An Abstract Machine Based on Linear Logic and Explicit Substitutions

Francisco J. Alberti

A thesis submitted to the
Faculty of Science of the University of Birmingham
for the degree of
MSc in Advanced Computer Science

September, 1997

School of Computer Science
Faculty of Science
The University of Birmingham

Abstract

This thesis presents xLIN, a *linear* abstract machine which may be regarded as a refinement over Krivine's machine, a very simple and efficient abstract machine for the evaluation of λ-terms to weak-head normal form using a *call-by-name* strategy. In the same way that Krivine's machine can be derived from a calculus of *explicit substitutions*, in the spirit of $\lambda\rho$ (or a subcalculus of $\lambda\sigma_w$), xLIN is derived from a weak linear λ-calculus with explicit substitutions, which we called xDILL$_w$. The linear calculus presented here is based on *Dual Intuitionistic Linear Logic* (DILL), which distinguishes explicitly between intuitionistic and linear assumptions by having two *separate* contexts, hence, following Girard's idea in the Unity of Logic. xLIN, like DILL, has the distinctive feature (as opposed to Lafont's linear abstract machine) of implementing two notions of substitution. Linear substitution can be realized effectively as an *in-place update*, while intuitionistic substitution is implemented using environments. We show that xLIN computes faster in the presence of linear resources, and as fast as Krivine's machine, in the presence of non-linear (intuitionistic) resources.

Dedicado a mis padres, Marta y Alfredo,
a mis hermanas, Patricia y Paula,
y a mi sobrino y ahijado, Nicolás.

Acknowledgements

First of all, I would like to express my gratitude to my supervisor, Eike Ritter, for his wisdom, insight, uncountably many discussions, and invaluable friendship.

I am indebted to my tutor, Valeria de Paiva, who also believed in me from the very beginning, encouraged me to work in this area, showed me the beauty of logic, and, above all, honoured me with her friendship. This thesis would not exist if it were not for their constant support.

Thanks to my old friends, Cecilia C. Crespo, Santiago M. Pericás, and, especially, Matías Giovannini, for being always a wonderful critic of my work.

Many thanks to Mathias Kegelmann for showing me the thrill of theorem proving; and to my former supervisor, Achim Jung, for introducing me to semantics.

Finally, my most heartfelt thanks to my family, which are a light in my life, who supported me not only with their love, patience, and understanding, but also financially. This thesis is my tribute to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 A refined logic

Linear logic was introduced in 1987 by the logician Jean-Yves Girard [19] as a refinement of traditional, i.e., classical and intuitionistic, logic. Since its introduction, many applications with the theme 'linear' appeared, including the present work, which modestly contributes to the fields of 'linear functional programming' and 'linear abstract machines'.

Usually, Linear Logic is described as a *resource sensitive* logic, in the following sense. In classical logic, assumptions may be used freely. The fact proved by a theorem may be used to derive other facts as many times as we want (or, simply, not used at all). This is because classical logic is concerned only with *truth*, and not with materialistic aspects like the way the different facts are 'consumed' in a logical deduction. Linear Logic, one the other hand, takes the view of assumptions as *resources*, so that a proof that uses an assumption once, and another one that uses it twice, cannot be proofs of the same theorem. As Girard himself put it, if classical logic is about *stable truths*, Linear Logic is about *actions*.

The *structural* rules that are explicit in Gentzen-style sequent calculus presentations of logic (and that are implicit in natural deduction presentations) bear now a special significance. Due to the presence of *Weakening*, the theorem $(A \wedge B) \Rightarrow A$ can be proved in classical logic, which states that a deduction need not use all its resources; B in this case. The rule of *Contraction* states that resources may be reused as often as necessary; thus, in classical logic $A \Rightarrow A \wedge A$ is provable. Because in Linear Logic the distinction between one occurence of a formula, and more occurences is crucial, the first step consists in eliminating the structural rules. The result is a *linear* system where assumptions that occur in a derivation must be used exactly once, since they cannot be discarded (using Weakening) or duplicated (using Contraction).

The intuitive meaning of linear formulas may be motivated by taking a process-oriented (or action-oriented) viewpoint. For example, the formula $A \multimap B$, read "A linearly implies B", is understood as a *causal* implication, or as a *transformation* of resources, where A is 'consumed' to produce B. Then, it is clear that we cannot have $A \multimap A \otimes A$, where $\otimes$, read "tensor", stands for linear conjunction; from one supply of A, there is no possible way we can produce two! Also, $A \otimes A \multimap A$ is not provable. Note that in classical logic, if we have $A \Rightarrow B$, and we have A, we can use it to derive B, but A will continue to be true. In Linear Logic, once we derived B, we do not longer count with A.

But, what is the general usefulness of Linear Logic, if we are obliged to use resources only

once? To be able to recover the full power of traditional logic, Girard introduced another operator '!', which is referred to by many names: "exponential", "pling", "shriek" (even, "bang!"). A formula $!A$ is intepreted as providing an unlimited number of resources A. Hence, we allow Contraction and Weakening only for 'plinged' formulas.

The classical, non-causal, implication has a natural decomposition in Linear Logic, bearing in mind a resource-oriented view, namely, $A \Rightarrow B$ can be regarded as $!A \multimap B$. In linear terms, linear implication will consume one instance of A, but now we can produce as many copies of A as we want. We cannot loose the possibility of copying and discarding resources, tbut at least we can aim at explicitly distinguishing where (in a deduction, for example) we have to do so.

Some of the first applications of Linear Logic investigated in the field of functional programming, is Lafont's [25] implementation of a linear abstract machine! Linear objects, once referenced, can be safely discarded. Recall that linear resources are used exactly once. Non-linear resources are explicitly duplicated and discarded, so Lafont's abstract machine mimics this behaviour and has instructions that duplicate and discard non-linear values. Garbage collection is automatically accounted for. The price paid for the correctness of this procedure is precisely in the expensive operation of copying. This is one crucial difference between xLIN and Lafont's abstract machine. (The reader is referred to Scedrov 'tour' guide on the research in the area of theory and applications of Linear Logic [34].)

In the same way that Intuitionistic Logic corresponds to the λ-calculus through the Curry-Howard isomorphism [24], Intuitionistic Linear Logic corresponds to a Linear λ-calculus. Since formulas are interpreted as types, $A \multimap B$ is understood as the type of a linear function from A to B that uses its argument exactly once; the type $!A \multimap B$ is understood as standing for functions (now in the classic sense of the λ-calculus) that use their arguments many times (possibly zero times).

A natural deduction presentation of Intuitionistic Linear Logic, as presented by Benton, Bierman, de Paiva, and Hyland [7], includes new four constructs that account for the ! connective. Weakening and Contraction appear as explicit 'copy' and '$\mathsf{discard}$' constructs, as well as the usual introduction and elimination rules. Note that in this presentation there are four rules for the ! connective instead of two, as is usual. Also, the promotion rule is not as simple as one expects. The obvious attempt

$$\frac{\Gamma \triangleright M : A}{\Gamma \triangleright !M : !A}$$

does not work in practice: the resulting system lacks an important property that is crucial for reduction. The solution yields a more complicated (and subtle) presentation of the logic (and the calculus).

An alternative presentation, which is the one used in this thesis, corresponds to Barber's [4] Dual Intuitionistic Linear Logic (DILL). The essential characteristic of DILL is the abscence of Weakening and Contraction as explicit rules! Instead, DILL distinguishes between intuitionistic (non-linear) and linear assumptions. This is achieved by regarding contexts as split into two $\Gamma\,;\Delta$, where Γ keeps track of the intuitionistic assumptions and Δ keeps track of the linear assumptions. In a way, Γ can be thought of as an 'intuitionistic' context (in the classical sense), so Weakening and Contraction are implicit. In DILL, a linear formula $!A$, is essentially equivalent to an intuitionistic formula A. The natural deduction presentation is simpler, since we recover the symmetry of the introduction-elimination scheme, and the Promotion rule, its simplicity.

1.2 Linear and intuitionistic explicit substitutions

The λ-calculus proved to be a successful model for functional programming languages, and as a theory of computable functions in general (refer to [5] for an exposition). However, from a more pragmatic viewpoint, we realize that the calculus is too high level, in the sense that it is not possible to reason about the operational behaviour of programs in a satisfactory way. The reason is that substitution, in the definition of the β-rule of the λ-calculus, appears as a meta-operation:

$$(\lambda x.M)\, A \to M[x := N],$$

The variable x may occur any number of times in M, and M may have an arbitrary size.

Abstract machines, in particular, *environment machines*, appear as mathematical devices where the operation of substitution is taken care of. A single reduction step in the calculus corresponds to many reduction steps in an abstract machine, which can be formalized as a transition system. Since Landin's SECD [26], many machines followed, most notably, Krivine's abstract machine [12], the simplest of all environment machines.

The disadvantage of abstract machines over reduction calculi is obvious. Abstract machines have a fixed strategy; in Krivine's machine, for example, the strategy is the call-by-name strategy. The best idea, then, would be to have a *confluent* reduction calculus as refined as a transition relation of an abstract machine, to be able to study clever strategies for the implementation of the substitution operation, and for addressing complexity issues like space and time requirements. This is the beginning of one of our central topics, calculi with *explicit substitutions*.

We should remark that we are not interested in proposing another calculus with explicit substitutions; there are a lot already. We have decided to take the design ideas behind a subcalculus of the original $\lambda\sigma$-calculus, and apply it to obtain a subcalculus for DILL, with the aim of deriving a linear abstract machine. Since, Abadi et al. [1] introduced the $\lambda\sigma$-calculus they showed that by restricting the reduction rules to a call-by-name strategy (in a particular way that is made more explicit by Curien's $\lambda\rho$), we obtain Krivine's abstract machine.

There is no special reason why we chose a call-by-name strategy for presenting xLIN, except that we wanted, naturally, the simplest exposition possible. Is not a surprise then, that our abstract machine comes as a refinement of Krivine's machine, in a way we will see in the development of this thesis. In fact, xLIN can be optimized in an obvious way to have Krivine's machine as a *subset*.

