:

```
type func = (term → term) → term list → term
type sign = (string × term × func option) list
```

Each element of the signature contains a name identifying the constant being defined, a term representing the type or kind of this constant, and optionally an OCaml function. This function will serve as the code associated to this function symbol: each time the kernel has to evaluate an application $f\,(S)$ (where S is already evaluated), it unstratifies the cSLF spine S into a CLF list of term, retrieves this function's code `f` : func and apply the unstratified terms to this function. The higher-order function in type func serves to call the evaluator recursively in the code of the functions when pattern-matching. Finally when the call to f returns, the result (a term) is stratified and passed back to the evaluator.

Example 3.10. The following is a value of type sign:

```
let s = «sign
  nat : ∗. o : nat. s : nat → nat.
  deux : nat = "fun eval [] → « s (s o) »".
  plus : nat → nat → nat = "fun eval [m;n] → match eval m with
    | « o » → n | « s "m" » → « s (plus "m" "n")".
»
```

which with some syntactic sugar is transformed into the syntax we saw up until now (function `eval` is made implicit, and the OCaml functions are currified by counting the number of arguments of the type of the function symbol).

¶ LOCALLY NAMED TERMS It is important to avoid variable capture when constructing a term in the code of a function. For instance, the (syntactically sugared) function

```
f : tm → tm = "fun t → « lam λx."t" »".
```

wraps a λ-term into a λ-abstraction which variable should not be used. A naive handling of names in CLF would allow term lam $\lambda x.f\ x$ to evaluate to lam $\lambda x.$ lam $\lambda x.x$, where the expected result is of course lam $\lambda x.$ lam $\lambda x'.x$.

Our solution is to distinguish two kinds of variables in CLF terms: named variables (e.g., Named "x"), as entered by the user, for the bound variables, and De Bruijn indices (e.g., Free 3) for free variables coming from unstratified terms. For instance, when the triggering the evaluation of object $f\ x$, the argument x (represented in cDLF as the de Bruijn index Var 0, since x is the 0-th binding in the current environment) is first unstratified into the term Free 0; the function is then called, and its return is the OCaml value « lam λx."Free 0" » of type term, or, preprocessing the quotation, App (Id "lam", Lam ("x", Free 0)). As we see, the free variable is "protected" from internal binders because it does not belong to the same namespace. Then, this value is restratified to a cDLF object OApp (OConst "lam", [OLam ("x", Var 1)]. The de Bruijn index is incremented since we traversed a λ-abstraction. Finally, the whole reconstructed term reads « lam $\lambda x.$ lam $\lambda x'.x$ » as expected.

Surprisingly, the exact opposite of this technique is very common when formalizing languages with binders, the *locally nameless* encoding [Aydemir et al., 2008]: there, free variables are represented with *names* and bound ones with *De Bruijn indices*.

¶ FROM NON-CANONICAL TERMS TO CANONICAL OBJECTS When we presented SLF in Chapter 1, we described its canonical, spine-form syntax: a syntax in which no β-redex can be written, and application is n-ary. The careful reader will have noted in Chapter 2 that writing the *eval* function (example 2.4) actually

necessitated a non-canonical, binary application to implement substitution by a β-reduction in LF. In substance, this example was written:

```
eval : tm → vl = " fun m → match m with
  | « lam "p" » → « vlam "p" »
  | « app "m" "n" » → let « vlam "p" » = « eval "m" » in « eval ("p" "n") »
```

In the second case, computational variable `p` is bound to an LF λ-abstraction (since it has an arrow type); yet, in this branch, it is applied to `n`. We eluded the problem by stating that an application $(\lambda x.\, M)\ N$ was a notation for the *cut* operation $(\lambda x.\, M) \star N$. Even if `p` was itself a partial application, this would require to add the last argument `n` at the end of its spine.

The distinction between a *surface language* CLF and an internal representation cDLF allows us to implement this properly. A CLF term will be non-canonical (allowing β-redexes) and will have the usual, left-associative binary application. This way, the previous term *eval* ("p" "n") is preprocessed simply as App (Id "eval", App (p, n)). The internal representation however is canonical and spine-form; the translation from surface to internal representation, i.e., stratification, thus turns the applications upside down, and if needed performs substitutions: the corresponding cDLF object is the OCaml expression OApp (OConst "eval", [cut (strato p) (strato n)]) where `strato` and `cut` respectively stratify a term into an object and perform a series of reductions.

3.5 | RELATED AND FURTHER WORK

The Gasp framework can be seen as a core language for verifying incrementally the well-typing of a program. Through the *Curry-Howard isomorphism*, in can equally well serve as an incremental verifier for proofs. It relies on five main aspects:

- already-checked programs are stored along with their typing derivation in the LF metalanguage, which makes typing annotation explicit;
- these typing derivation are *sliced* so that each subderivation has a name;
- a program *delta* is a program where already checked subterms refer to stored subderivation
- the type checker generates a derivation for each delta, reusing its shared subderivations;
- this reuse is expressed by function inverses, and *contraction* occurs when the type checker reuses a shared subderivation.

It draws ideas from both the DLF language of Chapter 2 for manipulating certificates, and from recent developments in contextual type theory.

¶ | CONTEXTUAL TYPES *Contextual Modal Type Theory* [Nanevski et al., 2008] extends LF with metavariables, which are placeholders referring to open LF objects. This development has interesting practical applications for the implementation of proof assistants, as mentioned earlier, but also in staged computation; it is based on the notion of contextual type, which can be viewed through the Curry-Howard isomorphism as a generalization of a modal logic of necessity [Pfenning and Davies, 2001]. In CMTT, we can for instance express the type of a function taking an argument that is necessarily closed, and returns a potentially open object: $[\cdot]A \to B$. In the body of this function, its argument will be a metavariable.

Our usage of this idea in cSLF is limited to the particular case where all metavariable definitions are known: they are not placeholders, but only notational definitions which body is known to be well-typed. This is why we have no need for the contextual operator $[\Gamma]A$ deep in our types: only defined metavariables in a context Δ have a toplevel contextual type. Since they are only definitions, a cSLF object is not canonical *per se*, and can be "expanded" (checked out as we said); we then lose the potential sharing induced by these definitions. On the contrary, CMTT is canonical: all metavariables in an object are undefined, and a hereditary meta-substitution operation normalizes objects on the fly.

As we discussed already, our deltas are inherently very verbose, especially at the contact point between a program and a reused derivation. It would be interesting to reduce this verbosity by making some arguments implicit, and implement *type reconstruction* as in Pientka [2013]; this would make the generation of deltas much easier. Notably, type reconstruction in presence of metavariables is still mostly unexplored: we conjecture that it would allow to omit some bindings in the substitutions attached to metavariables.

¶ | SLICING An object in sliced form is reminiscent of what is called *monadic form* or *A-normal form* in compilation [Flanagan et al., 1993, Hatcliff and Danvy, 1994, Danvy, 2003], in that every applicative term is named (here by metavariables), such that the resulting applications are all "flattened". Monadic translations are designed to explicit the order of evaluation; we however do not care about the order in which metavariables are registered in a context since our terms are already in canonical form; also, we give names only to rule applications (constant application) and not computations. The process of slicing an object also recalls another program transformation, *supercombinator conversion* [Hughes, 1982, Peyton Jones, 1985]: applicative, open pieces of programs that were buried deep inside a large program are lifted at toplevel and named in a large map; such a combinator is abstracted over its free variables, and can be referred to in other supercombinator definitions, provided that these calls instantiate all the free variables. Supercombinators are here to be compared with metavariables, and their (total) application to a metavariable

together with its instantiating substitution: slicing and supercombinator conversion produce a series of closed (with respect to variables) terms (but not closed with respect to metavariables). Once again, the role of this transformation is to take a functional program and put it in a form that is easier to interpret (by the G-machine [Peyton Jones, 1987]); on the contrary, the role of slicing is only to "cache" the typing of a large, canonical proof object.

¶ | INCREMENTAL COMPUTATION The problem of incremental type type checking is of course an instance of incremental computation, to what it must be compared. The literature on incremental computation is dominated by studies of *memoization*, also known a *function caching* or *tabling* [Michie, 1968, Pugh and Teitelbaum, 1989, Acar et al., 2003], and *dependency graphs à la* make [Paige and Koenig, 1982, Yellin and Strom, 1991]. These are techniques to *derive* an incremental program from a batch one.

As we discussed in the first section of this chapter, memoization is an automatic technique: a table stores input-output pairs of a function, and the recognition of an already-computed output is automated by looking up in this table. In particular, if the input is a large term, output will be reused only for *syntactically equal* subterms. Our method, on the contrary, is manual: the input of our incremental type checker explicitly mentions what input should be reused, found in a table of slices: the context. But in conjunction with the higher-order nature of the input and output, this allows to have a more fine-grained notion of reuse: not only can we reuse derivations for equal subterms, but we can also instantiate parametric slices of derivations, as witnessed by our running example. An interesting further study would be to devise an algorithm taking a context and a large term with no metavariables, and turning it into a "minimal" delta reusing as much slices as possible. In the general case, this is undecidable because it reduces to higher-order unification, but we might find reasonable compromises.

¶ | ADAPTIVE FUNCTIONAL PROGRAMMING Acar et al. [2002], Acar [2005], Acar et al. [2006b] are interested in combining memoization and dependency graphs in functional programming, and call the combination *adaptive functional programming*. They show that there exists an elegant duality between these two, describe a small ML library for turning a batch program into an incremental one by explicitly stating points in the program where computation can be resumed; this technique amounts to invert the control flow of the program when the input data is changed. It is, just as ours, explicit: the user has to explicitly point to the piece of input that changed (by mutating it imperatively), and trigger the propagation of this change. This library has been ported to Haskell by Carlsson [2002] by using monads and the lazyness of the language. The combination of the two techniques turns out to be

experimentally quite effective [Acar et al., 2006a].

A type checker written with this library could be turned into an adaptive one, but it would differ from ours by several aspects. First, using a purely first-order representation of programs and environments would hide some possible result caching that we let the metalanguage handle. For instance, if a piece of term is moved from one place to another, there is not reason for this type checker to reuse the its computed type in a compatible, but not syntactically equal environment. Secondly, substituting a piece of term for a variable in a large term, or α-renaming a symbol is an atomic operation in our framework; it would take as many change propagation as there are variables in an adaptive type checker.

¶ | STRUCTURED EDITOR The most direct application of our system is to build structured text editors, or Integrated Development Environments, for a particular programming language. It could reflect in real time the impact of the programmer's changes on the well-typing of the program, as well as display useful typing information or support type-directed programming. This is not a new idea: the Cornell Synthesizer [Teitelbaum and Reps, 1981, Reps et al., 1983, Reps and Teitelbaum, 1984] was such an editor: the language designer would describe the syntax and semantics of the language thanks to an *attribute grammar* [Knuth, 1968], and the system would generate a structured editor for it supporting incremental syntax and type checking. Unfortunately, we believe that if attribute grammars were a convenient notation for simple language semantics, it simply does not scale to contemporary language where type inference is much conveniently expressed by a fully-fledged functional program.

If Gasp is a core language for expressing type checking procedures and program deltas, there remains to build a front end for it to become a typing-aware editor. Keeping in memory the full typing derivation of a program would allow this editor to provide type information at all point, and refactoring commands like α-renaming could be easily automatized; the difficulty would be to translate user commands (insertions, deletion, copy-paste of text) into deltas. Incremental parsing would also be an interesting challenge, that we could base on recent work by Bernardy [2009].

¶ | VERSION CONTROL SYSTEM Another interesting application of Gasp is to build a typed *Version Control System*. A VCS (e.g., CVS, Subversion, Git) is a database that stores sequences of versions of a document (usually the source code of a large program), and gives the ability to record changes, rollback on previous modifications, merge changes coming from multiple sources... Yet, their view on source code is purely textual: they have absolutely no knowledge over the syntax, even less over the semantics of the programs they store. This results in numerous ad-hoc heuristics in their conception, and numerous : for instance, Git has several *merge* heuristics

based on the indentation of the source [Baudiš, 2008]. We believe that embedding syntactic but also typing information in these tools would make them much more precise and more secure: they could for instance base their merge choice on typing, and ensure that a merge is always well-typed.

Actually, the idea of incrementality by sharing subterms was inspired to us by the way Git stores files and directory structure in its *object database*.[7] The careful reader will have noted that most of the vocabulary surrounding cDLF comes from this inspiration (*commit*, *checkout*, *pull*).

[7] see `http://git-scm.com/book/en/Git-Internals-Git-Objects`

4 | FROM NATURAL DEDUCTION TO THE SEQUENT CALCULUS: A FUNCTIONAL CORRESPONDENCE

¶ | ABSTRACT How is the *sequent calculus* related to *natural deduction*? If they are equivalent, why are there "more" sequent calculus proofs than there are natural deductions? In this chapter, we analyze the connections between these two formalisms, not with the eyes of a proof theorist, as it is usually the case, but with the eyes of a functional programmer. We will see that, surprisingly, the program that verifies sequent calculus proofs is close to a *compiled* version of the program that verifies natural deductions. This will bring us to considerations about *focusing*, which is usually considered a property solely of sequent calculi.

4.1 | MOTIVATIONS

A typical introductory course to proof theory (like for instance Girard et al. [1989]) starts by presenting the two calculi introduced by Gentzen [1935]: first intuitionistic natural deduction (NJ for short, Figure 4.1), since it provides an intuitive view on how to define the meaning of each logical connectives by introduction and elimination, then intuitionistic sequent calculus (LJ for short, Figure 4.2), as an equivalent refinement of the latter, making it easier to search for proofs of a proposition. Unfortunately, one could remark, this finer-grained calculus reveals to leave more "space" to alternative proofs than the first: several sequent calculus proofs can have the same natural deduction equivalent, which leads to more unwanted non-determinism in proof search.