The reader may wonder what is the pay-off of having a linear abstract machine? The answer is that xLIN differentiates between two classes of substitution: intuitionistic and linear. This is reflected in DILL itself, which distinguishes between intuitionistic and linear variables. Since linear variables satisfy the property that they will occur *exactly once* in a term, linear substitution can be implemented very efficiently using a simple update operation. The idea is that xLIN computes very efficiently in the precense of linear resources, and computes as fast a Krivine's machine, in the precense of non-linear resources.

1.3 Overview of this thesis

The thesis commences in a tutorial fashion by introducing Intuitionistic Logic in chapter 2, where the idea is to give a flavour of the relevance of logic in this thesis. We briefly motivate

Linear Logic by looking closely at the role of the structural rules.

The topic of chapter 3 is the simple typed λ-calculus, where we pay special emphasis in the definition of the reduction calculus, leaving out semantic-related aspects, like extensionality. We introduce the call-by-name strategy and (also briefly) motivate abstract machines.

Chapter 4 adds explicit substitutions to the λ-calculus. As an introduction, we first consider a very simple calculus, and then extend it to be able to finally 'derive' Krivine's abstract machine. We show how correctness follows in an obvious way.

Intuitionistic Linear Logic is introduced in chapter 5. Basic knowledge on Linear Logic is advised, although it is not strictly needed. We hint at linear functional programming, which can also be used as a source of intuition to understand Linear Logic. In this chapter we will introduce the two notions of substitution that are crucial to understand the weak linear calculus of explicit substitutions presented in the next chapter.

In chapter 6 we present the calculus behind xLIN and, finally, the machine itself. We show its relationship with Krivine's machine, and explicit its correctness. Chapter 7 briefly describes the implementation of a simple interpreter that was used to test the abstract machine.

In chapter 8 we present our conclusions and give possible future directions.

Chapter 2

Intuitionistic Logic

Traditional logic and the λ-calculus seem to be two completely different incompatible worlds. It was recently shown, although it had been noticed some time before, that they are not. In fact, they may be regarded as two sides of the same coin, two views of the same abstract notion. This correspondence is widely known as the 'Curry-Howard isomorphism' or 'formulas-as-types' principle, where logical formulas and types, proofs and λ-terms, and proof normalization and term reduction are put in one-to-one correspondence. The 'traditional logic' that we are talking about is *Intuitionistic Logic* and the 'λ-calculus' is the simple typed λ-calculus. The isomorphism was first noticed by Curry between the implicational fragment of this logic, i.e., the fragment with only proposition variables and the implication connective '$\Rightarrow$', and combinatory logic. Howard [24] generalized it to first-order logic and an extended λ-calculus.

We will not discuss Intuitionistic Logic in its own right, nor its relevance to mathematics and logic in general. For the necessary historical background and a complete introduction to the subject, the reader may consult Troelstra and van Dalen [38]. In any case, we will give a natural deduction presentation of the logic and summarize some of the results in Proof Theory that are of particular relevance to our development in this thesis.

Althought the formulas-as-types principle is interesting in its own right, our focus in this thesis will be mainly methodological. In view of this principle, given an intutionistic (also termed 'constructive') logic, we can look at its *operational* side by studying its corresponding term-calculus. A sample application is the linear λ-calculus of chapter 5 and some of the related results that finally lead to the abstract machine of chapter 6. Other calculi were derived from other (constructive) well-known modal logics; for example, see Bierman and de Paiva [9]. The isomorphism also works the other way around, given a λ-calculus we can look at its, implicit, *logical* reading. An exploration of these ideas can be found in the book by Girard, Lafont, and Taylor [20].

To be able to characterize the Curry-Howard isomorphism and sketch our tools, we begin by presenting the syntax for intuitionistic propositional logic (IPL). We will not be interested in intuitionistic negation[1].

2.1 Natural deduction

2.1.1 Definition (well-formed formulas) The set of *well-formed formulas* of IPL, ranged

[1] In the literature, this fragment of the logic that does not include negation is called *minimal logic*.

over by the letters A,B,C is generated by the grammar

$$A, B, C ::= P \mid A \Rightarrow B \mid A \wedge B \mid A \vee B$$

where P is a set of propositional variables, and '$\Rightarrow$', '$\wedge$', and '$\vee$' stand for implication, conjunction, and disjunction, respectively. As is usual we will use parenthesis for grouping. The connective $\Rightarrow$ takes the lowest precedence and $\wedge$ the highest; for example, the formula $A \wedge B \Rightarrow A \wedge B \vee C$ is to be understood as $(A \wedge B) \Rightarrow ((A \wedge B) \vee C)$. Also, $\Rightarrow$ is right-associative; hence, $A \Rightarrow B \Rightarrow C$ will be used as an abbreviation for $A \Rightarrow (B \Rightarrow C)$.

In the sequel, we assume all formulas to be well-formed. □

Not only logics come in different flavours, but also in different ways of presenting them. The one presented here is a variant of Gentzen's *natural deduction* using *sequents*. This system has some practical advantages over, for example, a *sequent calculus* presentation. For a detailed survey of these formalisms see Troelstra and Schwichtenberg [37].

2.1.2 Definitions (sequents, rules, and derivations) The system of natural deduction consists of a set of rules, called *logical rules*, of the form

$$\frac{\langle \text{premise} \rangle_1 \quad \cdots \quad \langle \text{premise} \rangle_n}{\langle \text{conclusion} \rangle}$$

stating that if the ⟨premise⟩'s hold, then the ⟨conclusion⟩ holds, where each $\langle \text{premise} \rangle_i$ ($1 \leq i \leq n$), for $n \geq 0$, and the ⟨conclusion⟩ denote assertions, called *sequents*. A sequent is written

$$\Gamma \vdash A$$

and expresses the fact that from a *set of assumptions* Γ, usually called a *context*, we can *deduce* or *derive* the formula A. A *deduction* or *derivation* is a proof viewed as a tree; the nodes are formulas (where the leaves correspond to the assumptions), and a node has as successors the premises determined by the application of a rule. We will say that a formula A is a *theorem* if the sequent $\varnothing \vdash A$ (or $\vdash A$, for short) appears as the conclusion of the last rule in a derivation.

In natural deduction, the rules come in pairs; the *introduction* rules, marked $\circledast$-I (for $\circledast$ one of the connectives $\Rightarrow$,$\wedge$,$\vee$), introduce a connective in the conclusion; and the *elimination* rules, marked $\circledast$-E, eliminate it from the premises. The rule Axiom does not depend on any premises ($n = 0$) and states that a formula A can be derived if it already appears as an assumption.

The theorems of IPL are the formulas that can be derived using the rules of Figure 2.1. We write sets as sequences $A_1, A_2, \ldots, A_n$ of assumptions, where ',' is interpreted as union; then, we write Γ, A as an abbreviation for $\Gamma \cup \{A\}$. □

The interpretation of the rules is fairly straightforward; for example, $\Rightarrow$-I is the 'deduction theorem', $\Rightarrow$-E corresponds to 'modus ponens', and $\wedge$-I says that $A \wedge B$ can be proved if we have proved A and B separately. For example, the following is a derivation for $A \wedge B \Rightarrow$

$$\frac{}{\Gamma, A \vdash A}\ \text{Axiom}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-I} \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \Rightarrow\text{-E}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-I} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\text{-E}_1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\text{-E}_2$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\text{-I}_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\text{-I}_2 \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee\text{-E}$$

Figure 2.1: Intuitionistic Propositional Logic, IPL

$(B \Rightarrow C) \Rightarrow A \wedge C$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A \wedge B, B \Rightarrow C \vdash A \wedge B}{A \wedge B, B \Rightarrow C \vdash A}\ \wedge\text{-E}_1 \qquad \dfrac{A \wedge B, B \Rightarrow C \vdash B \Rightarrow C \qquad \dfrac{A \wedge B, B \Rightarrow C \vdash A \wedge B}{A \wedge B, B \Rightarrow C \vdash B}\ \wedge\text{-E}_2}{A \wedge B, B \Rightarrow C \vdash C}\ \Rightarrow\text{-E}}{A \wedge B, A \Rightarrow C \vdash A \wedge C}\ \wedge\text{-I}}{A \wedge B \vdash (B \Rightarrow C) \Rightarrow A \wedge C}\ \Rightarrow\text{-I}}{\vdash A \wedge B \Rightarrow (B \Rightarrow C) \Rightarrow A \wedge C}\ \Rightarrow\text{-I}$$

As a convention, we will not label the instances of the Axiom rule.

2.2 The Curry-Howard correspondence

Under the Curry-Howard isomorphism, formulas of IPL are viewed as types of the simple typed λ-calculus, terms as 'encoding' natural deduction proofs (derivations), and proof normalization (see next section) as β-reduction.

To illustrate this idea, let's look at an example first. The proof

$$\dfrac{\dfrac{\dfrac{A \wedge (A \Rightarrow B) \vdash A \wedge (A \Rightarrow B)}{A \wedge (A \Rightarrow B) \vdash A \Rightarrow B}\ \wedge\text{-E}_2 \qquad \dfrac{A \wedge (A \Rightarrow B) \vdash A \wedge (A \Rightarrow B)}{A \wedge (A \Rightarrow B) \vdash A}\ \wedge\text{-E}_1}{A \wedge (A \Rightarrow B) \vdash B}\ \Rightarrow\text{-E}}{\vdash A \wedge (A \Rightarrow B) \Rightarrow B}\ \Rightarrow\text{-I}$$

corresponds to the term $\lambda x{:}A{\times}(A{\to}B).(\mathsf{snd}\,x)\,(\mathsf{fst}\,x)$ by the following derivation

$$\dfrac{\dfrac{\dfrac{x : A \times (A \to B) \triangleright x : A \times (A \to B)}{x : A \times (A \to B) \triangleright \mathsf{snd}\,x : A \to B}\times\text{-E}_2 \qquad \dfrac{x : A \times (A \to B) \triangleright x : A \times (A \to B)}{x : A \times (A \to B) \triangleright \mathsf{fst}\,x : A}\times\text{-E}_1}{x : A \times (A \to B) \triangleright (\mathsf{snd}\,x)\,(\mathsf{fst}\,x) : B}\to\text{-E}}{\triangleright\ \lambda x{:}A{\times}(A{\to}B).(\mathsf{snd}\,x)\,(\mathsf{fst}\,x) : A \times (A \to B) \to B}\to\text{-I}$$

where we have written '$\triangleright$' instead of the turnstile '$\vdash$' to explicitly distinguish between the sequents in the logical derivations and the sequents in the derivations 'annotated' with terms. The term denotes the function that takes a pair and returns the result of applying the second component of the pair to the first component. The type of the pair is $A \times (A \to B)$, where $\times$ is as usual the cartesian product of A and $A \to B$, where the latter denotes the type of functions from A to B. The above derivation was obtained by annotating the formulas with terms, as follows.

Assumptions are annotated with variables, and for each introduction (elimination) rule there is a matching term constructor (destructor) of the corresponding formula regarded as a type. In the example, the term constructor for the type $A \to B$ is a function abstraction[2]; for the type $A \times B$ is the pair constructor $\langle -, - \rangle$. Likewise, the application, written simply by juxtaposing terms, is the deconstructor for the function space, and so on.

Note that this derivation is nothing else than a proof of how to type-check the given term. Here, the sequents, now sets of typing judgments, are read as *declarations* of variables, in the usual programming sense.

A sequent in the annotated version, or *typing judgement*, has the general form

$$\Gamma \triangleright M : A$$

which states that "(term) M has type A". The annotated rules are now viewed as *type checking-rules*. Figure 2.2 gives the rules for all the term constructs (which we will refer to as IPL$^{\lambda}$). The syntax for terms used in these rules correspond to an extended simple typed λ-calculus, in Church-style (see Barendregt [6]) where function abstractions indicate explicitly the type of the bound variable. In the next chapter we will discuss this calculus in more detail.

The type $A{+}B$ (the image under the isomorphism of $A \vee B$) is the disjoint sum of A and B. The terms 'inl' and 'inr' (*inject left* and *inject right*, respectively) construct an object of this type from objects of types A or B. It can be thought of as constructing a 'tagged' object; the tag indicates whether the object contained has type A (to the right of '$+$') or B (to the left). The 'case' term destructor is inspired from the corresponding statement for accessing *variant records* (as in Pascal). The terms 'fst' and 'snd' are pair destructors, returning the first or second component, respectively. Application is as usual noted by term juxtaposition. Modern functional programming languages include these, among many others, as their built-in types.

2.2.1 Remark (syntax-directed type-checking) Note that terms encode all the information necessary so that it actually stands for a proof of the formula it denotes, i.e. the type of the term. This type may be easily reconstructed by a recursive (type-checking) algorithm that would actually try to verify if the term denotes a valid prooftree, i.e. if the rules are applied

[2]The notation corresponds to the syntax of the *typed λ-calculus*. See next chapter.

$$\frac{}{\Gamma, x : A \triangleright x : A}\ \text{Axiom}$$

$$\frac{\Gamma, x : A \triangleright M : B}{\Gamma \triangleright \lambda x{:}A.M : A \to B}\ \to\text{-I} \qquad \frac{\Gamma \triangleright M : A \to B \quad \Gamma \triangleright N : A}{\Gamma \triangleright M\ N : B}\ \to\text{-E}$$

$$\frac{\Gamma \triangleright M : A \quad \Gamma \triangleright N : B}{\Gamma \triangleright \langle M, N \rangle : A \times B}\ \times\text{-I} \qquad \frac{\Gamma \triangleright M : A \times B}{\Gamma \triangleright \mathsf{fst}\, M : A}\ \times\text{-E}_1 \qquad \frac{\Gamma \triangleright M : A \times B}{\Gamma \triangleright \mathsf{snd}\, M : B}\ \times\text{-E}_2$$

$$\frac{\Gamma \triangleright M : A}{\Gamma \triangleright \mathsf{inr}\, M : A + B}\ +\text{-I}_1 \qquad \frac{\Gamma \triangleright M : B}{\Gamma \triangleright \mathsf{inl}\, M : A + B}\ +\text{-I}_2$$

$$\frac{\Gamma \triangleright M : A + B \quad \Gamma, x : A \triangleright N : C \quad \Gamma, y : B \triangleright L : C}{\Gamma \triangleright \mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\, x \to N\, ;\, \mathsf{inr}\, y \to L : C}\ +\text{-E}$$

Figure 2.2: Type-checking rules for IPL$^\lambda$

according to the pattern of Figure 2.2. In practice, one should not expect the programmer to indicate the type of every variable, and that is the approach taken by type systems in the style of Curry that do not enforce such a requirement, but where each term may have many possible types. The language ML is an example of such a language.

□

2.3 Proof normalization

Proving theorems in natural deduction is much of an art. There is not a unique way of deriving a particular formula, although, after some practice, one develops the skills to come about with a 'direct' proof.

While some proofs may represent different strategies, others may contain deduction steps that do not contribute to the proof in an essential way. These may represent 'detours' in the main logical argument, that can be eliminated in order to come up with a shorter, more direct, version. For example, the proof

$$\dfrac{\dfrac{\dfrac{\dfrac{(A \Rightarrow B), A, B \vdash B \quad (A \Rightarrow B), A, B \vdash A}{(A \Rightarrow B), A, B \vdash B \wedge A}\ \wedge\text{-I}}{(A \Rightarrow B), A \vdash B \Rightarrow B \wedge A}\ \Rightarrow\text{-I} \qquad \dfrac{(A \Rightarrow B), A \vdash A \Rightarrow B \quad (A \Rightarrow B), A \vdash A}{(A \Rightarrow B), A \vdash B}\ \Rightarrow\text{-E}}{(A \Rightarrow B), A \vdash B \wedge A}\ \Rightarrow\text{-E}}{(A \Rightarrow B) \vdash A \Rightarrow B \wedge A}\ \Rightarrow\text{-I}}{\vdash (A \Rightarrow B) \Rightarrow A \Rightarrow B \wedge A}\ \Rightarrow\text{-I}$$

proves the formula $B \wedge A$ twice. The first occurence appears as the conclusion of the rule

$\wedge$-I, which was used to introduce the implication $B \Rightarrow (B \wedge A)$, so that, by eliminating it immediately after, we would be able to prove $B \wedge A$, again! The evidence of this redundancy is the occurence of an introduction rule followed by an elimination rule for the implication in $B \Rightarrow (B \wedge A)$.

The above proof can be easily transformed into the shorter one by leaving the second derivation of $B \wedge A$ (at the top) and replacing the occurrences of the assumption B by its proof (on the right):

$$\dfrac{\dfrac{\dfrac{(A \Rightarrow B), A \vdash A \Rightarrow B \quad (A \Rightarrow B), A \vdash A}{(A \Rightarrow B), A \vdash B} \Rightarrow\text{-E} \qquad (A \Rightarrow B), A, B \vdash A}{\dfrac{(A \Rightarrow B), A \vdash B \wedge A}{(A \Rightarrow B) \vdash A \Rightarrow B \wedge A} \Rightarrow\text{-I}} \wedge\text{-I}}{\vdash (A \Rightarrow B) \Rightarrow A \Rightarrow B \wedge A} \Rightarrow\text{-I}$$

A proof that does not contain any detours, i.e., occurrences of an introduction rule followed by an elimination rule for the same connective, is said to be *normal* or in *normal form*. Figure 2.3 shows the normalization cases for the other connectives, where we have used the notation

$$\begin{array}{c} \vdots\,\pi \\ \Gamma \vdash A \\ \vdots\,\pi' \\ \Gamma \vdash B \end{array}$$

to mean that the proof π of A is substituted for the occurrences of A as an assumption in B.

Normal proofs enjoy the *subformula property*: Every formula that appears in the derivation is either a subformula of the conclusion or of any of the assumptions. In our non-normal proof above, $B \Rightarrow (B \wedge A)$ does not occur in $(A \Rightarrow B) \Rightarrow A \Rightarrow B \wedge A$, A, or $A \wedge B$. Tranforming a proof as we did above is called *proof normalization*. It has been shown by Prawitz (for an exposition see [32]), that every proof can be normalized, and that the sequence of normalizing steps is finite. This latter property is known as *strong normalization*.

It is the essence of the Curry-Howard isomorphism that proof normalization corresponds to β-reduction in the λ-calculus[3]. We can think of '$\rightsquigarrow$' as defining the same relation between proofs as $\underset{\beta}{\rightarrow}$ does between terms. Here, we will keep $\rightsquigarrow$ to mean "normalizes to" or "reduces to"; we will regard terms as an alternative, more compact way, of writing proofs.

If we 'decorate' the case for the implication with terms, we obtain the following transformation on terms:

$$\dfrac{\dfrac{\begin{array}{c} x : A \vdash x : A \\ \vdots \\ \Gamma, x : A \triangleright M : B \end{array}}{\Gamma \triangleright \lambda x{:}A.M : A \to B} \to\text{-I} \qquad \begin{array}{c}\vdots \\ \Gamma \triangleright N : A\end{array}}{\Gamma \triangleright (\lambda x{:}A.M)\, N : B} \to\text{-E} \quad \rightsquigarrow \quad \begin{array}{c} \vdots \\ \Gamma \triangleright N : A \\ \vdots \\ \Gamma \vdash M[x := N] : B \end{array}$$

where $M[x := N]$ is our notation for the meta-operation of *substitution* of N for the free occurences of the variable x in M. Since assumptions are labelled with variables in terms,

[3]Usually, β-reduction corresponds to the case for $\to$ only. We will generalize this notion to include the reductions for $\wedge$ and $\vee$.