It is easy to show that both calculi are equivalent in terms of provability, an "extensional" property, but what is the "intensional" relationship between them? Why do both exist at all? The answer usually goes through the analysis of *normal* proofs. Normal NJ proofs have the property that they admit a natural *bidirectional*

$$\textsc{ConjI}\ \frac{\vdash A \quad \vdash B}{\vdash A \wedge B} \qquad \textsc{ConjE1}\ \frac{\vdash A \wedge B}{\vdash A} \qquad \textsc{ConjE2}\ \frac{\vdash A \wedge B}{\vdash B} \qquad \textsc{ImpI}\ \frac{\begin{array}{c}[\vdash A]\\ \vdots \\ \vdash B\end{array}}{\vdash A \supset B}$$

$$\textsc{ImpE}\ \frac{\vdash A \supset B \quad \vdash A}{\vdash B} \qquad \textsc{DisjI1}\ \frac{\vdash A}{\vdash A \vee B} \qquad \textsc{DisjI2}\ \frac{\vdash B}{\vdash A \vee B}$$

$$\textsc{DisjE}\ \frac{\vdash A \vee B \qquad \begin{array}{c}[\vdash A]\\ \vdots \\ \vdash C\end{array} \qquad \begin{array}{c}[\vdash B]\\ \vdots \\ \vdash C\end{array}}{\vdash C}$$

Figure 4.1: Propositional natural deduction NJ

Structural rules

$$\textsc{Init}\ \frac{}{\Gamma, A \longrightarrow A} \qquad \textsc{Permut}\ \frac{\Gamma, A, B \longrightarrow C}{\Gamma, B, A \longrightarrow C} \qquad \textsc{Contract}\ \frac{\Gamma, A, A \longrightarrow C}{\Gamma, A \longrightarrow C}$$

Right rules

$$\textsc{ImpR}\ \frac{\Gamma, A \longrightarrow B}{\Gamma \longrightarrow A \supset B} \qquad \textsc{ConjR}\ \frac{\Gamma \longrightarrow A \quad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B} \qquad \textsc{DisjR1}\ \frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow A \vee B} \qquad \textsc{DisjR2}\ \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow A \vee B}$$

Left rules

$$\textsc{ImpL}\ \frac{\Gamma \longrightarrow A \quad \Gamma, B \longrightarrow C}{\Gamma, A \supset B \longrightarrow C} \qquad \textsc{ConjL1}\ \frac{\Gamma, A \longrightarrow C}{\Gamma, A \wedge B \longrightarrow C} \qquad \textsc{ConjL2}\ \frac{\Gamma, B \longrightarrow C}{\Gamma, A \wedge B \longrightarrow C}$$

$$\textsc{DisjL}\ \frac{\Gamma, A \longrightarrow C \quad \Gamma, B \longrightarrow C}{\Gamma, A \vee B \longrightarrow C}$$

Figure 4.2: Propositional sequent calculus

reading: introduction rules are read bottom-up, from the goal sequent until it is fully decomposed in all its atomic parts; elimination rules are read top-down, from the hypothesis down to *atoms*. Then, every judgment is a subterm of the previous one (the *subformula property*). Both reading directions meet in the middle of the proof, at atoms:

$$\dfrac{\begin{array}{c}[\Gamma] \\ \downarrow \mathrm{E} \end{array}}{\begin{array}{c} \vdash p \\ \uparrow \mathrm{I} \\ \vdash A \end{array}}$$

But this obliges the reader (or the proof search procedure) to read (or generate proof trees) in two directions. The sequent calculus is then presented as a response to this cumbersome bidirectionality, by turning all elimination subproofs *upside down*: then, every judgment is a subterm of the judgment directly *below*. Introductions are renamed "right rules", upside-down eliminations become "left rule" and operate directly on the environment Γ, and the meeting of these two becomes the identity rule Id.

$$\dfrac{\begin{array}{cc} \uparrow \mathrm{L} & \uparrow \mathrm{R} \\ \Gamma & \vdash A \end{array}}{\mathrm{Id}}$$

The aim of this chapter is to turn this *intuition* into a *system*, using off-the-shelf tools from functional programming theory (in the style of Ager et al. [2003]), and reasoning on the verification procedures for proofs: the *type checker*. We will see that there is a systematic transformation from a natural deduction-style checker into a particular form of sequent calculus-style checker: LJT, a *focused* LJ. Surprisingly, this transformation is the same as the transformation turning a recursive function into an equivalent iterative function in *accumulator-passing style*. This will lead us to a new completeness proof of LJT, by composition of semantics-preserving program transformation.

4.2 | CANONICAL NJ PROOFS

We are first interested in characterizing canonical NJ proofs. Here, we reconstruct a classical result dating back to Prawitz [1965] and formalized in Sieg and Byrnes [1998]—the *intercalation calculus*—through the looking glass of proof term assignment. This will allow a discussion on *bidirectional type checking*, the starting point of

our transformations.

In the following, we will deal only with a fragment of natural deduction to lighten the development. Our propositions are built out of implication, conjunction, disjunction and atoms:

$$A, B, C \ ::= \ A \supset B \ \Big| \ A \vee B \ \Big| \ A \wedge B \ \Big| \ \mathsf{p}$$

However, we will take great care in not making additional assumptions than the fact that each connective is defined in a "natural deduction style", i.e., by rules of introduction (which conclusion is the connective in question) and elimination (of which one *principal premise* is the connective) in a hypothetical notation (see Section 1.1). The development should then easily be extensible to the full calculus.

4.2.1 | BIDIRECTIONAL READING OF CANONICAL PROOFS

Let us take a concrete example of canonical proof in NJ, and how it reads bidirectionally:

Example 4.1. The *barbara* syllogism admits this normal proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\vdash (\mathsf{p} \supset \mathsf{q})] \quad [\vdash \mathsf{p}]}{\vdash \mathsf{q}}\,\textsc{ImpE} \quad [\vdash (\mathsf{q} \supset \mathsf{r})]}{\vdash \mathsf{r}}\,\textsc{ImpE}}{\vdash \mathsf{p} \supset \mathsf{r}}\,\textsc{ImpI}}{\vdash (\mathsf{q} \supset \mathsf{r}) \supset \mathsf{p} \supset \mathsf{r}}\,\textsc{ImpI}}{\vdash (\mathsf{p} \supset \mathsf{q}) \supset (\mathsf{q} \supset \mathsf{r}) \supset \mathsf{p} \supset \mathsf{r}}\,\textsc{ImpI}$$

First, we introduce all three hypotheses (bottom-up) to end up with the atomic goal $\vdash \mathsf{r}$. This bottom-up phase is over: if we were to continue, we would have to "invent" a new formula since r has no more subformulae. We thus start anew from hypothesis $[\vdash \mathsf{q} \supset \mathsf{r}]$ that can be eliminated, ending up with goal $\vdash \mathsf{q}$. Once again, it is an atom so we stop reading here and start back from hypothesis $[\vdash \mathsf{p} \supset \mathsf{q}]$ which is eliminated, leaving us with $\vdash \mathsf{p}$ to prove, which is directly a hypothesis. It turns out that all four subproofs we just constructed separately can be "glued together" at the atoms p, q, and r.

Note that bidirectional reading is not possible if the proof is non-canonical, i.e., if it features "roundabouts". A roundabout, or *cut*, is a subproof that starts with the introduction of a connective, and which is used in a context where this connective is directly eliminated. In other words, it is a subproof that forms a detour, avoidable by inlining it:

Example 4.2. *Barbara* also admits the non-canonical proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\vdash (q \supset r)] \quad [\vdash q]}{\vdash r}\,\textsc{ImpE}}{\vdash q \supset r}\,\textsc{ImpI} \quad \dfrac{[\vdash (p \supset q)] \quad [\vdash p]}{\vdash q}\,\textsc{ImpE}}{\vdash r}\,\textsc{ImpE}}{\vdash p \supset r}\,\textsc{ImpI}}{\vdash (q \supset r) \supset p \supset r}\,\textsc{ImpI}}{\vdash (p \supset q) \supset (q \supset r) \supset p \supset r}\,\textsc{ImpI}$$

It reads bottom-up until the middle $\vdash r$, then from the leaf e.g., starting with $[\vdash q \supset r]$. However after decomposing the latter, we end up with judgment $\vdash r$: if we continue reading top-down, a new formula $q \supset r$ is "recomposed" that is not a subterm of the last judgment.

4.2.2 | INTERCALATION THROUGH PROOF TERM ASSIGNMENT

In a first approximation, an introduction rule immediately followed by and elimination of the same connective forms a roundabout or *cut* in a natural deduction and blocks the bidirectional reading of that deduction. We know that these can always be avoided (*cut elimination*), but what is the syntactic shape of these normal deductions? The purpose of this section is to derive it.

Let us assign to our non-normal deductions the standard notation of the λ-calculus:

Definition 4.1 (Term assignment)**.** The set of terms assigned to natural deductions are the terms built out of the grammar:

$$\begin{aligned} T \;::=\; & \lambda x.\, T \;\big|\; \mathbf{inl}(T) \;\big|\; \mathbf{inr}(T) \;\big|\; T, T \\ & \big|\; x \;\big|\; \mathbf{case}\ T\ \mathbf{of}\ \langle x.\, T \mid x.\, T \rangle \;\big|\; T\ T \;\big|\; \pi_1(T) \;\big|\; \pi_2(T) \end{aligned}$$

Constructors for introduction rules are on the first line, constructors for eliminations on the second. This syntax comes with the usual typing judgment $\Gamma \vdash T : A$ and rules.

¶ | STRATIFICATION OF PROOF TERMS Now the derivation at the principal premise of each elimination should not be the introduction of the corresponding connective (for instance in $M\ N$, M is the notation for the principal premise of an ImpI rule; it should thus not be $\lambda x.\, M'$). Note that we can strengthen this requirement without loss of generality to "the principal premise of an elimination cannot be *any* introduction": this principal premise judges the connective defined, and any other

introduction is a derivation judging another connective. Assembling them would form an invalid derivation:

$$\dfrac{\vdash \bar{A} \heartsuit \bar{B} \;\diamondsuit\mathrm{I} \qquad \vdots}{\vdash C}\heartsuit\mathrm{E}$$

To ensure this restriction, we *stratify* the syntax of λ-terms into two syntactic classes: the class of all *canonical terms* M°, and the class of terms that are allowed to be principal premise of an elimination, or *atomic terms* R°, i.e., everything except introductions. The first class is included in the second, so we include a coercion from one to another:

$$\begin{aligned} M^\circ, N^\circ &::= \lambda x.M^\circ \mid \mathbf{inl}(M^\circ) \mid \mathbf{inr}(M^\circ) \mid M^\circ, N^\circ \mid R^\circ \\ R^\circ &::= x \mid \mathbf{case}\ R^\circ\ \mathbf{of}\ \langle x.M^\circ \mid x.M^\circ \rangle \mid R^\circ\ N^\circ \mid \pi_1(R^\circ) \mid \pi_2(R^\circ) \end{aligned}$$

Do we have a syntax of normal derivations? Unfortunately not yet.

¶ | THE CASE OF POSITIVE CONNECTIVES If we were to treat only the *negative* fragment of NJ ($\supset$, $\wedge$, $\top$, $\forall$), we would have a normal syntax. But some connectives, the *positive connectives* ($\vee$, $\bot$, $\exists$) behave "pathologically" and oblige to add so-called *commutation rules* (see Girard et al. [1989, chapter 10]). To convince us that M° are not yet notations for normal proofs, let us examine this contorted proof:

$$\dfrac{\dfrac{[\vdash p \vee p] \qquad \dfrac{\dfrac{[\vdash p] \quad [\vdash p]}{\vdash p \wedge p}\,\textsc{ConjI} \qquad \dfrac{[\vdash p] \quad [\vdash p]}{\vdash p \wedge p}\,\textsc{ConjI}}{\vdash p \wedge p}\,\textsc{DisjE}}{\vdash p}\,\textsc{ConjEi}}{\vdash p \vee p \supset p}\,\textsc{ImpI}$$

Surely this is not a canonical proof. First, it does not admit a bidirectional reading, because of the parasitical presence of formula $p \wedge p$ which is not a subformula of the goal nor of the hypotheses. Secondly, there exist an obvious, much shorter proof

$$\dfrac{\dfrac{[\vdash p \vee p] \quad [\vdash p] \quad [\vdash p]}{\vdash p}\,\textsc{DisjE}}{\vdash p \vee p \supset p}\,\textsc{ImpI}$$

admitting a bidirectional reading. Yet, the contorted proof admits a proof term

$$\lambda x.\, \pi_1(\mathbf{case}\ x\ \mathbf{of}\ \langle y.y, y \mid z.z, z\rangle)$$

in the syntax of stratified terms M°. It should not, since we are precisely looking for a notation for bidirectional proofs! The problem with the proof above is that the

DisjE rule hides a potential reduction between rules ConjI and ConjEi: if only we could commute rules DisjE and ConjEi, then we would have

$$\dfrac{[\vdash p \vee p] \qquad \dfrac{\dfrac{[\vdash p] \quad [\vdash p]}{\vdash p \wedge p}\,\textsc{ConjI}}{\vdash p}\,\textsc{ConjEi} \qquad \dfrac{\dfrac{[\vdash p] \quad [\vdash p]}{\vdash p \wedge p}\,\textsc{ConjI}}{\vdash p}\,\textsc{ConjEi}}{\dfrac{\vdash p}{\vdash p \vee p \supset p}\,\textsc{ImpI}}\,\textsc{DisjE}$$

which we would recognize as clearly forbidden: the principal premises of both ConjEi are ConjI, and the whole forms a clear roundabout that reduces to simply $[\vdash p]$.

This is because rule

$$\dfrac{\vdash A \vee B \qquad \begin{array}{c}[\vdash A]\\ \vdots \\ \vdash C\end{array} \qquad \begin{array}{c}[\vdash B]\\ \vdots \\ \vdash C\end{array}}{\vdash C}\,\textsc{DisjE}$$

has a conclusion $\vdash C$ that is not related to its principal premise $\vdash A \vee B$ (this is the case for $\bot$ and $\exists$ also). Or in other words, because its (non-principal) premises and conclusion are *the same* formula, it can come interfere between an introduction and an elimination that would otherwise be banned if put together directly.

Definition 4.2 (Connectives polarity)**.** A logical connective is *positive* if the conclusion of its elimination *is not* a subterm of its principal premise. A logical connective is *negative* if the conclusion of its elimination *is* a subterm of its principal premise.

How to recover full normal forms? Disjunction elimination should not be allowed as the principal premise of an elimination: it should be in the general class of *normal terms* M, not in the specific class of *atomic terms* R. Only then we can call these proofs canonical:

Definition 4.3 (Canonical term assignment)**.** The set of canonical terms are the terms built out of the grammar:

$$\begin{aligned} M, N &::= \lambda x.\, M \mid \mathbf{inl}(M) \mid \mathbf{inr}(M) \mid M, N \mid \mathbf{case}\ R\ \mathbf{of}\ \langle x.\, M \mid x.\, M \rangle \mid R \\ R &::= x \mid R\, M \mid \pi_1(R) \mid \pi_2(R) \end{aligned}$$

We end up with the same syntactical constructs as in the original term assignment, except for the transparent coercion R from normal to atomic terms. From this particular transformation, let us draw a general classification on canonical term assignment, independently of the set of connectives chosen:

Remark 4.1 (Classification of terms). For any set of connectives:

- Canonical terms M are either introductions, or positive eliminations, or atomic terms R
- Atomic terms R are either variables x, or negative eliminations

Besides, from the completeness of our restriction follows a remark that will have its importance in Section 4.3: the data structure of atomic terms look very much like lists:

Lemma 4.1 (Atomic terms are list-like)**.** *If there is a canonical term for each normal proof, then each constructor of atomic terms has at most one direct atomic subterm.*

Proof. Atomic terms are either variables (in which case the statement holds), or negative eliminations. Suppose a negative elimination constructor with two or more direct atomic subterms. Only one of them is the term assigned to its principal premise and must be restricted to be atomic; if one other is too, then there would be no canonical term corresponding to the normal proof involving an introduction on this non-principal subproof of the elimination. □

¶ INTERCALATION The term we assigned to normal proofs should not be a surprise to those familiar with the *intercalation calculus* [Sieg and Cittadini, 2005, Sieg and Byrnes, 1998]. They are exactly notations for intercalations, sometimes referred to as the system of *verifications and uses* [Pfenning, 2010], a system of normal natural deductions designed for proof search where we decompose NJ in two forms of judgment $\vdash A \uparrow$ and $\vdash A \downarrow$, each rule having a subformula property with a direction depending on the form of judgment they prove: $\vdash A \uparrow$ means that we can *verify* the proof of A, reading bottom-up, and $\vdash A \downarrow$ means that we can *use* a proof of A, reading top-down (Figure 4.3).