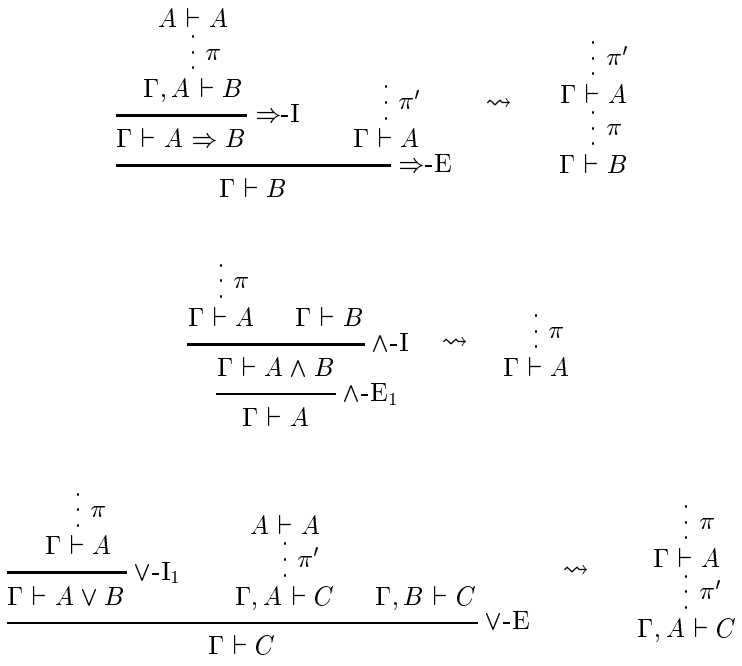

Figure 2.3: Key cases for proof normalization

one should verify that all the variables are distinct. A less strong convention is normally adopted in the context of the λ-calculus, as introduced in the next chapter.

The reductions suggested by annotating all the other cases are

$$\begin{array}{rcl}
(\lambda x{:}A.M)\,N & \rightsquigarrow & M[x := N] \\
\mathsf{fst}\,\langle M, N\rangle & \rightsquigarrow & M \\
\mathsf{snd}\,\langle M, N\rangle & \rightsquigarrow & N \\
\mathsf{case}\ (\mathsf{inl}\,M)\ \mathsf{of}\ \mathsf{inl}\,x \to N\,;\mathsf{inr}\,y \to P & \rightsquigarrow & N[x := M] \\
\mathsf{case}\ (\mathsf{inr}\,M)\ \mathsf{of}\ \mathsf{inl}\,x \to N\,;\mathsf{inr}\,y \to P & \rightsquigarrow & P\,[y := M]
\end{array}$$

We know, from the λ-calculus, that $\rightsquigarrow$ is confluent, which means that normal forms are unique. By Prawitz results, we know that the simple typed λ-calculus shown here satisfies *subject reduction*, since $\rightsquigarrow$ transforms proofs of a formula into proofs of the same formula (as we should expect); in λ-calculus terms, this means that reduction preserves the type of the term. It is also strongly normalizing, i.e., there is no infinite sequence of reductions; in other words, all terms have a normal form. The restrictions imposed by the typing rules on terms forbid, for example, terms of the form $\lambda x{:}A.x\,x$, which are the essence of the representation of recursion in the calculus! But this is not the end of the world. Typed functional languages abound, and are based on the theories described here. A general recursion construct is included in these languages as an extension. For a discussion on the computing capabilities of the theory refer to Plotkin [31].

2.4 Cut elimination

In this section we will look a bit closer at substitution. In the previous section, we introduced the notion of normalization, which consists in eliminating logical detours. This elimination is 'implemented' by substituting subproofs for assumptions in a particular way.

There exists a rule, which plays a major role in sequent calculus, that 'internalizes' the notion of substitution. In general this rule is shown to be superflous, in the sense that it can be defined in terms of the other logical rules. This result is known in Proof Theory as the *cut-elimination theorem* because the rule, called the Cut rule,

$$\frac{\Gamma, A \vdash B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,\text{Cut}$$

explicitly records the case where an assumption, the *cut-formula*, is not needed in a derivation, since a proof of it is available, i.e., A above. The rule may be read as follows. If we can prove A, and by adding A to our original set of assumptions, we can prove B, then we may conclude that B can be proved from the original set without A. This is because, technically, we do not need A as an assumption. If we think of A as a lemma that is proved separately, what this means is that anywhere in the proof where we need to refer to the lemma, we can write a copy of its proof. Although this is not recommended as every day mathematical practice (this is one of the reasons why we use lemmas on the first place!), it is logically valid. Let us say that our assumption is the variable x, our main proof is encoded in the term M, and N is our lemma, then $M[x := N]$ corresponds to our transformed proof where all occurences of x have been replaced by a (copy of) N:

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash N : A}{\Gamma \vdash M[x := N] : B}\,\text{Cut}$$

The cut-elimination theorem, i.e. the fact that the Cut rule is a derived rule, also holds for IPL (as one expects). The idea behind the proof of cut-elimination is very simple: The rule can be eliminated by showing that it commutes with all the logical rules, so that it can be propagated upwards in the derivation tree until the assumptions (the leaves) are reached; if the assumption is A, we replace it with its proof (the premise on the right of the Cut); otherwise, we can eliminate it altogether. The key cases are shown in Figure 2.4. As a convention we decided to subscript the relation $\rightsquigarrow$ with the letter σ (that stands for "substitution") to distinguish it from our previous relation of normalization that parallels β-reduction.

By the Curry-Howard correspondence, these cases may be characterized as reduction rules in the typed λ-calculus:

$$\begin{aligned}
x[x := N] &\underset{\sigma}{\rightsquigarrow} N \\
y[x := N] &\underset{\sigma}{\rightsquigarrow} y \\
(\lambda y : B.M)[x := N] &\underset{\sigma}{\rightsquigarrow} \lambda y : B.M[x := N] \\
(M_1\, M_2)[x := N] &\underset{\sigma}{\rightsquigarrow} (M_1[x := N])\,(M_2[x := N])
\end{aligned}$$

Adding the Cut rule in the calculus corresponds to adding substitution as an explicit construct, and the above reduction rules define the notion of reduction that describes how to actually perform the operation of substitution in a stepwise fashion. To mimic one step of β-reduction, we will need many more steps in the 'refined' version with the substitution explicit, i.e. one step to introduce the explicit substitutions, and some more to eliminate it[4].

[4]The untyped version of this calculus was already introduced by Rose [33], and will be discussed in chapter 4.

$$\dfrac{\dfrac{}{\Gamma, x : A \vdash x : A}\,\text{Axiom} \qquad \Gamma \vdash N : A}{\Gamma \vdash x[x := N] : A}\,\text{Cut} \quad \overset{\rightsquigarrow}{\sigma} \quad \Gamma \vdash N : A$$

$$\dfrac{\dfrac{}{\Gamma, x : A \vdash y : B}\,\text{Axiom} \qquad \Gamma \vdash N : A}{\Gamma \vdash y[x := N] : B}\,\text{Cut} \quad \overset{\rightsquigarrow}{\sigma} \quad \dfrac{}{\Gamma \vdash y : B}\,\text{Axiom} \quad (\Gamma \equiv \Gamma', y : B)$$

$$\dfrac{\dfrac{\Gamma, x : A, y : B \vdash M : C}{\Gamma, x : A \vdash \lambda y{:}B.M : B \to C}\,{\to}\text{-I} \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda y{:}B.M)[x := M] : B \to C}\,\text{Cut}$$

$$\overset{\rightsquigarrow}{\sigma}$$

$$\dfrac{\dfrac{\Gamma, y : B \vdash N : A \quad \Gamma, y : B, x : A \vdash M : C}{\Gamma, y : B \vdash M[x := N] : C}\,\text{Cut}}{\Gamma \vdash (\lambda y{:}B.M[x := N]) : B \to C}\,{\to}\text{-I}$$

$$\dfrac{\dfrac{\Gamma, x : A \vdash M_1 : B \to C \quad \Gamma, x : A \vdash M_2 : B}{\Gamma, x : A \vdash (M_1\, M_2) : B}\,{\to}\text{-E} \qquad \Gamma \vdash N : A}{\Gamma \vdash (M_1\, M_2)[x := N] : B}\,\text{Cut}$$

$$\overset{\rightsquigarrow}{\sigma}$$

$$\dfrac{\dfrac{\Gamma, x : A \vdash M_1 : B \to C \quad \Gamma \vdash N : A}{M_1[x := N] : B \to C}\,\text{Cut} \qquad \dfrac{\Gamma, x : A \vdash M_2 : B \quad \Gamma \vdash N : A}{M_2[x := N] : B}\,\text{Cut}}{\Gamma \vdash (M_1[x := N])\,(M_2[x := N]) : B}\,{\to}\text{-E}$$

Figure 2.4: Cut elimination for $\text{IPL}^{\lambda}_{\to}$

We will see the relevance of this view in the context of abstract machines and Linear Logic.

2.5 Structural rules

As we said at the very beginning of this chapter, there is not a unique and obvious way of presenting a logic. Different presentations differ in many subtle ways, and owe their origins to different observations and motivations.

We will present a variant here, with the objective of making explicit an aspect of the logic that is always taken for granted in classical and Intuitionistic Logic, but that is crucial in Linear Logic. In the following very simple proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A \wedge B, C \vdash A \wedge B}{A \wedge B, C \vdash A}\wedge\text{-E}_1 \qquad \dfrac{A \wedge B, C \vdash A \wedge B}{A \wedge B, C \vdash B}\wedge\text{-E}_2}{A \wedge B, C \vdash B \wedge A}\wedge\text{-I}}{A \wedge B \vdash C \Rightarrow B \wedge A}\Rightarrow\text{-I}}{\vdash A \wedge B \Rightarrow C \Rightarrow B \wedge A}\Rightarrow\text{-I} \qquad (*)$$

we have *used* the assumption $A \wedge B$ twice, to prove A first, and then B. Also, the assumption C was not used at all. We may use an assumption as many times as we want, even zero times. As Wadler [39] put it very eloquently, "truth is free". In Linear Logic, truth is not free, it has a cost. Assumptions are supposed to be used exactly *once*, unless otherwise stated. Linear Logic will be the matter of chapter 5. For the moment we will concentrate on understanding two rules, formerly studied in the context of sequent calculus, known as Weakening and Contraction, which (explicitly) indicate in the proofs where we are *discarding* an assumption, because we are not planning to use it, or *duplicating* an assumption, because we intend to use it more than once.

Weakening and Contraction are known as *structural rules* because they do not introduce or eliminate any connectives; rather, they affect assumptions only. They are useful when we need to have control on the way assumptions are handled in the derivations.

We will recast the the rules of IPL (only some of them) in such a way that we will be forced to duplicate and discard assumptions to achieve the effect of using an assumption repeatedly in a derivation. The new system appears in Figure 2.5 and includes a modified version of the Axiom rule, the structural rules, $\Rightarrow$-I and a modified $\Rightarrow$-E.

Here, Γ and Γ' are multisets, and Γ, Γ' denotes multiset union. It is needless to say that IPL with the structural rules explicit is equivalent to our old IPL with the rules implicit. To understand how these rules work, let's rewrite $(*)$ in the new system as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A \wedge B \vdash A \wedge B}{A \wedge B \vdash A}\wedge\text{-E}_1 \qquad \dfrac{A \wedge B \vdash A \wedge B}{A \wedge B \vdash B}\wedge\text{-E}_2}{A \wedge B, A \wedge B \vdash B \wedge A}\wedge\text{-I}}{A \wedge B \vdash B \wedge A}\text{Contraction}}{A \wedge B, C \vdash B \wedge A}\text{Weakening}}{A \wedge B \vdash C \Rightarrow B \wedge A}\Rightarrow\text{-I}}{\vdash A \wedge B \Rightarrow C \Rightarrow B \wedge A}\Rightarrow\text{-I}$$

$$\frac{}{A \vdash A}\ \text{Axiom}$$

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \text{Weakening} \qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ \text{Contraction}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-I} \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma' \vdash A}{\Gamma, \Gamma' \vdash B}\ \Rightarrow\text{-E}$$

$$\frac{\Gamma \vdash A \quad \Gamma' \vdash B}{\Gamma, \Gamma' \vdash A \wedge B}\ \wedge\text{-I} \qquad \frac{\Gamma \vdash A \wedge B}{A}\ \wedge\text{-E}_1 \qquad \frac{\Gamma \vdash A \wedge B}{B}\ \wedge\text{-E}_2$$

Figure 2.5: The rules for $\text{IPL}_{\Rightarrow\wedge}$ including structural rules

The formula $A \wedge B$ must be duplicated in order to be used in both derivations of the premises of $\wedge$-I; and C must be discarded since it will not appear as an assumption. Taking an economic-oriented view of assumptions as 'resources' that are consumed, we can say that our previous version of IPL is 'resource unconscious', while the new version is (more) 'resource conscious', because at least we make explicit the use of assumptions. If we did not include Weakening and Contraction we would be forced to use assumptions only once. For example, we could either prove A or B above, but not both. This is because contexts, i.e., Γ and Γ', are *non-shared* in the rules with more than one premise, as opposed to *shared*, where they are carried over each of the subderivations. If we did not use all the assumptions, at the leaves, i.e., at the occurences of the rule Axiom in the proof tree, we are forced to apply Weakening to all the assumptions that were not used. In this way, we keep an explicit history of assumption 'consumption' in a derivation.

Chapter 3

The simple typed λ-calculus

Here we present the simple typed λ-calculus, with the aim of giving the basic concepts and properties needed, and establishing the notation and terminology we will use throughout this thesis. We will concentrate exclusively on defining β-reduction and motivating the call-by-name strategy, as well as providing some basic notions concerned with the family of abstract machines we are going to work with. We will not discuss extensionality here. For a more detailed and general presentation the reader is referred to Barendregt's encyclopaedic treatment of the untyped λ-calculus [5]; for the typed version see [6] and Hindley and Seldin [23].

The term calculus we will consider corresponds to the term-assignment described in the previous, where we will concentrate only in the implicational fragment, what we called $\mathrm{IPL}^{\lambda}_{\rightarrow}$. Types will correspond to formulas in this fragment and terms to proofs. We will also recast the notion of reduction and provide all the necessary auxiliary notions that will be needed for later chapters.

The typed system described belongs to the class of Church systems, where every term and all its subterms have a unique type, as opposed to Curry systems, also known as systems of *type assignment*, where terms do not carry explicit annotations, and thus correspond to the terms of the untyped λ-calculus. A given untyped term may have many possible types, or none at all. This raises the notion of a *more general* or *principal type* for a term; see Milner [17].

The reader is assumed familiarity with, at least, the untyped version of the calculus and the style of programming in a typed functional programming language.

3.1 Basic Notions

We begin by giving the syntax of the simple typed λ-calculus.

3.1.1 Definitions (types and λ-preterms) The set of *types*, ranged over by the letters A,B,C is defined by

$$A, B ::= G \mid A \rightarrow B,$$

where G is a set of *ground* (or *constant*) types. Intuitively, ground types correspond to the basic data types in functional languages (for example, integers, booleans, etc.), and $A \rightarrow B$ correspond to the type of functions with domain A and codomain B. We use the same notational conventions as for IPL formulas.

The set of *preterms* is ranged over by the letters M,N,P and is defined inductively by

$$M, N ::= x \mid \lambda x{:}A.M \mid MN,$$

where the letters x,y,z,u,v and w range over an infinite set of variables. As usual we use parenthesis to indicate precedence. Application is left-associative, $M\,N\,P$ is then $(M\,N)\,P$; the 'λ' operator has lower precedence than application and is right-associative; we write $\lambda x{:}A.\lambda y{:}B.M\,N\,P$ for $\lambda x{:}A.(\lambda x{:}A.(M\,N)\,P)$. □

We can think of λ-preterms as proof 'candidates'; for example, $\lambda x{:}A.x\,x$ is a preterm, but does not encode an intuitionistic proof. From a programming point of view, preterms are programs that need to be checked for type consistency. But, before restating and expanding our notions of typing, we will state some auxiliary definitions.

3.1.2 Definitions (preterm notions) The *free variable* set of a preterm M, $\mathrm{FV}(M)$, is defined by induction on the structure of M, as follows:

$$\begin{aligned}\mathrm{FV}(x) &= \{x\}\\ \mathrm{FV}(\lambda x{:}A.M) &= \mathrm{FV}(M) \setminus \{x\}\\ \mathrm{FV}(M\,N) &= \mathrm{FV}(M) \cup \mathrm{FV}(N)\end{aligned}$$

We say that a preterm M is *closed* if $\mathrm{FV}(M) = \varnothing$.

The dual notion, the *bound variable* set, $\mathrm{BV}(M)$, is defined as

$$\begin{aligned}\mathrm{BV}(x) &= \varnothing\\ \mathrm{BV}(\lambda x{:}A.M) &= \mathrm{BV}(M) \cup \{x\}\\ \mathrm{BV}(M\,N) &= \mathrm{BV}(M) \cup \mathrm{BV}(N)\end{aligned}$$

The operation of *substitution* of N for all free occurences of a variable x in M, written $M[x := N]$, is defined by induction on M as follows

$$\begin{aligned}x[x := N] &\equiv x\\ y[x := N] &\equiv y, \text{ if } x \not\equiv y\\ (\lambda x{:}A.M)[x := N] &\equiv \lambda x{:}A.M\\ (\lambda y{:}A.M)[x := N] &\equiv \lambda z{:}A.M[y := z][x := N],\\ &\qquad \text{if } x \not\equiv y \text{ and } z \notin \mathrm{FV}(N) \cup \{x, y\} \qquad (*)\\ (M_1\,M_2)[x := N] &\equiv (M_1[x := N])\,(M_2[x := N])\end{aligned}$$

Substitution as a meta-operation on preterms verify the following *substitution lemma*:

$$M[x := N][y := P] \equiv M[y := P][x := N[y := P]]$$

We will refer to the special case $M[x := y]$ as the operation of *renaming* in M the occurences of the variable x to y. □

In the case for substitution of N in a λ-abstraction (*), suppose that $y \in \mathrm{FV}(N)$, then substitution would not be well-defined, i.e., the free variable y would become bound, 'captured' by the abstraction. Renaming is then necessary to ensure well-definedness in all cases.

Since the revelance of variables names is in the positions they occupy in the structure of a preterm and not in the names themselves, we will assume that the names of the variables in a term are such that (*) can be restated as $\lambda y{:}A.M[x := N]$. This will simplify our exposition. It is common, for example, to assume what is known as the *Variable Convention*. This convention states that (in any mathematical context) free variables are chosen to be different from bound variables, thus $y \in \mathrm{FV}(N)$ could never happen, since y is a bound variable and by the convention must be different to all the free variables in N. This simplify most of the proofs greatly. In practice, either variables will be renamed before the evaluation of a program, or during substitution to avoid variable capture. Note that in logic, variables stand for assumption labels, and thus are assumed to be different from any other variables. This stronger condition also avoids us to consider the case $(\lambda y{:}A.M)[x := N]$ where $x \equiv y$.

We will freely disregard differences in bound variable names, in the sense suggested by the following definition.