Both judgments communicate through the Atom rule, which states that "if I am given a proof of an atom p to use, and if I was looking to verify a proof of p, then I can connect the two". The fact that this rule can only be applied on atoms p and not on any formula A is a restriction that we have not seen in the original system. The stratification above was interested in restricting proofs to β-normal forms (no roundabouts). Another common restriction is to make them also η-*long* [Huet, 1976]: this enforces that the proof of a formula begins by introducing its principal connective. Restricting Atom to atoms guarantees that all proofs are fully η-expanded. Since η-expansion depends not on the shape of a proof but on its judgment, in other words it is a property of a *typed* term, the restriction was not enforceable syntactically, but rather by this restriction on the judgment at the boundaries between verifications and uses.

Is this stratification not too restrictive? Can we still write all NJ proofs? Yes, if we can let all problematic **case** constructs "slide" over all other eliminators:

Theorem 4.1 (Soundness and completeness of intercalation)**.** $\vdash A$ *if and only if* $\vdash A \uparrow$.

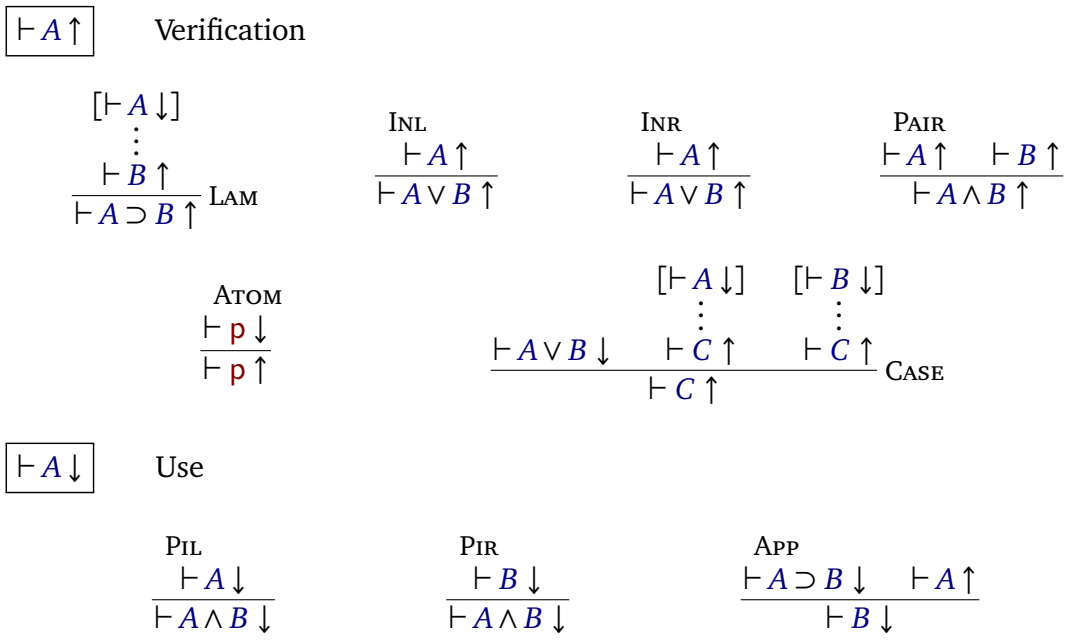

Figure 4.3: The Intercalation calculus

Proof. The direction backwards (soundness) is a straightforward induction on the derivation of $\vdash A \uparrow$, since all intercalations are natural deductions by erasing the direction marker and the occurrences of the Atom rule.

The direction forward (completeness) is proved by induction on the derivation of $\vdash A$, using two lemmas:

Lemma 4.2 (η-expansion)**.** *for any A (not only atomic), if $\vdash A \downarrow$ then $\vdash A \uparrow$.*

Lemma 4.3 (Substitution)**.** *if $\vdash A \uparrow$ and* $\begin{array}{c}[\vdash A \downarrow] \\ \vdots \\ \vdash C \uparrow\end{array}$ *then $\vdash C \uparrow$.*

See e.g., Pfenning [2010] for the details. □

This proof gives rise to a series of reductions on terms: the usual β-reduction, plus four *commutative reductions* allowing a **case** construct to "slide" along a stack of eliminations, gathered on Figure 4.4.

4.2.3 | BIDIRECTIONAL TYPE CHECKING

Consider the grammar of terms M and R assigned to normal proofs we just built. An important remark is that, without any more type annotation, we can easily write a simple type checker for them. We carry the following proof (almost) independently

$$(\lambda x.M)\,N \longrightarrow M[x/N] \tag{4.1}$$
$$\pi_1(M,N) \longrightarrow M \tag{4.2}$$
$$\pi_2(M,N) \longrightarrow N \tag{4.3}$$
$$\textbf{case inl}(M)\ \textbf{of}\ \langle x_1.N_1 \mid x_2.N_2\rangle \longrightarrow N_1[x_1/M] \tag{4.4}$$
$$\textbf{case inr}(M)\ \textbf{of}\ \langle x_1.N_1 \mid x_2.N_2\rangle \longrightarrow N_2[x_2/M] \tag{4.5}$$
$$(\textbf{case}\ M\ \textbf{of}\ \langle x_1.N_1 \mid x_2.N_2\rangle)\,N \longrightarrow \textbf{case}\ M\ \textbf{of}\ \langle x_1.N_1\,N \mid x_2.N_2\,N\rangle \tag{4.6}$$
$$\pi_1(\textbf{case}\ M\ \textbf{of}\ \langle x_1.N_1 \mid x_2.N_2\rangle) \longrightarrow \textbf{case}\ M\ \textbf{of}\ \langle x_1.\pi_1(N_1) \mid x_2.\pi_1(N_2)\rangle \tag{4.7}$$
$$\pi_2(\textbf{case}\ M\ \textbf{of}\ \langle x_1.N_1 \mid x_2.N_2\rangle) \longrightarrow \textbf{case}\ M\ \textbf{of}\ \langle x_1.\pi_2(N_1) \mid x_2.\pi_2(N_2)\rangle \tag{4.8}$$
$$\begin{aligned}&\textbf{case}\ (\textbf{case}\ M\ \textbf{of}\ \langle x_1.N_1 \mid x_2.N_2\rangle)\ \textbf{of}\ \langle y_1.M_1 \mid y_2.M_2\rangle \longrightarrow\\ &\textbf{case}\ M\ \textbf{of}\ \langle x_1.\textbf{case}\ N_1\ \textbf{of}\ \langle y_1.M_1 \mid y_2.M_2\rangle \mid x_2.\textbf{case}\ N_2\ \textbf{of}\ \langle y_1.M_1 \mid y_2.M_2\rangle\rangle\end{aligned} \tag{4.9}$$

Figure 4.4: β-reduction rules of NJ

of the particular set of connective, reasoning inductively on the *classes* of atomic and canonical terms (remark 4.1): atomic terms R consist uniquely of eliminations, and the variable case; canonical terms M consist of introductions, positive, "pathological" eliminations, and the coercion to atomic terms. It is performed constructively, so its statement amounts to a decidability statement.

Theorem 4.2 (Bidirectional type checking)**.** *Let Γ be an environment.*

1. *Given an atomic term R, either there is A such that $\Gamma \vdash R : A$, or there is no such A;*
2. *Given a canonical term M and a type A, either $\Gamma \vdash M : A$ or not.*

Proof.

1. By induction on R: it is either a variable, or a "well-behaved", negative elimination. If $R = x$, then if $x : A \in \Gamma$ we conclude by the variable rule, otherwise no other rule is applicable so there is no such A. If R is a negative elimination with principal premise R', then by induction, either there exists A' such that $\Gamma \vdash R' : A'$ or not. If A' exists, and since it is a negative elimination, the type A of its conclusion must be a subterm of A', so we can conclude. If there is no A', then there is no A either since no other rule applies. Note that R cannot be a positive elimination, otherwise the calculus would not be normal (see previous section).
2. By induction on M: it is either an introduction, or an R (through the coercion), or a "pathological", positive elimination. If $M = R$, then if there exists A' such that $\Gamma \vdash R : A'$ by 1. and $A = A'$ then $\Gamma \vdash R : A$; otherwise there is no such derivation. If M is an introduction of connective $\heartsuit(\mathrm{A}_1, \ldots, \mathrm{A}_n)$, and $A = \heartsuit(B_1, \ldots, B_n)$, then the premises of the rule are canonical terms $M_1 \ldots M_m$: we made no restriction

on subterms of introductions. By induction, if there are derivations $\Gamma \vdash M_1 : B_1$, $\ldots$, $\Gamma \vdash M_n : B_n$, we conclude by rule $\heartsuit$I. If M is a positive elimination, we have no choice but to reason case-by-case on the connective: for instance if $M = \mathbf{case}\ R\ \mathbf{of}\ \langle x.M \mid y.N \rangle$, then if by induction there is $\Gamma, x : B_1 \vdash M : C$ and $\Gamma, y : B_2 \vdash N : C$, we can conclude by DisjE; otherwise, no other rule apply.

□

From this constructive proof, we deduce an algorithm for type checking terms that can *infer* the type of R and *check* if an M has a given type: it is *bidirectional*. This algorithm is presented on Figure 4.5 by two mutually recursive judgments $\Gamma \vdash M \Leftarrow A$ and $\Gamma \vdash R \Rightarrow A$.

Note the rule Atom. At the boundaries of canonical and atomic terms, the type must be an atom p: it ensures that all proof terms are in η-*long* normal form. For instance, judgment $\vdash \lambda xy.x\ y \Leftarrow ((\mathsf{p} \supset \mathsf{q}) \supset \mathsf{r}) \supset (\mathsf{p} \supset \mathsf{q}) \supset \mathsf{r}$ is *not* derivable, but $\vdash \lambda xy.x\ (\lambda z.y\ z) \Leftarrow ((\mathsf{p} \supset \mathsf{q}) \supset \mathsf{r}) \supset (\mathsf{p} \supset \mathsf{q}) \supset \mathsf{r}$ is. Note also that, thanks to bidirectionality, it is never necessary to annotate introductions with types: for instance, the type of variable x in $\lambda x.M$ can always be left out.[1]

This algorithm translates directly to an OCaml program (Figure 4.6), that will be the starting point of our work in the next section. For concision, we use pattern-matching failure to signal errors, and often write patterns on the left of a **let** construct when there is only one case, hence the constant complaints of OCaml about non-exhaustiveness of pattern-matching. Because we are not performing any substitutions in terms, we adopt a named representation of terms. Therefore, environments are association lists from strings to types. We also take the convention to take names of variables reflecting their types.[2]

4.3 | REVERSING THE TYPE CHECKER

In this section, we do a series of program transformations on the type checker of Figure 4.6, going from a recursive to an iterative, accumulator-based one. This requires to understand the concept of *reversal* of list-like data structures, that we first introduce on two simple examples.

[1] There is actually no need for annotations at all with this particular set of connectives. It is not always the case because of "pathological" eliminations: for instance, an elimination $\mathbf{abort}_C(M)$ of $\bot$ would need a type annotation C.

[2] The same way we called M a canonical term *and* the set of all canonical terms when writing "$\ldots$ is a M".

$$A,B \;::=\; A \supset B \;\big|\; A \vee B \;\big|\; A \wedge B \;\big|\; \mathsf{p}$$

$$M,N \;::=\; \lambda x.\, M \;\big|\; \mathbf{inl}(M) \;\big|\; \mathbf{inr}(M) \;\big|\; M,N \;\big|\; \mathbf{case}\ R\ \mathbf{of}\ \langle x.\, M \mid x.\, M\rangle \;\big|\; R$$

$$R \;::=\; x \;\big|\; R\, M \;\big|\; \pi_1(R) \;\big|\; \pi_2(R)$$

$\boxed{\Gamma \vdash M \Leftarrow A}$ Verification/checking

$$\textsc{Lam}\ \frac{\Gamma, x : A \vdash M \Leftarrow B}{\Gamma \vdash \lambda x.\, M \Leftarrow A \supset B} \qquad \textsc{Inl}\ \frac{\Gamma \vdash M \Leftarrow A}{\Gamma \vdash \mathbf{inl}(M) \Leftarrow A \vee B} \qquad \textsc{Inr}\ \frac{\Gamma \vdash M \Leftarrow A}{\Gamma \vdash \mathbf{inr}(M) \Leftarrow A \vee B}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash M \Leftarrow A \quad \Gamma \vdash N \Leftarrow B}{\Gamma \vdash M,N \Leftarrow A \wedge B} \qquad \textsc{Atom}\ \frac{\Gamma \vdash R \Rightarrow \mathsf{p}}{\Gamma \vdash R \Leftarrow \mathsf{p}}$$

$$\textsc{Case}\ \frac{\Gamma \vdash R \Rightarrow A \vee B \quad \Gamma, x : A \vdash M \Leftarrow C \quad \Gamma, y : B \vdash N \Leftarrow C}{\Gamma \vdash \mathbf{case}\ R\ \mathbf{of}\ \langle x.\, M \mid y.\, N\rangle \Leftarrow C}$$

$\boxed{\Gamma \vdash R \Rightarrow A}$ Use/inference

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \textsc{Pil}\ \frac{\Gamma \vdash R \Rightarrow A}{\Gamma \vdash \pi_1(R) \Rightarrow A \wedge B} \qquad \textsc{Pir}\ \frac{\Gamma \vdash R \Rightarrow B}{\Gamma \vdash \pi_2(R) \Rightarrow A \wedge B}$$

$$\textsc{App}\ \frac{\Gamma \vdash R \Rightarrow A \supset B \quad \Gamma \vdash M \Leftarrow A}{\Gamma \vdash R\, M \Rightarrow B}$$

Figure 4.5: Bidirectional typing of canonical NJ

```
type a =
  | Nat | Imp of a × a | Or of a × a | And of a × a

type env = (string × a) list

type m =
  | Lam of string × m | Inl of m | Inr of m | Pair of m × m
  | Case of r × string × m × string × m | Atom of r

and r =
  | Var of string | App of r × m | Pil of r | Pir of r

let rec check env : m × a → unit = function
  | Lam (x, m), Imp (a, b) → check ((x, a) :: env) (m, b)
  | Inl m, Or (a, _) → check env (m, a)
  | Inr m, Or (_, b) → check env (m, b)
  | Pair (m, n), And (a, b) → check env (m, a); check env (n, b)
  | Case (r, x, m, y, n), c → let (Or (a, b)) = infer env r in
    check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c)
  | Atom r, Nat → let Nat = infer env r in ()

and infer env : r → a = function
  | Var x → List.assoc x env
  | App (r, m) → let (Imp (a, b)) = infer env r in
          check env (m, a); b
  | Pil r → let (And (a, _)) = infer env r in a
  | Pir r → let (And (_, b)) = infer env r in b
```

Figure 4.6: Bidirectional type checker of canonical NJ, in OCaml

4.3.1 | A TUTORIAL ON REVERSING LISTS-LIKE STRUCTURES

Let us thus depart completely from our logical motivations for a while, for a detour into program transformation techniques. The goal of this section is to illustrate how to systematically reverse a data structure and the program it operates on. On the program, this amounts to turning a *recursive* function into the composition of a *reversal* and an *iterative* function. We show this on two gradually involved example data structures: first lists, and then *herds*.