3.1.3 Definition (syntactic equality) Two preterms M and N are syntactically equal if they are α-equivalent, namely if they differ only in the names of their bound variables. We will write $M \equiv N$ instead of the more explicit $M \equiv_\alpha N$. □

3.1.4 Definitions (λ-terms) Let's consider first a *context* as a set of *declarations* written in the form

$$\Gamma ::= x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n,$$

where $n \geq 0$, and all x_i, $1 \leq i \leq n$, are pairwise distinct. This condition is necessary for contexts to be *well-formed*. The comma ',' notation is due to Prawitz and in this context it is used to mean set-theoretic union. We will write $\mathrm{dom}(\Gamma)$, the *domain* of Γ, to denote the set $\{x_1, x_2, \ldots, x_n\}$. In the typing rules, contexts are assumed to be well-formed, so if we write $\Gamma, x : A$ and Γ, Γ', then $x \notin \mathrm{dom}(\Gamma)$ and $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \varnothing$, respectively.

A *typing judgement* is a sequent

$$\Gamma \triangleright M : A$$

read "M has type A in context Γ". Intuitively, Γ contains the declarations of the free variables of M.

Finally, the set of *λ-terms* Λ are the preterms M such that for some context Γ and type A, the sequent $\Gamma \triangleright M : A$ can be derived using the following *typing*, or *type-checking*, rules:

$$\frac{}{\Gamma, x : A \triangleright x : A}\ \mathrm{Ax}$$

$$\frac{\Gamma, x : A \triangleright M : B}{\Gamma \triangleright \lambda x{:}A.M : A \to B}\ {\to}\text{-I} \qquad \frac{\Gamma \triangleright M : A \to B \quad \Gamma \triangleright N : A}{\Gamma \triangleright M\,N : B}\ {\to}\text{-E}$$

We will write Λ° for the set of closed λ-terms. □

We have seen already some example derivations in the previous chapter. In functional programming, a program is always a closed expression. We are interested only in computing values of basic data types, hence free variables are not considered values, and are not meaningful as program results. Bound variables should be regarded as formal function arguments, which will be substituted for expressions that denote objects of some basic data type.

We now state two useful properties of typing judgements.

3.1.5 Propositions (context properties) *The type of a term M depends only on the types of the free variables syntactically present in M. This is given by* context weakening,

$$\Gamma \triangleright M : B \text{ implies } \Gamma, x : A \triangleright M : B,$$

and context strengthening,

$$\Gamma, x : A \triangleright M : B \text{ implies } \Gamma \triangleright M : B \quad \text{if } x \notin \mathrm{FV}(M).$$

□

3.1.6 Remark (variable binding) It is easy to show by induction that if $\Gamma \triangleright M : A$, then (a) $\mathrm{dom}(\Gamma) \cap \mathrm{BV}(M) = \varnothing$, so $\mathrm{FV}(M) \cap \mathrm{BV}(M) = \varnothing$ (i.e., $\mathrm{dom}(M) \supseteq \mathrm{FV}(M)$), and (b) if $\lambda x{:}A.N$ is some subterm of M, then $x \notin \mathrm{BV}(M)$. These follows from the fact that to type a function abstraction $\lambda x{:}A.N$, x must not occur already in Γ; this condition is necessary to prevent ill-formed contexts, i.e., contexts with repeated variables. Moreover, if some subterm of N is a function abstraction, by this same fact, its bound variable must necessarily be distinct to x. This conditions imposed on bound and free variables are sufficient to avoid variable capture during reduction: for any redex $(\lambda x{:}A.M)\,N$, (a) implies that x is not free in N.

Note that the preterm $\lambda x{:}A.f\,(\lambda x{:}B.x)\,x$ breaks context well-formedness; hence, the second occurence of x is renamed to avoid the clash with the first (outermost) x. An alternative approach, which is generally used consists in regarding a context as a stack rather than a set. In this way, variables that appear in terms are bound to the nearest λ with the same name, which appear in the stack also closer to the top. This mechanism coincides to the idea of local scope in programming languages, where a variable is bound to the innermost declared occurence. Note also that this mechanism may be used also when translating to De Bruijn indices. □

3.2 Reduction and Strategies

The notion of reduction on λ-terms we will consider in this thesis is β-reduction.

3.2.1 Definitions (β-reduction) A *notion of reduction* is a binary relation defined on λ-terms, written as an arrow '$\to$', possibly subscripted with the name of the reduction. We will write $M \to N$, "*M contracts to N*", as an abbreviation for $(M, N) \in \to$. Here, M is called the *redex* and N the *contractum*.

The usual notion of β-reduction is defined for all terms M and N as

$$(\lambda x{:}A.M)\,N \to M[x := N] \qquad (\beta)$$

We will consider other notions of reduction. In general, we will refer to a notion of reduction as β-reduction if it corresponds to a normalization step in the underlying logic. For example, the above notion corresponds to the normalization case for implication (cf. §2.3).

We say that "*M reduces to N*" ("*in one step*") if N is obtained by contracting any redex of M. One-step reduction is defined by taking the *contextual* (or *compatible*) closure of $\to$, which is given by the following inference rules:

$$\frac{M \to M'}{\lambda x{:}A.M \to \lambda x{:}A.M'} \qquad (\xi)$$

$$\frac{M \to M'}{M\,N \to M'\,N} \qquad (\mu)$$

$$\frac{N \to N'}{M\,N \to M\,N'} \qquad (\nu)$$

Many-step reduction, written '$\twoheadrightarrow$', is obtained by taking the reflexive and transitive closure of $\to$, and is given by the following rules:

$$\frac{M \to N}{M \twoheadrightarrow N} \qquad \frac{}{M \twoheadrightarrow M} \qquad \frac{M \twoheadrightarrow N \quad N \twoheadrightarrow P}{M \twoheadrightarrow P}$$

If a term M does not contain any redex (of the notion of reduction considered) we say that M is in *normal form*, for which we will write $\mathrm{nf}(M)$; where we want to be explicit we will write, for example, $\beta\text{-nf}(M)$. We will also use the notation $\downarrow(M)$ to denote the normal form of M (a term N such that $M \twoheadrightarrow N$ and $\mathrm{nf}(N)$); or, to be more explicit, $\downarrow_\beta(M)$, for example. □

3.2.2 Definitions (properties) We will consider the following properties:

- A reduction relation $\to$ satisfies *subject reduction* if $\Gamma \triangleright M : A$ and $M \to N$ implies $\Gamma \triangleright N : A$, hence typing is preserved by reduction.
- A one-step reduction relation $\to$ is said to be *confluent* (or to satisfy the *Church-Rosser property*), if whenever $M \twoheadrightarrow N_1$ and $M \twoheadrightarrow N_2$, then there is a P such that $N_1 \twoheadrightarrow P$ and $N_2 \twoheadrightarrow P$.
- A term M is *strongly normalizing* if there in no infinite reduction sequence $M \to M_1 \to M_2 \to \ldots$, i.e., reduction terminates. We will say that a reduction relation is strongly normalizing if all terms are strongly normalizingwith respect to that reduction relation.

3.2.3 Proposition *The β-reduction relation satisfies subject reduction, confluence, and strong normalization.*

Proof. See Hindley and Seldin [23]. □

From confluence and strong normalization it follows that all β-strategies will terminate in a unique normal form.

Having settled down basic terminology and notation, we will now motivate a particular strategy for reduction of terms. We regard a *strategy* as a deterministic restriction of the notion of reduction under consideration (with some extra properties). Since we advocate the lazy style of functional programming, we will consider a *weak normal order* strategy, which corresponds to *call-by-name* as it appears in Abramsky [2]. For brevity, we will drop the type annotations from the terms, as we will do wherever we start discussing reduction strategies.

Let's consider the reduction system with (β) as the only axiom, and (ξ), (μ), and (ν) as reduction rules. This system can be thought as the most liberal; at any given time, any β-redex can be chosen for reduction.

Weak strategies are motivated from the fact that functional languages do not reduce to normal form; functional abstractions, for instance, are reduced only if they are applied,

otherwise they are considered as values. This motivates the study of a 'weaker' notion of value, *weak-head* normal forms (WHNF); these are terms of the form

$$V ::= \lambda x.M \mid x\, M_1 \ldots M_n$$

Since we are considering only closed terms, we can disregard the second form (i.e., it is open). Weak normal forms can be reached by dropping the (ξ) rule, which allows reduction to occur inside λ's. The main drawback of the resulting system is that it lacks confluence. system lacks the confluence property. For example, for $(\lambda x.\lambda y.x)\,(\mathbf{I}\,\mathbf{I})$ (the term $\mathbf{I}$ is the identity function), we may choose to reduce $(\mathbf{I}\,\mathbf{I})$ first, and then reduce again to obtain $\lambda x.\lambda y.\mathbf{I}$; alternatively we may reduce the outer β-redex first to obtain $(\lambda x.\lambda y.\mathbf{I}\,\mathbf{I})$. In any case, since reduction was forbidden under λ's the resulting terms will not have a common normal form. This property is recovered with explicit substitutions; see Curien et al. [13].

The lack of confluence is certainly not dramatic if we just want to consider a particular (deterministic) strategy. Non-determinism appears from having two rules for application, (μ) and (ν). In practice, either *leftmost-outermost* or *rightmost-innermost* strategies are considered. The former reduces the leftmost-outermost redex, and corresponds to normal order. The latter corresponds to applicative order. Applicative order (call-by-value) restricts the application of the β-rule, $(\lambda x.M)\,N \to M[x := N]$, to the case where N is already in WHNF, as opposed to normal order (call-by-name) which forbids the reduction of the argument before doing the substitution. Weak-head normal forms in normal order for closed terms can be computed by dropping the (ν) rule. Since the untyped λ-calculus is not strongly normalizing, a well-known standardization result shows that if a term has a WHNF, it can be found by leftmost-outermost weak reduction; see Barendregt [5] for details.

3.2.4 Definition (call-by-name) We will understand *call-by-name* for the simple typed λ-calculus as weak leftmost-outermost β-reduction for closed terms, generated by the inference rules:

$$\frac{}{(\lambda x.M)\,N \to M[x := N]} \qquad (\beta)$$

$$\frac{M \to M'}{M\,N \to M'\,N} \qquad (\mu)$$

□

Lazy strategies correspond to the call-by-need paradigm, which is the same as call-by-name but with the extra requirement that if the substituted argument is evaluated (if it does) then it should be evaluated *only once*. This may suggest some form of sharing of argument evaluation, as we will explain in §7.3. Until then, we will restrict ourselves to call-by-name.

3.2.5 Remark (values) What are regarded as values in the calculus is dictated by the evaluation paradigm of the language under consideration. If we restrict our context to just functions, values in both call-by-name and call-by-value strategies are terms in WHNF. The idea of call-by-name is that reduction should not compute more information than that strictly necessary. From a logical viewpoint, constructors are already values, i.e., terms that correspond to an introduction rule, and only a destructor-constructor form triggers reduction. For example, if we consider pairs as an extension, terms of the form $\langle M, N\rangle$ would be already in normal form; and a projection, $\mathsf{fst}\,M$ (for instance), would immediately trigger reduction on M to identify a constructor of the preceding form. □

3.3 Towards abstract machines

We will not intend to define abstract machines formally; we will be more pragmatic in this sense. It is a well-known fact that abstract machines, like the one presented here, and in the next chapters, can be obtained from a reasonable[1] deterministic strategy. We will not get to that level of detail; the reader will immediately see that the gap between the reduction rules and the actual machine is so narrow that one system could be put in terms of the other by considering a different representation. We borrowed the idea of presenting different machines, using names and De Bruijn indices, as stepwise refinements of simpler machines, from Sestoft [35].

An abstract machine may be formalized as a *transition system* $(\Sigma; \to; F)$, where Σ is a set of states, or *configurations*, $\to$ is a transition function, and F is a set of final states. Before explaining more concretely what these mean, let's first look at a very simple 'abstract machine' to reduce λ-terms using the above strategy.

A configuration will be a pair (M, S), for M a term, and S a stack of terms, represented as a sequence.

3.3.1 Notation (sequences) We will write *sequences* as $\langle M_1, \ldots, M_n \rangle$. $M \cdot S$ stands for the sequence obtained by prefixing a term M into S. We will write an empty sequence as ϵ or $\langle\rangle$, depending on the context. Note that $\langle M_1, \ldots, M_n \rangle$ can be regarded as an abbreviation for $M_1 \cdot \cdots \cdot M_n \cdot \epsilon$.

Also, in the context of abstract machines, when sequences denote *stacks*, we will use '::' for prefixing ("push"). □

Initially, evaluation starts with an empty stack; for a given term M, the initial configuration is (M, ϵ). If M is an application term $(N\,P)$, (μ) suggests reducing N first. Indeed, if a term is not already in WHNF it will be of the form $(\lambda x.M)\,N_1\,\ldots\,N_m$, for $m \geq 1$, and then we would have to 'search down' the left-hand side of the application until a function abstraction is found. This observation motivates the use of the stack. While searching down for the redex, the N_i are pushed onto the stack. This transition sequence stops when a function abstraction is hit, at which point the (unevaluated) argument will already be at the top of the stack. Therefore, we will have two transition *schemes*, or rules:

$$(M\,N, S) \to (M, N \cdot S) \qquad \text{(Search)}$$

$$(\lambda x.M, N \cdot S) \to (M[x := N], S) \qquad \text{(Reduce)}$$

The (Search) transition rule corresponds to (μ) (more or less), and (Reduce) corresponds to (β) in an obvious way. The sloppiness of this machine is naturally the fact that substitution is left unspecified! We will consider this more deeply in the next chapter.

Note that one reduction step in the calculus correspond to several reduction steps in the abstract machine; in fact, a single reduction in the calculus is implemented as n instances of (Search), where n is the depth of the redex in the term tree (cf. §7.2.2), followed by an instance of (Reduce). In general, evaluation consists on an *identification* phase, that looks for the redex, and a *reduction* phase, which is the one that actually performs some computation.

It is easy to see, that the only final states of our transition system are the configurations given by the scheme $(\lambda x.M, \epsilon)$.

[1] With 'reasonable' we mean that the strategy must satisfy certain properties other than just being deterministic; refer to [21].

3.3.2 Remark (pragmatics of representation) The simple machine of this section understands code as an internal tree-like representation of a term, like, for example, the one used for the implementation of the xLIN machine that appears in §7.2.2.

For the transition system to serve as a realistic model of time complexity, the overhead necessary to determine the next transition step should be realizable in a real implementation in constant time. Note that for our simple machine it relies on testing the state of the stack and the tag of the root cell of the term tree. This holds for all the machines that appear in this thesis. □

3.3.3 Remark (operational semantics) A strategy not only determines a particular order of evaluation, but also gives the operational semantics of the language under consideration. Both lazy and eager (call-by-name and call-by-value) functional languages are based on the same calculus, but they imply different programming styles since their operational behaviour is quite different. Formally and practically, we cannot do without specifying the operational semantics of a language. For completeness, we will present it in the style of Plotkin, also known as *natural semantics*. For call-by-name we have the following set of rules, which correspond to the *lazy λ-calculus* of Abramsky [2]:

$$\frac{}{\lambda x.M \Downarrow \lambda x.M}$$

$$\frac{M \Downarrow \lambda x.M' \quad M'[x := N] \Downarrow V}{M\,N \Downarrow V}$$

The rules define the evaluation relation, $M \Downarrow N$, read "M evaluates to N", where N is in WHNF. This rules can be obtained by restricting the reflexive and transitive closure of the reduction relation defined by the rules for call-by-name to the case where N is a value.

We could have derived our arguments for the derivation of our abstract machines from an operational semantics like the one just given. A particular advantage is that the formalism allows for extensions to be added in a very simple way. The most impressive of the examples is the natural semantics for the ML language [29]. □

3.4 De Bruijn's notation

The fact that bound variable names can be arbitrary, left solely to the taste of the writer, leads to a definition of substitution that requires renaming of bound variables in the context of a function abstraction. Bound variables can be renamed on the grounds that the name in itself is irrelevant; it is used merely to relate each occurence with an outer 'λ' that has the same name attached to it.

An alternative to names was proposed by N.G. De Bruijn [18] for the project AUTOMATH, an automatic theorem prover. A variable is represented as an integer n, called an *index*, that relates it with the λ operator that occurs n levels up in the tree representation of the term. This hints at the convention that indices start at 0. The simplest of the examples is the combinator **I**, which is simply written $\lambda 0$. (Note that we do not attach anything to function abstractions.) We will define the *namefree* preterms, λ_{NF}-preterms, as those generate inductively by

$$M, N ::= n \mid \lambda M \mid M\,N,$$

where $n \in \mathbb{N}_0$. Other examples are the combinators **K** and **S** which are written $\lambda\lambda 2$, and $\lambda\lambda\lambda 3\,1\,(2\,1)$, respectively.

Theoretically using namefree terms avoids the incoveniences of renaming. Note that α-equivalent preterms correspond to the same namefree preterm. The disadvantage is that indices are unreadable; also, the meta-operation of substitution does not get particularly more interesting (i.e., note that indices need to be adjusted if they come under the scope of a function abstraction). Since we are not going to perform complex indices manipulations the unacquainted reader is referred to De Bruijn's original paper. Our interest in indices relies on the fact that they yield a more efficient treatment of environments.

3.4.1 Proposition (translation to $\lambda_{\mathbf{NF}}$-preterms) *The λ-preterms can be translated to namefree preterms using the following $[\![\cdot]\!]\,\phi$ translation; ϕ is a stack of bound variables which records the name; the scope is given by its location in the stack. Again, we focus on closed terms:*

$$\begin{array}{rcl} [\![x]\!]\,(x \cdot \phi) & = & 0 \\ [\![x]\!]\,(y \cdot \phi) & = & 1 + [\![x]\!]\,\phi \ \textit{if}\ x \not\equiv y \\ [\![\lambda x{:}A.M]\!]\,\phi & = & \lambda A([\![M]\!]\,(x \cdot \phi)) \\ [\![M\ N]\!]\,\phi & = & ([\![M]\!]\,\phi)\,([\![N]\!]\,\phi) \end{array}$$

□

The expression $x \cdot \phi$, pushes a variable on top of the stack (so it gets index 0). Note that the translation is type preserving.

Chapter 4

Explicit substitutions

In this chapter we follow the tradition of Abadi et al. [1], and 'refine' the notion of reduction of the typed λ-calculus by adding substitution as an explicit construct. In this way, we obtain a reduction calculus where a reduction step corresponds closely to a computational step as it would be done by a 'real' computer, or at least, by an environment machine, which is the main focus of this thesis.

The objective we pursue is not to offer (yet) another calculus of explicit substitutions, but to give the motivations of the calculus we have chosen: a subcalculus of the original $\lambda\sigma$ studied by Curien et al. [13], $\lambda\sigma_{cw}$ (conditional weal $\lambda\sigma$), which considers *weak* reduction in the context of *closures*. This restrictions are reasonable for our present purposes, which confine ourselves to environment machines.

We have seen in §3.2 that weak reduction can be obtained by dropping the (ξ) rule, i.e., that allows reduction inside function abstractions. The drawback is that the resulting calculus is not confluent, since β-redexes may be captured inside abstractions. This problem is remedied by adding explicit substitutions.

There are too many possible ways of adding explicit substitutions. (A survey through some of the explicit substitution calculi was given by Lescanne in [27].) As an introduction, we begin by looking at a typed version of λx, the simplest of the calculi with explicit substitutions, which was formerly studied by Bloo and Rose [10]. This calculus may be regarded as 'canonical', in the sense that substitution appears by considering the Cut rule in Intuitionistic Logic (see §2.4), annotated as follows:

$$\frac{\Gamma, x : A \triangleright M : B \qquad \Gamma \triangleright N : A}{\Gamma \triangleright M\langle x := N\rangle}\ \text{Sub}$$

where $\langle x := N\rangle$ now appears as a new term of the calculus, typed as given. The reduction rules for propagating substitution are given by the cases of cut elimination. In this way, many properties follow from proof-theoretic results. In the sequel, we will refer to this as the 'naïve approach'.