A | PRELUDE: THE POWER TOWER

Consider the function:

```
let rec tower = function
  | [] → 1.
  | x :: xs → x ** tower xs
```

It computes the power tower of all elements of the input list:

$$\texttt{tower}\ [x_1; \ldots; x_n] = x_1^{\cdot^{\cdot^{x_n^1}}}$$

This function is non tail-recursive: the maximal size of the stack at run-time is proportional to the length of the list. It begins by looking for the last constructor `[]`, all the way at the bottom of the structure, accumulating continuations (stack frames); only then it starts computing exponents.

What if we are given the same input *xs*, but in reverse order? A priori, because exponentiation is not commutative, there seem to be no easy way to write it similarly, as a recursive descent. Surely we could run `tower (List.rev xs)`, but there is actually a more efficient solution: we can write it in *accumulator-passing style*: we accumulate the result of the partial calculations in an additional argument `acc`, that we return at the end of the computation:

```
let rec rev_tower xs acc = match xs with
  | [] → acc
  | x :: xs → rev_tower xs (x ** acc)
```

This version is tail-recursive: it uses a constant stack space. It should be initially passed 1. as accumulator what was the base case before:

```
let tower' xs = rev_tower (List.rev xs) 1.
```

These are equivalent: for all lists *xs*, `tower` *xs* = `tower'` *xs*. How to prove it? More generally, what is the relationship between these two styles? It turns out that we can go from one into the other by a composition of semantics-preserving program transformations, unveiling two important facts:

- First, the data structure taken by `tower` is different than that taken by `rev_tower`: the first takes a real list, the second takes the *context*, or *zipper* [Huet, 1997] of a list, which happens to be isomorphic to a list. This context is made apparent by the composition of *CPS transformation* and *defunctionalization* [Danvy and Nielsen, 2001].
- Secondly, this context is only half of the story: when generalizing this transformation, we will need to expose the *reverse* data structure, a context closed by *extrusion*. Extrusion is a transformation of our own decoupling the reversal of the list from the function it is used in.

Let us embark on the transformation.

¶ | STEP 1: CPS TRANSFORMATION This well-known program transformation takes a program with general function calls, thus potentially needing a stack to execute, to a higher-order program where all function calls are tail-recursive.[3] We abstract each non-tail call by a *continuation*, a function representing the computation left to do after the call. Let us take our start function. We rename it `aux`, and add to it a continuation argument `k`. In the recursive case, we just add the exponentiation `x ** acc` to the remaining work to do `k` (`acc` representing the "result" of the call to `aux`). We then continue tail-recursively with the rest of the list `xs`.

```
(* aux : float list → (float → float) → float*)
let rec aux xs k = match xs with
  | [] → k 1.
  | x :: xs → aux xs (fun acc → k (x ** acc))
```

At the end of this accumulation in the base case, i.e., when the initial function was about to return, the continuation is finally called with the value 1 that was returned in the original code. This triggers all the work that was accumulated in the continuation. The main call is to `tower0`, which applies the initial, identity continuation `fun acc → acc`:

```
let tower0 xs = aux xs (fun acc → acc)
```

Note that while we got rid of non-tail calls, we introduced a higher-order function `aux` that was not there before the transformation. Let us call respectively `C1` and `C0` the two anonymous functions applied to it.

¶ | STEP 2: DEFUNCTIONALIZATION Reynolds [1972] introduced this transformation for higher-order programs, which has been showcased in various aspects by Danvy and Nielsen [2001] . It takes a program with higher-order functions to a purely first-order program, where higher-order function have been *reified*, that is

[3] in the syntactic sense, but in reality, the stack is represented by linked continuations

turned into a concrete representation, i.e., into a data type of *closures* `k`. In our case, they are all *continuations*. An auxiliary function `apply` emulates the application of these: it takes the "opcode" of a function and returns the function it stands for. We thus replace every call to `k x` by `apply k x`. Every higher-order function is replaced by its "opcode", parameterized by all its free variables, if any. Let us apply *defunctionalization* to our CPS-transformed program:

```
type α k =
  | C0
  | C1 of α × α k

(* apply : k → (float → float) *)
let rec apply = function
  | C0 → (fun acc → acc)
  | C1 (x, k) → (fun acc → apply k (x ** acc))

(* aux : float list → k → float *)
let rec aux xs k = match xs with
  | [] → apply k 1.
  | x :: xs → aux xs (C1 (x, k))

let tower1 xs = aux xs C0
```

As `apply` indicates, type `k` has two constructors: `C0` corresponds to the identity continuation, and `C1` to (`fun acc → apply k (x ** acc)`); since the latter had two free variables `x` and `k`, they become parameters of the constructor. This type `k` is isomorphic to a list: it represents the *context* of an `α list`.[4] Defunctionalization preserves tail-recursion: the resulting program is still tail-recursive, only now first-order. Let us refactor that code a bit to make it clearer:

```
let rec rev_tower acc = function
  | [] → acc
  | x :: xs → rev_tower (x ** acc) xs

let rec aux xs k = match xs with
  | [] → rev_tower 1. k
  | x :: xs → aux xs (x :: k)

let tower2 xs = aux xs []
```

Observe what we end up with: by reifying continuations, we highlighted the fact that our original function was recursive, and was doing all the interesting

[4] McBride [2001] would say $\partial_\alpha(\alpha\ \texttt{list}) = (\alpha\ \texttt{list})^2$

work (the exponentiation) "on the way back". Function `aux` performs the descent, accumulating a list of work `k` to do, and calls `rev_tower` with this list when it is done; `rev_tower` then unrolls this list accumulator and computes the exponentiation in an *accumulator-passing style*.

The scheme just described—defunctionalization of the CPS-transform to reify continuations—is well-known since Danvy and Nielsen [2001], who showed this way the correspondence between a recursive program and an iterative program with accumulator. It has been used further, notably to transform small-step operational semantics into abstract machines [Ager et al., 2003] and big-step semantics [Danvy et al., 2011]. It has also been exploited in the context of type checkers as we are about to do by Sergey and Clarke [2011, 2012], however with a different purpose than what we are about to do.

Their story usually stops here. We extend this scheme one step further, to decouple the reversal from the computation phase: we call this step *extrusion*, as it finishes to separate the context from the actual list and reveals the *reversed* data structure.

¶ STEP 3: EXTRUSION Observe the function `aux`: it takes a list and a continuation, and pushes that list *in reverse order* onto the continuation stack `k`; then only in the base case does it triggers `rev_tower`'s work, passing it the stack and the initial argument 1. Modulo this last detail, this function is exactly the list reversal function `List.rev_append` of OCaml's standard library. Why not isolate this standard function, that does only the reversal, from the particular work of `rev_tower`? This would make the code smaller and more generic. It would also help to show where the "real work" is done, i.e., in function `rev_tower`. In other words, we need to let `aux` return the reversed list `k`, and let the main function `tower2` call `rev_tower`, instead of `aux`. For this we "lift" the base case of `aux` to the main function `tower2` and make `aux` return its accumulator instead; it becomes `List.rev_append [] xs`, which is exactly `List.rev xs`. The body of `tower2` (renamed `tower'`) now reads `rev_tower 1. (List.rev xs)`: first perform the reversal, and then pass the reversed list to `rev_tower`. The result is exactly the accumulator-passing style we were trying to construct:

```
let rec rev_tower acc = function
  | [] → acc
  | x :: xs → rev_tower (x ** acc) xs

let tower' xs = rev_tower 1. (List.rev xs)
```

Note that this last transformation reintroduced a non-tail call to `List.rev`, however it happens only once by call to `tower'`, and not at every recursive call as in the original code. It allows to clearly separate the reversal from the actual computations on the reversed list. Is `tower'` really an optimization of `tower` in terms of stack use? It calls

List.rev, which is recursive, so as for tower, the maximum size of stack needed is proportional to the length of xs. If however the input list xs was given reversed, then rev_tower xs would be more efficient than tower (List.rev xs).

If you are convinced that all transformations we performed preserved semantics, then you will be convinced that:

Theorem 4.3. *for all list xs,* tower *xs* = tower' *xs*.

Proof. by composition of all three semantics-preserving transformations. □

We just showed how to systematically turn a *recursive* function operating on lists into the composition of list reversal and an *iterative* function. In this particular example, extrusion preserves the data structure: List.rev is an endomorphism. This is not the case of all abstract data types: we will now take a small variant of lists, herds, and start again. The transformation is similar, save that the reversed type will not be equal to the input type: the extrusion step is then made a little bit trickier.

B | VARIATION ON THE SAME THEME: *HERDS*

Consider the data type of *herds*, a list of α with a distinguished element of type β at the bottom:

```
type (α, β) herd =
  | Cons of α × (α, β) herd
  | Nil of β
```

The exponentiation function above used a fixed initial value of 1. We could instead put this initial value in the distinguished element:[5]

```
let rec tower = function
  | Nil x → x
  | Cons (x, xs) → x ** tower xs
```

¶ | STEP 1: CPS TRANSFORMATION This step is identical to the previous example, save the base case which now depends on the argument to Nil:

```
let rec aux k = function
  | Nil x → k x
  | Cons (x, xs) → aux (fun acc → k (x ** acc)) xs

let tower1 xs = aux (fun x → x) xs
```

[5] In this case, both α and β are instantiated by type float. Besides, for the sake of the argument, let us ignore that this could be done simply by prepending the initial value, since we have the property of exponentiation $n^1 = n$.

¶ | STEP 2: DEFUNCTIONALIZATION The anonymous functions being the same as before, *defunctionalization* is unchanged. Notably, the type of contexts is still isomorphic to lists: the difference between herds and lists is not at the top of the structure, it is at the bottom.[6]

```
let rec rev_tower acc = function
  | [] → acc
  | x :: xs → rev_tower (x ** acc) xs

let rec aux k = function
  | Nil x → rev_tower x k
  | Cons (x, xs) → aux (x :: k) xs

let tower2 xs = aux [] xs
```

¶ | STEP 3: EXTRUSION The difficulty arises in extrusion: now, the call to `rev_tower` at the end of `aux` depends not only on the reversed list `k`, but also on `x`, the argument of Nil. Therefore, this call cannot be directly lifted to the calling function `tower2`: function `aux` needs to return both pieces of information. We declare a new type

```
type (α, β) dreh = Lin of β × α list
```

encapsulating them both. The base case of `aux` is replaced by the construction of a dreh, and its content deported to an auxiliary function `dreh`. As before, the role of `aux` is to *reverse* a herd; we rename it `rev`:

```
let dreh = function Lin (x, k) → rev_tower x k

let rec rev k : (α, β) herd → (α, β) dreh = function
  | Nil x → Lin (x, k)
  | Cons (x, xs) → rev (x :: k) xs
```

We can now proceed to the extrusion: function `tower2` becomes

```
let tower' xs = dreh (rev [] xs)
```

Note the type of `rev`: it takes a *herd* to a *dreh*.[7] The reverse of a herd is exactly a *dreh* (Figure 4.7): the distinguished element of the Nil constructor at the bottom of the herd is placed on Lin at the top of the *dreh*, the order of all α elements is reversed, and it finishes by a [] (corresponding to the information needed at the "top" of a herd: nothing else than the series of Cons).

As before, we can state the equivalence theorem:

[6] Or in McBride [2001]'s words, $\partial_\alpha((\alpha,\beta)\ \mathsf{herd}) = \partial_\alpha(\mu\gamma.\,\beta + \alpha\times\gamma) = \alpha\ \mathsf{list}\times(\alpha,\beta)\ \mathsf{herd}$.

[7] *dreh* is a semi-palindrome on "herd": it means "turn around" in German.

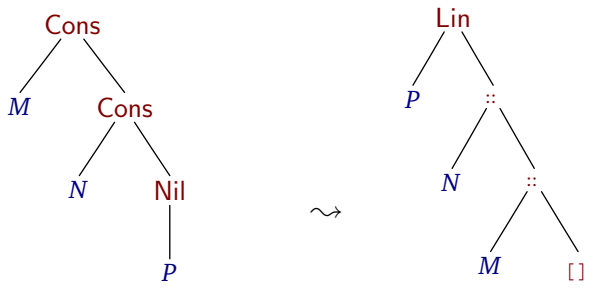

Figure 4.7: From a *herd* to a *dreh*

Theorem 4.4. *for all list* `xs`, `tower xs` = `tower' xs`.

Proof. by composition of all three semantics-preserving transformations. □

4.3.2 | REVERSING ATOMIC TERMS

We are now ready to apply the same series of transformations to our bidirectional type checker of Figure 4.6. Observe the calls to function `infer`: they are all (non tail-)recursive.[8] The same way our list of float stacked up a series of continuations in the `tower` example, atomic terms are traversed recursively, until a variable Var `x` is met: its type is inferred by looking into the environment. On the way back, each atomic element (App, Pil and Pir) is checked. In other words, atomic terms are processed *in reverse order*, just like in the initial `tower` example. Isn't there something to optimize here?

¶ | STEP 1: CPS TRANSFORMATION We begin by *partially* CPS-transforming our type checker, in the sense that it will affect only the `infer` function, and not `check`. We add to it a functional argument `k`, and transform all calls of the form **let** `p` = `infer env r` **in** ... to `infer env r` (**fun** `p` → ...), without forgetting to apply continuation `k`:

```
let rec check env : m × a → unit = function
  | Lam (x, m), Imp (a, b) → check ((x, a) :: env) (m, b)
  | Inl m, Or (a, _) → check env (m, a)
  | Inr m, Or (_, b) → check env (m, b)
  | Pair (m, n), And (a, b) → check env (m, a); check env (n, b)
  | Case (r, x, m, y, n), c → infer env r
    (fun (Or (a, b)) → check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c))
  | Atom r, Nat → infer env r (fun Nat → ())
```

[8] Some calls to `check` are also recursive, but we focus on `infer`: in some sense it is more "urgent" an optimization.

```
and infer env : r → (a → unit) → unit = fun r k → match r with
  | Var x → k (List.assoc x env)
  | App (r, m) → infer env r (fun (Imp (a, b)) → check env (m, a); k b)
  | Pil r → infer env r (fun (And (a, _)) → k a)
  | Pir r → infer env r (fun (And (_, b)) → k b)
```

Note the type of `infer`: it now returns the same type as `check`, i.e., `unit`. This gives rise to five anonymous functions, that we call SCase, SAtom, SApp, SPil and SPir appearing respectively in the Case, Atom, App, Pil and Pir pattern-matching branches.

¶ | STEP 2: DEFUNCTIONALIZATION Let us introduce the new data type of continuations with constructors for each anonymous functions, i.e., apply *defunctionalization*. For reasons that will eventually become clear, let us call it `s` and speak of a *spine*. As before, we parameterize the constructors by the free variables appearing in the body of the corresponding anonymous function, replace all these by their opcode, and all applications of continuation `k x` by `apply k x` where `apply` is a new function associating an opcode to the function it stands for (we uncurry it for readability):

```
let rec check env : m × a → unit = function
  | Lam (x, m), Imp (a, b) → check ((x, a) :: env) (m, b)
  | Inl m, Or (a, _) → check env (m, a)
  | Inr m, Or (_, b) → check env (m, b)
  | Pair (m, n), And (a, b) → check env (m, a); check env (n, b)
  | Case (r, x, m, y, n), c → infer env r (SCase (env, x, m, y, n, c))
  | Atom r, Nat → infer env r SAtom
and apply : s × a → unit = function
  | SPil (_, s), And (a, _) → apply (s, a)
  | SPir (_, s), And (_, b) → apply (s, b)
  | SAtom, Nat → ()
  | SCase (env, x, m, y, n, c), Or (a, b) →
      check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c)
  | SApp (env, m, s), Imp (a, b) → check env (m, a); apply (s, b)
and infer env : r → s → unit = fun r s → match r with
  | Var x → apply (s, List.assoc x env)
  | App (r, m) → infer env r (SApp (env, m, s))
  | Pil r → infer env r (SPil (env, s))
  | Pir r → infer env r (SPir (env, s))
```

Sergey and Clarke [2011, 2012] also use the composition of CPS and defunctionalization to transform a type checker by recursive descent into an abstract typing machine, without any logical connection. For this, they use full CPS transformation and not partial, and go in a different direction afterwards.

¶ STEP 2.5: ENVIRONMENT THREADING Unlike in the examples above, let us stop here and simplify a bit this code. The code generated by the bare defunctionalization complicates uselessly the type checker. For instance, constructor SCase takes an environment as argument. Moreover, if we want the result of the whole transformation to be a type checker itself, we must separate terms, types and environments.

Function `infer` unrolls the atomic term `r`, saving each time in `s` the environment `env`. It just so happens that this environment is constant throughout the recursive invocations of `infer`: we could as well avoid saving it in the constructors, but instead thread it as an additional argument to `apply`: (we show only a few changed lines)

```
(* ... *)
and apply env : s × a → unit = function
  | SPil s, And (a, _) → apply env (s, a)
  | SPir s, And (_, b) → apply env (s, b)
  | SAtom, Nat → ()
  | SCase (x, m, y, n, c), Or (a, b) →
      check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c)
  | SApp (m, s), Imp (a, b) → check env (m, a); apply env (s, b)
and infer env : r → s → unit = fun r s → match r with
  | Var x → apply env (s, List.assoc x env)
  | App (r, m) → infer env r (SApp (m, s))
  | Pil r → infer env r (SPil s)
  | Pir r → infer env r (SPir s)
```

¶ STEP 2.75: RETURN TYPE THREADING Next, observe the path of type `c`, the argument of the SCase constructor: it is placed at the bottom of the spine, `infer` then piles up constructors on top, then it is used in the base case of function `apply`. Let us do the same as for environment: thread it as an argument to `infer` and `apply`. We get: (again showing only a few changed lines)

```
(* ... *)
  | Case (r, x, m, y, n), c → infer env c r (SCase (x, m, y, n))
  | Atom r, Nat → infer env Nat r SAtom
and apply env c : s × a → unit = function
(* ... *)
  | SCase (x, m, y, n), Or (a, b) →
      check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c)
)
and infer env c : r → s → unit = fun r s → match r with
```

```
  | Var x → apply env c (s, List.assoc x env)
(* ... *)
```

The final type of spines, in the light of these optimizations, is the following:

```
type s = SPil of s | SPir of s | SAtom
  | SCase of string × m × string × m | SApp of m × s
```

¶ | STEP 3: EXTRUSION We are left off with an `infer` function whose only role is to turn around an atomic term into a spine, and, in the base case, pass this spine to function `apply`. This function performs the actual check of the spine, so we rename it to `spine`. As in the *herd* example, we now want to isolate the actual work in `spine` from the reverse pass `infer`: we want to *extrude* the calls to `spine` in the body `infer` outside this body, in the caller of `infer`: `check`.

Let us lift off this call to `spine` directly into the call sites of `infer`. For this, we look at the free variables of this base case (`env` and `c` do not count as free variables since they are also arguments at the call sites), and create a type `head` for them:

```
type h = HVar of string × s
```

The base case of `infer` becomes `HVar (x, s)`; we can now isolate it from its mutually recursive block, lighten it from unused argument `env`, and rename it to the more appropriate `rev_spine`:

```
let rec rev_spine s : r → h = function
  | Var x → HVar (x, s)
  | App (r, m) → rev_spine (SApp (m, s)) r
  | Pil r → rev_spine (SPil s) r
  | Pir r → rev_spine (SPir s) r
```

Instead, we insert a new function `head` performing this base case's work:

```
(* ... *)
and head env c : h → unit = function
  | HVar (x, s) → spine env  (s, List.assoc x env)
```

All call sites of former `infer` become the composition of `rev_spine` and `head`:

```
(* ... *)
  | Case (r, x, m, y, n), c → head env c (rev_spine r (SCase (x, m, y, n)))
  | Atom r, Nat → head env Nat (rev_spine r SAtom)
(* ... *)
```

Function `spine` is left unmodified (apart from the renaming). Here it is: the reverse of an atomic term R with respect to a canonical term M is a *head* H.

¶ | STEP 2.5: MORE EXTRUSION The type checker we just constructed is a strange hybrid: given a term M, it type checks as usual all its canonical tip; when it arrives to an atom R, it reverses it to a spine S, and check this spine. Once again, we would like to extrude this reversal step *outside* the actual type checker; this way, both processes would be completely decoupled: given a term M, we could reverse it once and for all, and then begin checking it. For this however, we need to modify the structure of our canonical terms M: where atoms appeared (i.e., in the Atom constructor, but also in the Case constructor), we need to have a `head` (a.k.a upside-down atom). Calling *values* `v` this new type, we end up with:

```
type v = VLam of string × v | VInl of v | VInr of v | VPair of v × v | VHead of h
and h = HVar of string × s
and s = SPir of s | SPil of s | SApp of v × s
  | SAtom | SCase of string × v × string × v
```

Extrusion creates a new function `rev_term : m → v` reversing the canonical terms into values, mutually recursive with `rev_spine`: it calls it to reverse spines, and is called by it for terms in spines to be reversed (App). Function `head` and `spine` are left untouched, only `check` is changed to operate on values.

A | EPILOGUE

Figure 4.8 shows the whole code of this transformed type checker. Notice how `rev_term` is a homomorphism on all cases of type `v`, except Case and Atom, that it transforms into `head`s with `rev_spine`. As we were expecting, function `spine` is now tail-recursive: we transformed a recursive function (`infer`) into an iterative one (`spine`) with an accumulator, the input type `a`. Looking at the whole type checker, the recursive nature of the initial program has been deported to the reversal (which is recursive): we went from a single-pass, recursive program (`check`) to a two-pass one, with an isolated and minimal recursive pass (`rev_term`) followed by an iterative accumulator-based pass (`check'`).[9]

Figure 4.9 shows the reversal of a piece of canonical term into a piece of what we will call a *spine-form* term for now. If we were to construct these terms out of pieces of cord and pearls, and hold the left term by the **case** construct, we would obtain the second by holding it in the middle of the cord attached to x. The variable corresponds to the $\star$ (or HVar) construct, and where before **case** was a canonical term and had a *scrutinee* R (an atomic term), it is now at the tail of a spine, without

[9] Once again, we must precise what we intend by recursive and iterative: only with respect to the helper functions `infer` and `spine`. Indeed, `check'` is still recursive (e.g., in the VPair case), since we performed only a *partial* CPS-translation. Had we performed a complete CPS-translation, we would have come up with an abstract typing machine similar to that of Sergey and Clarke [2012] but we would have lost the logical correspondence that follows.

```
type v =
  | VLam of string × v | VInl of v | VInr of v | VPair of v × v | VHead of h
and h =
  | HVar of string × s
and s =
  | SPil of s | SPir of s | SApp of v × s | SCase of string × v × string × v | SAtom

let rec rev_spine : r → s → h = fun r s → match r with
  | Var x → HVar (x, s)
  | App (r, m) → rev_spine r (SApp (rev_term m, s))
  | Pil r → rev_spine r (SPil s)
  | Pir r → rev_spine r (SPir s)
and rev_term : m → v = function
  | Lam (x, m) → VLam (x, rev_term m)
  | Inl m → VInl (rev_term m)
  | Inr m → VInr (rev_term m)
  | Pair (m, n) → VPair (rev_term m, rev_term n)
  | Case (r, x, m, y, n) →
      VHead (rev_spine r (SCase (x, rev_term m, y, rev_term n)))
  | Atom r → VHead (rev_spine r SAtom)

let rec check' env : v × a → unit = function
  | VLam (x, m), Imp (a, b) → check' ((x, a) :: env) (m, b)
  | VInl m, Or (a, _) → check' env (m, a)
  | VInr m, Or (_, b) → check' env (m, b)
  | VPair (m, n), And (a, b) → check' env (m, a); check' env (n, b)
  | VHead h, c → head env c h
and head env c = function
  | HVar (x, s) → spine env c (s, List.assoc x env)
and spine env c : s × a → unit = function
  | SPil s, And (a, _) → spine env c (s, a)
  | SPir s, And (_, b) → spine env c (s, b)
  | SAtom, Nat → ()
  | SCase (x, m, y, n), Or (a, b) →
      check' ((x, a) :: env) (m, c); check' ((y, b) :: env) (n, c)
  | SApp (m, s), Imp (a, b) → check' env (m, a); spine env c (s, b)
```

Figure 4.8: The final type checker, after transformation

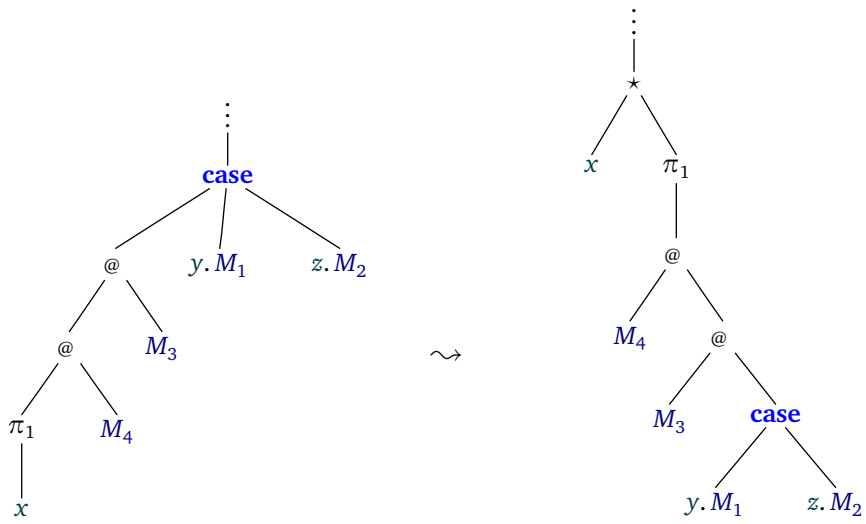

Figure 4.9: From canonical to spine-form terms

scrutinee.

Once again, if you believe that all program transformations did preserve the semantics, then you will agree to the following:

Theorem 4.5. *For all term* M *and environment* `env`*,* `check env` M *if and only if* `check' env (rev_term` M`)`.

Proof. By composition of the semantics-preserving transformations. □

Theorem 4.6. *There is an isomorphism between types* `m` *and* `v`.

Proof. One direction is function `rev_term`, the other is provided by the total function:

```
let rec ver_term : v → m = function
  | VLam (x, m) → Lam (x, ver_term m)
  | VInl m → Inl (ver_term m)
  | VInr m → Inr (ver_term m)
  | VPair (m, n) → Pair (ver_term m, ver_term n)
  | VHead h → ver_head h
and ver_head : h → m = function
  | HVar (x, s) → ver_spine (Var x) s
and ver_spine r = function
  | SAtom → Atom r
  | SCase (x, m, y, n) → Case (r, x, ver_term m, y, ver_term n)
  | SPil s → ver_spine (Pil r) s
  | SPir s → ver_spine (Pir r) s
```

```
    | SApp (m, s) → ver_spine (App (r, ver_term m)) s
```

Then we prove that `rev_term (ver_term v) = v` (*resp.* `ver_term (rev_term m) = m`) by induction on the values of `v` (*resp.* `m`). □

What is this language of spine-form terms we end up with? This is what we shall expose in the next section.

4.4 | THE LJT CALCULUS

Let us write Figure 4.8 in a more natural way. Figure 4.10 presents it as a typed λ-calculus. It is in fact precisely the $\bar{\lambda}$-calculus of Herbelin [1994], which is in Curry-Howard correspondence with LJT.[10] And LJT is a logic in the style of, and provably equivalent to first-order, intuitionistic sequent calculus! This calculus was devised as an intuitionistic restriction of the classical LKT [Danos et al., 1995], itself accepting a dual calculus LKQ.

4.4.1 | DEFINITION

¶ | SYNTAX Values V contain all terms formed of introductions, plus a variable construct. These variables, which were before buried under elimination constructs are now exposed at top-level in the $x(S)$ constructs.[11]

Restricted to its applicative fragment, we can understand the difference between NJ and LJT as the difference between binary and n-ary application: a curryfied application $((x\ M_1)\ M_2)\ M_3$ is now written $x(M_1, M_2, M_3)$, hence the name: a variable against a *spine* of arguments.[12] In the full fragment, a **case** construct like **case** $\pi_1(x)$ **of** $\langle x_1.M_1 \mid x_2.M_2\rangle$ is now written $x(\pi_1, \textbf{case}\langle x_1.M_1 \mid x_2.M_2\rangle)$: destructors are piled up in reverse order.

The fact that variables are visible at top-level of spines has been already recognized crucial for the efficiency of e.g., unification algorithms [Cervesato and Pfenning, 1997]. In practice, all systems manipulating concrete representations of λ-terms of

[10] The name LJT happens to clash with the proof search calculus LJT of Dyckhoff [1992], discussed in Section 2.4. One should not try to find connections between these two: this name clash is merely an unfortunate coincidence, as mentioned in Dyckhoff and Pinto [1998]. Dyckhoff renames it to MJ to point that it is a trade-off between LJ and NJ, but we stay faithful to the original name as long as confusion is avoided.

[11] With respect to data type `v` of Figure 4.8, we inlined the *head* data type `h` into the syntax of values for compactness.

[12] This terminology comes from Cervesato and Pfenning [1997, 2003], and is also used in Section 1.2. Curien and Herbelin [2000] speak more willingly of *continuations* or *contexts* for their relation to abstract machines, terminology that we could have adopted for a different reason, seen that spines emerged precisely from the defunctionalization of continuations.

$$V, W ::= \lambda x.V \mid V, W \mid \mathbf{inl}(V) \mid \mathbf{inr}(V) \mid x(S)$$
$$S ::= V, S \mid \pi_1, S \mid \pi_2, S \mid \mathbf{case}\langle x.V \mid y.W\rangle \mid \cdot$$

$\boxed{\Gamma \vdash V : A}$ Right rules

$$\textsc{ImpR}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \supset B} \qquad \textsc{ConjR}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash M, N : A \wedge B} \qquad \textsc{DisjR1}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathbf{inl}(M) : A \vee B}$$

$$\textsc{DisjR2}\ \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathbf{inr}(M) : A \vee B} \qquad \textsc{Focus}\ \frac{x : A \in \Gamma \quad \Gamma \mid A \vdash S : C}{\Gamma \vdash x(S) : C}$$

$\boxed{\Gamma \mid A \vdash S : C}$ Focused left rules

$$\textsc{ImpL}\ \frac{\Gamma \vdash V : A \quad \Gamma \mid B \vdash S : C}{\Gamma \mid A \supset B \vdash V, S : C} \qquad \textsc{ConjL1}\ \frac{\Gamma \mid A \vdash S : C}{\Gamma \mid A \wedge B \vdash \pi_1, S : C}$$

$$\textsc{ConjL2}\ \frac{\Gamma \mid B \vdash S : C}{\Gamma \mid A \wedge B \vdash \pi_1, S : C} \qquad \textsc{DisjL}\ \frac{\Gamma, x : A \vdash V : C \quad \Gamma, y : B \vdash W : C}{\Gamma \mid A \vee B \vdash \mathbf{case}\langle x.V \mid y.W\rangle : C} \qquad \textsc{Id}\ \frac{}{\Gamma \mid \mathsf{p} \vdash \cdot : \mathsf{p}}$$

Figure 4.10: The LJT/$\bar{\lambda}$ calculus [Herbelin, 1994]

our knowledge adopt n-ary applications instead of binary in their implementation:[13] Coq [Coq development team, 2012], Matita [Asperti et al., 2009], Twelf [Pfenning and Schürmann, 1999] (described in Pfenning and Simmons [2007]), OCaml [Leroy et al., 2012], Beluga [Pientka and Dunfield, 2010].

¶ | TYPING Like the canonical λ-calculus of Figure 4.5, it is composed of two judgment forms. Unlike it, no type can be synthesized, they are all checked: the modes (in the Prolog sense) of terms, types and environment are all *positive* (inputs). This is different from what is found in Cervesato and Pfenning [2003] who define a bidirectional type checker for spine-form terms, but with a smaller, only negative set of formulae: extending a spine-form calculus to disjunction, we must provide the return type. This mode arises in the translation as early as CPS-translation: function `infer` takes the return type of `check`, i.e., `unit`.

Through Curry-Howard, we see environments Γ as *sets* of formulae. Judgment $\Gamma \vdash V : A$ classifies values as having type A. It corresponds to right rules: in this "mode", we can only affect the conclusion A of the judgment by decomposing it (reading the rules upwards). Judgment $\Gamma \mid A \vdash S : B$ is for spines, and corresponds to left rules. A special type A of its environment is distinguished, and commonly called the *stoup*, following Girard [1991]: in this judgment, we can only act on type A. In the light of previous section, we could have called it as well an *accumulator*. Once in "right mode", working on the conclusion, one goes in "left mode" by Focusing on a chosen type of the environment. Once in "left mode", one has no choice but to decompose the type in focus, or apply the Identity on an atomic type (guaranteeing η-long proofs, the equivalent of the Atom restriction above) to finish the proof, or to go back to "left mode" with a **case** construct. In that, LJT is a *focused* sequent calculus.

4.4.2 | FOCUSING IN LJT

Focusing is a proof search strategy that originated from the concept of *uniform proofs* due to Miller et al. [1991], and was formalized by Andreoli [1992]. Originally formulated in the context of Linear Logic [Girard, 1987], it is applicable to many sequent calculi, in particular intuitionistic [Liang and Miller, 2007]. A *focused* sequent calculus has fewer proofs than an unfocused one, thus reducing the non-determinism in proof search; yet the focused system is complete: any proposition provable "unfocusedly" admits a focused proof. The idea is to restrict the application of left or right rules only to a particular *focused* formula: once we begin decomposing it, we cannot switch to work on another one until it is fully decomposed into its

[13] Even if their treatment of other logical connectives (the **case** construct notably) often follows natural deduction.

parts. The definition of what is to be "fully decomposed" determines the kind of focusing. Besides, *strong focusing* (see e.g., Liang and Miller [2009]) implies that the switch can be made only if *all* formulae in the sequent are fully decomposed. On the contrary, *weak focusing* does not enforce this requirement: as long as the *current* formula is fully decomposed, the switch can be made.

¶ | A CHARACTERIZATION OF LJT To define this notion of decomposition, we split the syntax of formulae into two categories of positive and negative connectives (their *polarity*), depending on their inversion properties:

$$\begin{aligned} A^- &::= \mathsf{p}^- \mid A^- \wedge A^- \mid A^+ \supset A^- \mid \uparrow A^+ \\ A^+ &::= \mathsf{p}^+ \mid A^+ \vee A^+ \mid \downarrow A^- \end{aligned}$$

For A to be fully decomposed then means to have arrived at the boundaries of its current polarity, i.e., A is translated to a polarized formula by inserting coercions ($\uparrow$, $\downarrow$); focus change is possible only at these coercions. The translation in question is critical, since there are many ways to translate a given formula and as many search spaces, covering the full range between "optimal" focusing (inserting a minimal number of coercions) and bare sequent calculus: the weakest focusing, introducing coercions at every subformula, allows to switch active formula at every proof step, which is what LJ permits.

In this context, we can redefine LJT as a translation from formulae to polarized formulae:

Definition 4.4. LJT is the weakly focused calculus corresponding to the following encoding of formulae into polarized ones:

$$(A \wedge B)^* = (A)^* \wedge (B)^* \tag{4.10}$$
$$(A \supset B)^* = \downarrow (A)^* \supset (B)^* \tag{4.11}$$
$$(A \vee B)^* = \uparrow (\uparrow (A)^* \vee \uparrow (B)^*) \tag{4.12}$$
$$(\mathsf{p})^* = \mathsf{p}^- \tag{4.13}$$

This encoding is biased towards negative polarity: it returns only negative formulae. This is related to the fact that LJT only has a focused judgment for hypotheses, and is therefore biased towards hypotheses: the default judgment with no focus allows to decompose the goal as well as switch to the focused hypotheses judgment. Specifically, every domain of an implications is coerced to be negative (reflected in the V, S construct: the argument is a value) and so are disjunctions (reflected in the $\mathbf{case}\langle x.\, V \mid y.\, W\rangle$ construct).

¶ | FOCUSING IN NATURAL DEDUCTION We now know that LJT proofs still have a little bit of "slack": they are not fully focused; but they are not either completely unfocused as is LJ. We also know that they are in one-to-one correspondence with NJ proofs. What can we conclude on the "slack" present in NJ proofs? We will illustrate what we learn on a couple of examples:

Example 4.3. The proposition $(p \supset q \supset r) \supset (p \supset q) \supset p \supset r$ admits a focused proof in LJ:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \longrightarrow p}\,\textsc{Id} \quad \dfrac{}{q \longrightarrow q}\,\textsc{Id}}{p \supset q, p \longrightarrow q}\,\textsc{ImpL} \quad \dfrac{}{r \longrightarrow r}\,\textsc{Id}}{p \supset q, p, q \supset r \longrightarrow r}\,\textsc{ImpL}}{\dfrac{\dfrac{}{p \longrightarrow p}\,\textsc{Id}}{p \supset q \supset r, p \supset q, p \longrightarrow r}\,\textsc{ImpL}}}{\longrightarrow (p \supset q \supset r) \supset (p \supset q) \supset p \supset r}\,\textsc{ImpR}\times 3$$

It is also a valid LJT proof, represented by the $\bar{\lambda}$ term:

$$\lambda x y z.x\,(z\,(\cdot), y\,(z\,(\cdot)))$$

It also admits another unfocused proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \longrightarrow p}\,\textsc{Id} \quad \dfrac{\dfrac{}{p \longrightarrow p} \quad \dfrac{\dfrac{}{q \longrightarrow q}\,\textsc{Id} \quad \dfrac{}{r \longrightarrow r}\,\textsc{Id}}{q \supset r, q \longrightarrow r}\,\textsc{ImpL}}{p \supset q, q \supset r \longrightarrow r}\,\textsc{ImpL}}{p, p \supset q, p \supset q \supset r \longrightarrow r}\,\textsc{ImpL}}{\longrightarrow (p \supset q \supset r) \supset (p \supset q) \supset p \supset r}\,\textsc{ImpR}\times 3$$

where we permuted the application of the two last ImpL rules. It is unfocused because we start decomposing hypothesis $p \supset q \supset r$ once, but then switch to decompose another hypothesis $p \supset q$ even if the first was not fully decomposed. This proof does not have a representation in $\bar{\lambda}$: we would need to have a construct allowing to go out of "left mode" back to "right mode" with an implication.

Both proofs inject back into the same NJ proof:

$$\dfrac{\dfrac{\dfrac{[\vdash (p \supset q \supset r)] \quad [\vdash p]}{\vdash q \supset r}\,\textsc{ImpE} \quad \dfrac{[\vdash (p \supset q)] \quad [\vdash p]}{\vdash q}\,\textsc{ImpE}}{\vdash r}\,\textsc{ImpE}}{\dfrac{\dfrac{\vdash r}{\vdash p \supset r}\,\textsc{ImpI}}{\vdash (p \supset q) \supset p \supset r}\,\textsc{ImpI}}{\vdash (p \supset q \supset r) \supset (p \supset q) \supset p \supset r}\,\textsc{ImpI}$$

where the choice (destructing hypothesis $p \supset q$ or $p \supset q \supset r$) is encoded into the notation: after reading upwards until atom $\vdash r$, the reader decides which branch to read first downwards.

As far as implication on the left is concerned, LJT thus makes the "right choice", the one dictated by focusing: the right subformula of the implication is part of the same focus as the implication itself. So does NJ, since both calculi are equivalent, but there it is present as a *quotient* on proofs, thanks to the tree representation. The same remark applies to conjunction on the left.

Disjunction, on the other hand, does not behave as nicely, as we expected from the inefficient polarization LJT makes of it:

Example 4.4. the proposition $p \vee p \supset p \wedge p$ admits a focused proof in LJ:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \longrightarrow p}\,\textsc{Id} \quad \dfrac{}{p \longrightarrow p}\,\textsc{Id}}{p \vee p \longrightarrow p}\,\textsc{DisjL} \quad \dfrac{\dfrac{}{p \longrightarrow p}\,\textsc{Id} \quad \dfrac{}{p \longrightarrow p}\,\textsc{Id}}{p \vee p \longrightarrow p}\,\textsc{DisjL}}{p \vee p \longrightarrow p \wedge p}\,\textsc{ConjR}}{\cdot \longrightarrow p \vee p \supset p \wedge p}\,\textsc{ImpR}$$

represented by the $\bar{\lambda}$-term:

$$\lambda x.x(\mathbf{case}\langle y.y(\cdot) \mid z.z(\cdot)\rangle), x(\mathbf{case}\langle y.y(\cdot) \mid z.z(\cdot)\rangle)$$

It also admits another unfocused LJ proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \longrightarrow p}\,\textsc{Id} \quad \dfrac{}{p \longrightarrow p}\,\textsc{Id}}{p \longrightarrow p \wedge p}\,\textsc{ConjR} \quad \dfrac{\dfrac{}{p \longrightarrow p}\,\textsc{Id} \quad \dfrac{}{p \longrightarrow p}\,\textsc{Id}}{p \longrightarrow p \wedge p}\,\textsc{ConjR}}{p \vee p \longrightarrow p \wedge p}\,\textsc{DisjL}}{\cdot \longrightarrow p \vee p \supset p \wedge p}\,\textsc{ImpR}$$

with rules DisjL and ConjR permuted. It is unfocused because after applying ImpR, the goal is non-atomic ($p \wedge p$), and still we change focus to the hypothesis (rule DisjL). Nonetheless, it is a valid LJT proof, represented by the well-typed $\bar{\lambda}$ term:

$$\lambda x.x(\mathbf{case}\langle y.(y(\cdot), y(\cdot)) \mid z.(z(\cdot), z(\cdot))\rangle)$$

These proofs inject respectively in the NJ proofs:

$$\dfrac{\dfrac{[\vdash p \vee p] \quad [\vdash p] \quad [\vdash p]}{\vdash p}\,\textsc{DisjE} \quad \dfrac{[\vdash p \vee p] \quad [\vdash p] \quad [\vdash p]}{\vdash p}\,\textsc{DisjE}}{\dfrac{\vdash p \wedge p}{\vdash p \vee p \supset p \wedge p}\,\textsc{ImpI}}\,\textsc{ConjI}$$

and

$$\dfrac{[\vdash p \vee p] \quad \dfrac{[\vdash p] \quad [\vdash p]}{\vdash p \wedge p}\,\textsc{ConjI} \quad \dfrac{[\vdash p] \quad [\vdash p]}{\vdash p \wedge p}\,\textsc{ConjI}}{\dfrac{\vdash p \wedge p}{\vdash p \vee p \supset p \wedge p}\,\textsc{ImpI}}\,\textsc{DisjE}$$

with rules ConjI and DisjE permuted. The first proof can be called focused, since it corresponds to a strongly focused LJT proof, whereas the second cannot: the corresponding LJT proof breaks strong focusing.

4.5 | RELATED AND FURTHER WORK

The basis of the work we just exposed, its starting point and finishing line so to say, are two well-known facts: it is well-known since Herbelin [1994, 1995] that LJT proofs are isomorphic to NJ proof; the machinery used to transform a recursive program into an iterative one has been studied in many ways since Danvy and Nielsen [2001]. To our best knowledge however, the connection between these two that we put to light has remained to this day unexplored. It goes through the systematic transformation of a canonical type checker for NJ into a canonical type checker for LJT. Our main observation is that a canonical type checker for a logic in natural deduction style *necessarily* consist of a recursive analysis of its elimination chains (atomic terms), done *on the way back* of the recursion. Reversing the order of these eliminations and turning that recursion into a *loop* gets us a type checker for the same logic, but presented in sequent calculus-style.

Specifically the transformation was a composition of:

- partial *CPS-transformation* of the function for atomic terms, evidencing the recursive nature of atomic terms;
- *defunctionalization*, reifying the continuations into a data structure of reversed atomic terms, i.e., *spines*;
- *optimization* by threading of constant arguments;
- *extrusion*, isolating the reversal of a term from its actual checking.

We observed that the resulting calculus, LJT is more *focused* than the historical sequent calculus of Gentzen [1935], and, knowing that it is by construction isomorphic to our starting point, NJ, we drew conclusions on the focused nature of NJ.

Naturally, this work does not end at this observation, and a large number of variants and extensions remain to be explored to really comprehend what this transformation reveals of the relationship between natural deduction and sequent calculus. Let us present a few of them, some of which we already investigated and some of which remain to be tackled in the future.

4.5.1 | VARIANTS

Although we chose in this development a quite reduced set of logical connectives ($\wedge$, $\vee$, $\supset$), the same scheme extends to all connectives of intuitionistic predicate logic: $\top$, $\bot$, $\forall$, $\exists$, as well as variant definitions of these. For instance, there is an alternative, *multiplicative* definition of the conjunction via the unique elimination:

$$\frac{\vdash A \wedge B \qquad \begin{array}{c}[\vdash A]\quad[\vdash B]\\ \vdots \\ \vdash C\end{array}}{\vdash C}\ \textsc{ConjEí}$$

which makes it a positive connective. It leads to the following normal term assignment (we show only the $\wedge$, $\supset$ fragment):

$$\begin{aligned} M,N &::= \lambda x.M \mid M,N \mid \textbf{let}\ x,y = R\ \textbf{in}\ M \mid R \\ R &::= x \mid R\,M \end{aligned}$$

The new **let** construct corresponding to ConjEí ends up in canonical terms. Writing the type checker and reversing it, we get the following syntax:

$$\begin{aligned} V &::= \lambda x.V \mid V,V \mid x(S) \mid R \\ S &::= \cdot \mid M,S \mid (x,y).M \end{aligned}$$

and the corresponding elimination rule:

$$\frac{\Gamma, y : B, x : A \vdash M : C}{\Gamma \mid A \wedge B \vdash (x,y).M : C}\ \textsc{ConjLí}$$

which, seeing the stoup as a hypothesis and erasing the term information, gives the usual multiplicative left rule of conjunction in the sequent calculus. Note that the premise loses the focus on the hypothesis, and focuses back on the goal. The exact same rule was proposed in Herbelin [1995].

4.5.2 | EXTENSIONS

¶ | MODAL LOGIC OF NECESSITY Pfenning and Davies [2001] propose a reconstruction of modal logic in terms of the Gentzen apparatus, as opposed to the traditional presentation in terms of combinators. They add (among other) a logical connective $\Box A$ to the syntax of propositions, denoting the *necessity* for A to be true *under no hypotheses*, and the following introduction and elimination rules:

$$\frac{\Delta; \cdot \vdash A}{\Delta; \Gamma \vdash \Box A}\ \textsc{BoxI} \qquad \frac{\Delta; \Gamma \vdash \Box A \quad \Delta, A; \Gamma \vdash C}{\Delta; \Gamma \vdash C}\ \textsc{BoxE}$$

This new connective obliges us to split the environment of hypotheses into two distinct sets Γ and Δ, *resp.* the true and the necessarily true assumptions. To use a necessary hypothesis, we use rule:

$$\frac{A \in \Delta}{\Delta; \Gamma \vdash A}\ \textsc{Meta}$$

Note that the elimination suggests that $\Box$ is a *positive connective*. The authors also propose a term assignment for these rules, that we easily make canonical by stratification (again showing only the $\supset$, $\Box$ fragment):

$$\begin{aligned} M, N &::= \lambda x.\, M \mid \mathbf{box}(M) \mid \mathbf{let\ box}\ \mathtt{X} = R\ \mathbf{in}\ M \\ R &::= x \mid \mathtt{X} \mid R\ M \end{aligned}$$

Note the new set of *metavariables* $\mathtt{X}$ referring to necessary hypotheses, stored in Δ. Again, we can write a type checker for these terms, and reverse it. We get the following reversed syntax:

$$\begin{aligned} V &::= \lambda x.\, V \mid \mathbf{box}(V) \mid x(S) \mid \mathtt{X}(S) \mid R \\ S &::= \cdot \mid M, S \mid \mathtt{X}.\, M \end{aligned}$$

Contrarily to all our examples yet, there is two different ways to focus on a particular formula: by choosing among hypotheses or necessary hypotheses. The system we end up with is LJT extended with the left and right rules for necessity, and a focus rule for necessary hypotheses:

$$\text{BoxR}\ \frac{\Delta; \cdot \vdash M : A}{\Delta; \Gamma \vdash \mathbf{box}(M) : \Box A} \qquad \text{BoxL}\ \frac{\Delta, \mathtt{X} : A; \Gamma \vdash M : C}{\Delta; \Gamma \mid \Box A \vdash \mathtt{X}.\, M : C}$$

$$\text{FocusM}\ \frac{\mathtt{X} : A \in \Delta \qquad \Delta; \Gamma \mid A \vdash S : C}{\Delta; \Gamma \vdash \mathtt{X}(S) : C}$$

Seeing the stoup as a (non-necessary) hypothesis, and erasing all term information, we get back the exact sequent calculus rules proposed for the modal logic of necessity in Nanevski et al. [2008].

¶ | THE LJQ SEQUENT CALCULUS Besides taking natural deduction-style calculi and reversing their type checker, we can play the converse game: start from a well-known sequent calculus, and ask ourselves if it is in the image of this transformation, i.e., if there is a corresponding natural deduction-style calculus. The intuitionistic sequent calculus LJQ makes up for an interesting continuation of this work, that we keep for future exploration.

LJQ [Danos et al., 1995] is an intuitionistic sequent calculus that is dual to LJT in the sense that its focusing is biased not on the hypotheses but on the (unique) conclusion. There are therefore two different judgments $\Gamma \vdash A$ (unfocused) and $\Gamma \vdash A|$ (focused on the conclusion). Danos et al. [1995] expose how this duality

relates to the duality between the *call-by-name* and *call-by-value* reduction strategies. Its rules in the minimal fragment of implication are:

$$\textsc{Var}\ \frac{A \in \Gamma}{\Gamma \vdash A \mid} \qquad \textsc{ImpL}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B \mid} \qquad \textsc{ImpR}\ \frac{\Gamma \vdash A \mid \quad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C} \qquad \textsc{Focus}\ \frac{\Gamma \vdash A \mid}{\Gamma \vdash A}$$

to what we can assign the canonical proof terms:

$$\begin{aligned} V &::= x \mid \lambda x.\, L \\ L &::= \mathbf{let}\ x = y\ V\ \mathbf{in}\ L \mid V \end{aligned}$$

that are strongly connected with Moggi [1991]'s *monadic metalanguage*. This chapter's thesis was that there are exactly the same proofs in the usual natural deduction than in LJT, that is the focused proofs for a *left-biased* notion of focus. Is there an alternative natural deduction corresponding to a *right-biased* notion of focus? This is what we could answer to if we could find out the type checker whose transformation leads to the type checker for LJQ.

¶ | (UN)FOCUSED NATURAL DEDUCTION As we observed in the last section, LJT is not a fully focused calculus: it only presents some *features* of focusing, specifically a kind focusing biased toward the hypotheses of its sequent. We showed examples of this "slack" (in the mechanical sense) in LJT proofs, and how it translates back into "slack" in NJ proofs. Two natural questions emerge from this observation. Does it exist a fully focused natural deduction? In other words, what is the calculus that, reversed as we did, projects into a focused intuitionistic sequent calculus, such as the one of Liang and Miller [2007]? Recently, Brock-Nannestad and Schürmann [2010] proposed a linear, focused natural deduction; is its intuitionistic equivalent the calculus we are looking for? On the other side of the focusing spectrum, which natural deduction corresponds to the unfocused, historical sequent calculus? We showed that the tree structure of natural deduction encoded some of the quotients made by focusing sequent calculus; is there a natural deduction-style calculus that realizes this quotient?

¶ | CLASSICAL LOGIC Finally, another interesting extension of this work that we leave for future investigation is classical logic. Since the late eighties and Griffin [1990]'s logical account of **call/cc**, a large body of literature has been written on the computational content of classical logic. The operators and type systems devised are sometimes natural deduction-style [Parigot, 1992], sometimes sequent calculus-style [Curien and Herbelin, 2000]. A legitimate question is then: what is the reversal of such a natural deduction-style type checker? Is there a chance that this reversal turns

out to be the term assignment for a known classical sequent calculus variant? We would start to ask ourselves: what are the canonical forms of e.g., the $\lambda\mu$-calculus? Since the reductions of these languages manipulate the context of a proof in a non-local way, this question is hard and remains unanswered to us.

CONCLUSION

Some two hundred pages ago, we started off our journey by posing a simple question: what is the relationship between a program and its specification, and how can we exploit it to the advantage of the user of the program? This question lead us to the theory of proofs and types, seen as syntactical objects, and the algorithms to verify these proofs, type checkers. We argued that with a universal representation of proofs and purposely designed programming tools, we can increase the trust users have in software and the facility for developers to program such trusted pieces of software; to this end, we proposed a tool to develop programs emitting proof certificates. The scope of our study spanned beyond this field of research, as we learned along the way two significant lessons: first, this very scheme—proof certificates—can be used for a seemingly unrelated purpose, that of checking the static typing of programs in an incremental fashion; secondly, that type checking programs, like any program, are subject to transformations and compilation, and these transformations map back to the proof theory world into well-known theoretical objects, natural deduction and the sequent calculus.

Taking a step back, we can now emphasize the central technical contributions of this thesis:

- the concept of *inverse function* as a facility to write programs on data structures containing binders without maintaining an explicit environment;
- the use of *full, typed evaluation* to type check certificates dynamically in the context of an otherwise untyped language;
- the idea of incremental type checking, its realization by maintaining a shared typing derivation, and its relationship with the inverse of the type checking function;
- the reinterpretation of a sequent calculus as natural deduction "in accumulator passing style", by the systematic derivation of a type checker using off-the-self program transformations.

It is now time to assess and evaluate these accomplishments, taking into account

the bias induced by our subjective eyes, and judge the work that they open for further investigation.

During the elaboration of our framework, we wrote many examples and test cases to validate the approach. Some of them were reproduced here; others can be found in the source code of Gasp. All in all, we can say that the "environment-free" programming style that we advocated seems to be quite natural, once one understands that computations on variables are "expedited" with respect to the usual, first-order style: they have to be made explicit at binding time, not when encountering a variable. In this sense, this style can be seen as a sane restriction, a constraint over the usual, unbounded approach. In terms of conciseness, it compares well with other approaches based on explicit environment. We did not yet consider other applications, like constraint-based type inference, or System F-like inference, which would help assess its scalability further. The reader will also have noticed that our methodology was largely driven by examples. In terms of expressiveness, there would remain to find an adequate criterion of completeness to ensure that we capture "all" computations on binding structures; it is still not clear to us how these can be characterized. To better convince of the generality of the approach and allow it to be applied to different settings, a line of work that we intend to pursue is to modularize it: separate the treatment of binding from typing and evaluation, restrict the latter to a simply-typed setting, enforce our programming style by typing.

Throughout this investigation, maintaining an implementation of our framework took a capital importance. A great care was taken to reduce the size of the code as much as possible. More than a mere programming exercise in brevity, we strongly believe that it can be elevated to a methodology to write principled code, even to a tool that guides the discovery process, and maps back into clear theories. Code factorization for instance can give the intuition of the relationship between two otherwise distinct objects. The last chapter of this thesis emerged this way, and is intended as an illustration of this belief: we used programming tools to relate two logical concepts. In particular, a number of design choices were made in the implementation of Gasp, that are reflected here in the formal presentation. The choice of spine forms in the representation of LF (as opposed to the usual canonical objects) is one of these. Our practical experience is that it makes many concepts easier to code, is more efficient, and exhibits interesting properties, otherwise hidden. In particular, it emerged from this implementation that only a fragment of the hereditary substitution algorithm usually considered was practically necessary, namely one without implicit η-expansion; we conjectured on Page 18 that it is a strongly normalizing fragment, which would have interesting foundational implications: does non-termination in the λ-calculus always result from implicit η-expansions? Many other questionings emerged in the same manner, from programming artifacts, suggesting a much more profound nature to programs.

BIBLIOGRAPHY

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991. A preliminary version was presented at the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL 1990). 24

Andreas Abel and Brigitte Pientka. Explicit substitutions for contextual type theory. In Crary and Miculan [2010], pages 5–20. 24

Umut A. Acar. *Self-adjusting computation*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2005. 121

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In John Launchbury and John C. Mitchell, editors, *POPL*, pages 247–259, Portland, Oregon, January 2002. ACM. ISBN 1-58113-450-9. 121

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In Alex Aiken and Greg Morrisett, editors, *POPL*, pages 14–25, New Orleans, Louisisana, USA, January 2003. ACM. ISBN 1-58113-628-5. 83, 121

Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In Michael I. Schwartzbach and Thomas Ball, editors, *PLDI*, pages 96–107. ACM, 2006a. ISBN 1-59593-320-4. 122

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, 2006b. 121

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press. 127, 141

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992. 55, 74, 153

Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In Juzar Motiwalla and Gene Tsudik, editors, *ACM Conference on Computer and Communications Security*, pages 52–62. ACM, November 1999. ISBN 1-58113-148-8. 74

Andrew W. Appel and Amy P. Felty. Lightweight lemmas in lambda-prolog. In *ICLP*, pages 411–425, 1999. 74

Andrew W. Appel, Neophytos G. Michael, Aaron Stump, and Roberto Virga. A trustworthy proof checker. *J. Autom. Reasoning*, 31(3-4):231–260, 2003. 74

Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the matita proof assistant. *J. Autom. Reasoning*, 39(2):109–139, 2007. 80

Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A compact kernel for the calculus of inductive constructions. *Sādhanā*, 34(1):71–144, 2009. ISSN 0256-2499. 72, 153

Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8(1), 2012. 102

David Aspinall. Proof general: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000. ISBN 3-540-67282-6. 80

Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In Necula and Wadler [2008], pages 3–15. ISBN 978-1-59593-689-9. 71, 118

Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Robinson and Voronkov [2001], pages 19–99. ISBN 0-444-50813-9, 0-262-18223-8. 55

John W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference. In *IFIP Congress*, pages 125–131, 1959. xviii

Hendrik P. Barendregt. Lambda calculi with types. In *Handbook of logic in computer science*, volume 2, pages 117–309. Oxford Univ. Press, New York, 1992. xviii

Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris-Diderot—Paris VII, 1999. 87

Petr Baudiš. Current concepts in version control systems. Bachelor's thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2008. URL `http://pasky.or.cz/bc/bcthesis-final.pdf`. 123

Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in coq. *J. Autom. Reasoning*, 49(2):141–159, 2012. 95

Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Gilles Kahn, editor, *LICS*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE, IEEE Computer Society Press. 73

Jean-Philippe Bernardy. Lazy functional incremental parsing. In Stephanie Weirich, editor, *Haskell*, pages 49–60. ACM, 2009. ISBN 978-1-60558-508-6. 122

Matthieu Boespflug. *Conception d'un noyau de vérification de preuves pour le $\lambda\Pi$-calcul modulo*. PhD thesis, École Polytechnique, Palaiseau, January 2011. 35, 73, 74

Taus Brock-Nannestad and Carsten Schürmann. Focused natural deduction. In Christian G. Fermüller and Andrei Voronkov, editors, *LPAR*, volume 6397 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2010. ISBN 978-3-642-16241-1. 160

Nicholas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972. 5, 71

Rod M. Burstall, David B. MacQueen, and Donald Sannella. HOPE: An experimental applicative language. In *LISP Conference*, pages 136–143, 1980. 71

Luca Cardelli. Program fragments, linking, and modularization. In Lee et al. [1997], pages 266–277. ISBN 0-89791-853-3. 80

Magnus Carlsson. Monads for incremental computing. In Wand and Peyton Jones [2002], pages 26–35. ISBN 1-58113-487-8. 121

Iliano Cervesato and Frank Pfenning. Linear higher-order pre-unification. Technical Report CMU-CS-97-160, Department of Computer Science, Carnegie Mellon University, 1997. 13, 14, 20, 151

Iliano Cervesato and Frank Pfenning. A linear spine calculus. *J. Log. Comput.*, 13(5): 639–688, 2003. 10, 11, 151, 153

Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2): 56–68, 1940. xviii, 1, 6, 73

The Coq development team. *The Coq proof assistant reference manual*. TypiCal Project, 2012. URL `http://coq.inria.fr`. Version 8.4. 80, 153

Karl Crary and Marino Miculan, editors. *Proceedings 5th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice*, volume 34, 2010. 165, 169

Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In Martin Odersky and Philip Wadler, editors, *ICFP*, pages 233–243. ACM, 2000. ISBN 1-58113-202-6. 151, 160

Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. LKQ and LKT: sequent calculi for second order logic based upon dual linear decompositions of classical implication. *Advances in Linear Logic*, 222:211–224, 1995. 151, 159

Olivier Danvy. A new one-pass transformation into monadic normal form. In Görel Hedin, editor, *CC*, volume 2622 of *Lecture Notes in Computer Science*, pages 77–89, Warsaw, Poland, April 2003. Springer. ISBN 3-540-00904-3. 120

Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *PPDP*, pages 162–174. ACM, 2001. ISBN 1-58113-388-X. vi, 139, 141, 157

Olivier Danvy, Jacob Johannsen, and Ian Zerny. A walk in the semantic park. In Siau-Cheng Khoo and Jeremy Siek, editors, *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2011)*, SIGPLAN Notices, Vol. 37, No 3, pages 1–12, Austin, Texas, January 2011. ACM Press. Invited talk. 141

Gilles Dowek. Higher-order unification and matching. In Robinson and Voronkov [2001], pages 1009–1062. ISBN 0-444-50813-9, 0-262-18223-8. 102

Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher order unification via explicit substitutions. *Inf. Comput.*, 157(1-2):183–235, 2000. 87

Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *J. Autom. Reasoning*, 31(1):33–72, 2003. 74

Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. Symb. Log.*, 57(3):795–807, 1992. 55, 56, 151

Roy Dyckhoff and Luis Pinto. Cut-elimination and a permutation-free sequent calculus for intuitionistic logic. *Studia Logica*, 60(1):107–118, 1998. 151

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In Wand and Peyton Jones [2002], pages 48–59. ISBN 1-58113-487-8. 75

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In Robert Cartwright, editor, *PLDI*, pages 237–247. ACM, 1993. ISBN 0-89791-598-4. 120

G. Frege. Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought. *From Frege to Gödel: A source book in mathematical logic*, 1931:1–82, 1879. 1

Gerhard Gentzen. Untersuchungen über das logische Schließen. I. *Math. Z.*, 39(1): 176–210, 1935. ISSN 0025-5874. 1, 2, 55, 56, 125, 157

Herman Geuvers, Robbert Krebbers, James McKinna, and Freek Wiedijk. Pure type systems without explicit contexts. In Crary and Miculan [2010], pages 53–67. 35, 39, 73

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université de Paris VII, Paris, France, 1972. 102

Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. 55, 153

Jean-Yves Girard. A new constructive logic: Classical logic. *Mathematical Structures in Computer Science*, 1(3):255–296, 1991. 153

Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989. 125, 130

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979. ISBN 3-540-09724-4. 80

Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Wand and Peyton Jones [2002], pages 235–246. ISBN 1-58113-487-8. 40, 51, 73

Timothy Griffin. A formulae-as-types notion of control. In Frances E. Allen, editor, *POPL*, pages 47–58. ACM Press, 1990. ISBN 0-89791-343-4. 160

Reiner Hähnle. Tableaux and related methods. In Robinson and Voronkov [2001], pages 100–178. ISBN 0-444-50813-9, 0-262-18223-8. 55

R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. 75

Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, 2007. 6, 8, 11, 13, 16, 20

Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Log.*, 6(1):61–101, 2005. 106

Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993. v, 1, 10, 74

John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *POPL*, pages 458–471, Portland, Oregon, USA, January 1994. ACM Press. ISBN 0-89791-636-0. 120

Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993. 65

Hugo Herbelin. A λ-calculus structure isomorphic to gentzen-style sequent calculus structure. In Leszek Pacholski and Jerzy Tiuryn, editors, *CSL*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75, Kazimierz, Poland, September 1994. Springer. ISBN 3-540-60017-5. vi, 11, 151, 152, 157

Hugo Herbelin. *Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes*. PhD thesis, Université Paris-Diderot—Paris VII, 1995. 157, 158

Paul Hudak and Stephanie Weirich, editors. *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, 2010. ACM. ISBN 978-1-60558-794-3. 175, 176

Gérard Huet. *Résolution d'équations dans les langages d'ordre 1, 2, ..., ω*. Thèse d'état, Université de Paris VII, Paris, France, 1976. 13, 15, 132

Gérard P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997. 139

John Hughes. Super combinators: A new implementation method for applicative languages. In Daniel P. Friedman and David S. Wise, editors, *LFP*, pages 1–10, Pittsburgh, Pennsylvania, August 1982. ACM, ACM Press. 120

Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987. ISBN 3-540-17219-X. 10

Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. 122

Peter Lee, Fritz Henglein, and Neil D. Jones, editors. *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium*, Paris, France, January 1997. ACM Press. ISBN 0-89791-853-3. 167, 172

Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006. ISBN 1-59593-027-2. 29, 74

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system—Documentation and user's manual*, 2012. 153

Stéphane Lescuyer. *Formalisation et développement d'une tactique réflexive pour la démonstration automatique en Coq*. Thèse de doctorat, Université Paris-Sud, January 2011. URL `http://proval.lri.fr/publications/lescuyer11these.pdf`. 55

Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. In Jacques Duparc and Thomas A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 451–465, Lausanne, Switzerland, September 2007. Springer. ISBN 978-3-540-74914-1. 153, 160

Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theor. Comput. Sci.*, 410(46):4747–4768, 2009. 154

Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic J. Philos. Logic*, 1(1):11–60 (electronic), 1996. ISSN 0806-6205. v, xiii

Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 17. Bibliopolis Naples, Italy, 1984. 1, 5, 16

Conor McBride. The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, 2001. 140, 143

Conor McBride and James McKinna. Functional pearl: I am not a number—I am a free variable. In Henrik Nilsson, editor, *Haskell*, pages 1–9. ACM, 2004. 71

S. Michaylov and F. Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Proceedings of the Workshop on the Prolog Programming Language*, pages 257–271, 1992. 11

Donald Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, 1968. 83, 121

Dale Miller. A proposal for broad spectrum proof certificates. In Jean-Pierre Jouannaud and Zhong Shao, editors, *CPP*, volume 7086 of *Lecture Notes in Computer Science*, pages 54–69, Kenting, Taiwan, December 2011. Springer. ISBN 978-3-642-25378-2. 74

Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Y. Shapiro, editor, *ICLP*, volume 225 of *Lecture Notes in Computer Science*, pages 448–462. Springer, 1986. ISBN 3-540-16492-8. 73

Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. 73

Dale Miller and Catuscia Palamidessi. Foundational aspects of syntax. *ACM Comput. Surv.*, 31(3es):11, 1999. 6

Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. Comput. Log.*, 6(4):749–783, 2005. 72

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Ann. Pure Appl. Logic*, 51(1-2):125–157, 1991. 153

Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. 102

Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991. 160

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3), 2008. vi, 72, 87, 88, 120, 159

George C. Necula. Proof-carrying code. In Lee et al. [1997], pages 106–119. ISBN 0-89791-853-3. 28, 29, 74

George C. Necula and Peter Lee. Efficient representation and validation of proofs. In Vaughan Pratt, editor, *LICS*, pages 93–104, Indianapolis, Indiana, USA, June 1998. IEEE Computer Society. ISBN 0-8186-8506-9. 74

George C. Necula and Philip Wadler, editors. *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, San Francisco, California, USA, January 2008. ACM. ISBN 978-1-59593-689-9. 166, 174

Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In Robinson and Voronkov [2001], pages 371–443. ISBN 0-444-50813-9, 0-262-18223-8. 55

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(t). *J. ACM*, 53(6):937–977, 2006. 55

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007. 63, 82

Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, 1991. 83

Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, 1982. 121

Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *LPAR*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992. ISBN 3-540-55727-X. 160

Simon L. Peyton Jones. An introduction to fully-lazy supercombinators. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 175–206. Springer, 1985. ISBN 3-540-17184-3. 120

Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1987. 121

Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Kevin Hall, Will Partain, and Phil Wadler. The Glasgow Haskell Compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93. Citeseer, 1993. 63

F. Pfenning. Lecture notes on judgments and propositions. Carnegie Mellon University, 15-816: Modal Logic, Lecture 1, 2010. URL `http://www.cs.cmu.edu/~fp/courses/15816-s10/lectures/01-judgments.pdf`. 132, 133

Frank Pfenning. Logical frameworks. In Robinson and Voronkov [2001], pages 1063–1147. ISBN 0-444-50813-9, 0-262-18223-8. 10, 87

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001. 72, 120, 158

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *PLDI*, pages 199–208. ACM, 1988. ISBN 0-89791-269-1. 6

Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. *Electr. Notes Theor. Comput. Sci.*, 17:1–13, 1998. 102, 106

Frank Pfenning and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206, Trento, Italy, July 1999. Springer. ISBN 3-540-66222-7. 2, 73, 153

Frank Pfenning and Robert J. Simmons. Term representation. Notes of the LF seminar, Meeting 1, 2007. URL `http://www.cs.cmu.edu/~rjsimmon/papers/lf-meeting/meeting01.pdf`. 10, 13, 153

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In Necula and Wadler [2008], pages 371–382. ISBN 978-1-59593-689-9. 72

Brigitte Pientka. An insider's look at LF type reconstruction: everything you (n)ever wanted to know. *J. Funct. Program.*, 23(1):1–37, 2013. 102, 120

Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In Sergio Antoy and Elvira Albert, editors, *PPDP*, pages 163–173. ACM, 2008. ISBN 978-1-60558-117-0. 72

Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 15–21, Edinburgh, UK, July 2010. Springer. ISBN 978-3-642-14202-4. 34, 72, 153

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000. 14, 111

Adam Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, ESOP'08/ETAPS'08, pages 93–107. Springer, 2008. ISBN 3-540-78738-0, 978-3-540-78738-9. 72

François Pottier. An overview of Cαml. *Electr. Notes Theor. Comput. Sci.*, 148(2): 27–52, 2006. 72

Nicolas Pouillard and François Pottier. A fresh look at programming with names and binders. In Hudak and Weirich [2010], pages 217–228. ISBN 978-1-60558-794-3. 71

Dag Prawitz. *Natural deduction: A proof-theoretical study*, volume 3. Almqvist & Wiksell Stockholm, 1965. 127

William Pugh and Tim Teitelbaum. Incremental computation via function caching. In Michael J. O'Donnell and Stuart Feldman, editors, *POPL*, pages 315–328, Austin, Texas, January 1989. ACM Press. ISBN 0-89791-294-2. 121

David J. Pym. A unification algorithm for the lambda-pi-calculus. *Int. J. Found. Comput. Sci.*, 3(3):333–378, 1992. 102

I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term indexing. In Robinson and Voronkov [2001], pages 1853–1964. ISBN 0-444-50813-9, 0-262-18223-8. 55

Jason Reed. Redundancy elimination for LF. *Electr. Notes Theor. Comput. Sci.*, 199: 89–106, 2004. 74

Thomas W. Reps and Tim Teitelbaum. The synthesizer generator. In William E. Riddle and Peter B. Henderson, editors, *SDE*, pages 42–48. ACM, 1984. ISBN 0-89791-131-8. 122

Thomas W. Reps, Tim Teitelbaum, and Alan J. Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.*, 5 (3):449–477, 1983. 122

John C. Reynolds. Definitional interpreters for higher-order programming languages. *Proceedings of the 25th ACM National Conference*, pages 717–740, 1972. Boston, Massachusetts. 139

John C. Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974. ISBN 3-540-06859-7. 102

John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8. 166, 168, 169, 173, 175

Susmit Sarkar, Brigitte Pientka, and Karl Crary. Small proof witnesses for LF. In Maurizio Gabbrielli and Gopal Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 387–401. Springer, 2005. ISBN 3-540-29208-X. 74

J. Sarnat. *Syntactic finitism in the metatheory of programming languages*. PhD thesis, Yale University, 2010. 11, 16, 20

Ilya Sergey and Dave Clarke. From type checking by recursive descent to type checking with an abstract machine. In Claus Brabrand and Eric Van Wyk, editors, *LDTA*, page 2. ACM, 2011. ISBN 978-1-4503-0665-2. 141, 145

Ilya Sergey and Dave Clarke. A correspondence between type checking via reduction and type checking via evaluation. *Inf. Process. Lett.*, 112(1-2):13–20, January 2012. ISSN 0020-0190. 141, 145, 148

Mark R. Shinwell, Andrew M. Pitts, and Murdoch Gabbay. FreshML: programming with binders made simple. In Colin Runciman and Olin Shivers, editors, *ICFP*, pages 263–274. ACM, 2003. ISBN 1-58113-756-7. 72

Wilfried Sieg and John Byrnes. Normal natural deduction proofs (in classical logic). *Studia Logica*, 60(1):67–106, 1998. 127, 132

Wilfried Sieg and Saverio Cittadini. Normal natural deduction proofs (in non-classical logics). In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 169–191. Springer, 2005. ISBN 3-540-25051-4. 132

Arnaud Spiwack. An abstract type for constructing tactics in coq. In *Proof Search in Type Theory*, 2010. 87

Antonis Stampoulis and Zhong Shao. VeriML: typed computation of logical terms inside a language with effects. In Hudak and Weirich [2010], pages 333–344. ISBN 978-1-60558-794-3. 72

Antonis Stampoulis and Zhong Shao. Static and user-extensible proof checking. In John Field and Michael Hicks, editors, *POPL*, pages 273–284. ACM, 2012. ISBN 978-1-4503-1083-3. 72

Aaron Stump and David L. Dill. Faster proof checking in the Edinburgh Logical Framework. In Andrei Voronkov, editor, *CADE*, volume 2392 of *Lecture Notes in Computer Science*, pages 392–407, Copenhagen, Denmark, July 2002. Springer. ISBN 3-540-43931-5. 74

Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In François Pottier and George C. Necula, editors, *TLDI*, pages 53–66. ACM, 2007. ISBN 1-59593-393-X. 63

Tim Teitelbaum and Thomas W. Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981. 122

D. Vytiniotis. *Practical type inference for first-class polymorphism*. PhD thesis, University of Pennsylvania, 2008. 63

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In Giuseppe Castagna, editor, *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2009. ISBN 978-3-642-00589-3. 75

Mitchell Wand and Simon L. Peyton Jones, editors. *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, 2002. ACM. ISBN 1-58113-487-8. 167, 169

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, March 2003. 10, 13, 14, 16

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer, 2004. ISBN 3-540-22164-6. 10

Joe B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In Gilles Kahn, editor, *LICS*, pages 176–185, Paris, France, July 1994. IEEE, IEEE Computer Society Press. 102

Makarius Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. *Electr. Notes Theor. Comput. Sci.*, 285:101–114, 2012. 80

Freek Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*, 2006. Springer. ISBN 3-540-30704-4. 72

Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *TPHOLs*, volume 3223 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2004. ISBN 3-540-23017-3. 75

Daniel M. Yellin and Robert E. Strom. Inc: A language for incremental computations. *ACM Trans. Program. Lang. Syst.*, 13(2):211–236, 1991. 121

INDEX