We will then proceed as follows.

Firstly, we will consider extending λx with *parallel* substitutions or *environments*, which generalize $\langle x := M\rangle$ by allowing multiple bindings, i.e., $\langle x_1 := M_1, x_2 := M_2, \ldots x_n := M_n\rangle$. We will observe that this is convenient if we want to address (properly) *delayed* substitutions in environment machines.

Secondly, we introduce $\lambda\sigma_{cw}$, and note that environment machines treat terms and environments separately, as a *closure* $M\langle\sigma\rangle$, where σ is an environment containing the bindings

of the free variables in M (hence the name "closure"). This hints at the fact that no operator for composing substitutions is needed in a weak setting (and with closed terms). Function application introduces a new binding in the environment, and the evaluation of a variable accesses the environment. This is a (partial) description of the working principle behind Krivine's abstract machine (KAM) which we present in two versions: with names and with De Bruijn indexes, the latter yielding a more efficient treatment of environments. KAM is 'derived' by considering a call-by-name strategy.

Thirdly, and finally, we informally remark on how a call-by-need version may be obtained.

4.1 The naïve approach

In §2.4 we observed that proof normalization may be decomposed into two normalization relations, one that eliminates a detour and introduces a cut, and another one that eliminates the cut by propagating it up the prooftree to the assumptions, where it is finally replaced by the cut-formula, or eliminated altogether. Since Cut is interpreted as (explicit) substitution, from the viewpoint of the calculus, an introduction of a cut by contraction of a β-redex corresponds to the rule

$$(\lambda x{:}A.M)\, N \to M\langle x := N\rangle \tag{b}$$

We write $\langle x := N\rangle$ to distinguish it from the usual $[x := N]$ (as well as using a different name for the new rule). The cases of cut elimination, the σ-transformations on proofs, describe how to actually 'implement' substitution by small local transformations.

This simple approach yields the typed version of λx, as summarized in Figure 4.1. The (x?) rules propagate substitutions in the obvious way and correspond to the σ-transformations just mentioned. They may also be obtained by replacing $\equiv$ by $\to$ in the definition of substitution of Definitions 3.1.2, while disregarding syntactic issues like name capture.

4.1.1 Remark (variable binding) Note that the substitution binds a variable like a function abstraction. The same comments we made on variable binding hold for the new construct; for example, the typing system rules out preterms like $M\langle x := N\rangle\langle x := P\rangle$ which would break context well-formedness. Basic notions generalize easily; for free variables, we have that $\mathrm{FV}(M\langle x := N\rangle) = (\mathrm{FV}(M) \setminus \{x\}) \cup \mathrm{FV}(N)$. □

4.1.2 Proposition (properties) *Typed λx satisfies subject reduction, confluence, and strong normalization.*

Proof. Subject reduction follows immediately from IPL; confluence and preservation of strong normalization (PSN) were already proved for the untyped case by Bloo and Rose [10]. □

Note that the refinement of this calculus comes from the fact that one β-step is decomposed into the reduction sequence

$$M \underset{\mathrm{b}}{\to} M' \underset{\mathrm{x}}{\to} \cdots \underset{\mathrm{x}}{\to} N,$$

where M does not contain any explicit substitutions, and $N = \downarrow_{\mathrm{x}}(M')$.

Since with bx-reduction, i.e., the contextual closure of the new notion of reduction $\underset{\mathrm{b}}{\to} \cup \underset{\mathrm{x}}{\to}$, we can choose to reduce any redex, be it a b-redex or an x-redex, several substitutions may

Preterms $M, N ::= x \mid \lambda x{:}A.M \mid M\ N \mid M\langle x := N\rangle$

Typing rules

$$\frac{}{\Gamma, x : A \triangleright x : A}\ \text{Axiom}$$

$$\frac{\Gamma, x : A \triangleright M : B}{\Gamma \triangleright \lambda x{:}A.M : A \to B}\ \to\text{-I} \qquad \frac{\Gamma \triangleright M : A \to B \qquad \Gamma \triangleright N : A}{\Gamma \triangleright M\ N : B}\ \to\text{-E}$$

$$\frac{\Gamma, x : A \triangleright M : B \qquad \Gamma \triangleright N : A}{\Gamma \triangleright M\langle x := N\rangle : B}\ \text{Sub}$$

Reduction rules

$$(\lambda x{:}A.M)\ N \ \to\ M\langle x := N\rangle \qquad \text{(b)}$$

$$x\langle x := N\rangle \ \to\ N \qquad \text{(xv)}$$

$$y\langle x := N\rangle \ \to\ y \quad \text{if } x \not\equiv y \qquad \text{(xvgc)}$$

$$(\lambda y{:}A.M)\langle x := N\rangle \ \to\ \lambda y{:}A.M\langle x := N\rangle \qquad \text{(xab)}$$

$$(M_1\ M_2)\langle x := N\rangle \ \to\ M_1\langle x := N\rangle\ M_2\langle x := N\rangle \qquad \text{(xap)}$$

Figure 4.1: Typed λx

appear in the terms, so we have two consider how substitutions interact. In proof-theoretic terms, we would have to consider the following situation of two cuts:

$$\dfrac{\dfrac{\Gamma, y : B, x : A \triangleright M : C \quad \Gamma \triangleright N : A}{\Gamma, y : B \triangleright M\langle x := N\rangle : C}\,\mathrm{Cut} \qquad \Gamma \triangleright P : B}{\Gamma \triangleright M\langle x := N\rangle\langle y := P\rangle : C}\,\mathrm{Cut}^{(*)}$$

The first cut $(*)$ permutes with the second with the aid of Weakening as follows:

$$\dfrac{\dfrac{\Gamma, y : B, x : A \triangleright M : C \quad \Gamma \triangleright P : B}{\Gamma, y : B, x : A \triangleright M\langle y := P\rangle : C}\,\mathrm{Cut}^{(\dagger)} \qquad \dfrac{\Gamma, y : B \triangleright N : A \quad \Gamma \triangleright P : B}{\Gamma \triangleright N\langle y := P\rangle : A}\,\mathrm{Cut}}{\Gamma \triangleright M\langle y := P\rangle\langle x := N\langle y := P\rangle\rangle : C}\,\mathrm{Cut}^{(\dagger)}$$

This corresponds to the transformation on terms

$$M\langle x := N\rangle\langle y := P\rangle \to M\langle y := P\rangle\langle x := N\langle y := P\rangle\rangle$$

Note that this is the substitution property that appears in Definitions 3.1.2, now a property of the combined reduction relation. The original paper on λx did not include this reduction rule, i.e., composition of substitutions, since it obviously breaks PSN[1]. (There is an infinite reduction sequence of cut permutations, beginning with the redex marked (†).)

4.2 A weak calculus of functional closures

The ancestor of all explicit substitution calculi, $\lambda\sigma$, not only has a way of substituting inside substitutions, i.e., composition as above, but also substitutions can be put in parallel, forming a list of substitutions

$$\langle x_1 := N_1, x_2 := N_2, \ldots, x_m := N_m\rangle.$$

This can be regarded as the 'simultaneous' substitution of N_1 for x_1, N_2 for x_2, etc., up to m. From a logical viewpoint, this substitution is interpreted as a sort of 'multicut', where several assumptions are cut in parallel as follows:

$$\dfrac{\Gamma, x_1 : A_1, \ldots, x_m : A_m \triangleright M : B \qquad \Gamma \triangleright N_1 : A_1 \quad \cdots \quad \Gamma \triangleright N_m : A_m}{\Gamma \triangleright M\langle x_1 := N_1, \ldots, x_m := N_m\rangle : B}$$

Parallel substitutions, or environments, are convenient if we consider strategies that *delay* substitutions, which is the approach implemented by environment machines. To illustrate the idea, let us suppose that we generalize our calculus to include environments, henceforth denoted with the letters σ and τ. The syntax we will use is slightly different:

$$\sigma, \tau ::= \mathrm{id} \mid (x := M) \cdot \sigma.$$

Here, id denotes the empty substitution $\langle\rangle$, and the $\cdot$ operator (called "cons") prefixes a binding to a substitution σ. Hence, we can regard the above syntax for environments as an abbreviation for $(x_1 := N_1) \cdot \cdots \cdot (x_m := N_m) \cdot \mathrm{id}$. Applying substitution to a term will be written $M\langle\sigma\rangle$.

[1] In any case, there are many cases where we need composition of substitutions. For example, it seems that composition is needed to prove the correctness of Cardelli's FAM [11]; see [22].

Since there are many possible strategies, at best we could aim at delaying (avoiding) substitution as much as we can. This is possible, and is exactly how KAM works.

To simplify matters, we will focus on closed terms, and consider a restricted syntax. A computation always starts with a closed term of the form $M\langle\sigma\rangle$, i.e., a closure, where M is a *pure* term (a term without any explicit substitutions). The application of a function abstraction to an argument introduces a binding in the environment; thus, the suggested modified (b) rule[2] is

$$(\lambda x{:}A.M)\langle\sigma\rangle\, N \to M\langle(x := N)\cdot\sigma\rangle \qquad \text{(b')}$$

Note that if we use this reduction rule, then a usual application (which is allowed in the syntax), $(\lambda x{:}A.M)\, N$, would not be a redex. This is not a problem, since we are not interested in being able to write any term that is allowed in the syntax. As we said before, we are supposing that we start reduction with closures, so that function application will always be of the above form; namely, if we start with an application $(M\, N)\langle\sigma\rangle$, the next step is to propagate the substitution (using (xap), but for environments), obtaining $M\langle\sigma\rangle\, N\langle\sigma\rangle$, which matches (b') if M is a function abstraction. The rules (xv) and (xvgc) are rewritten for environments as $x\langle(x := M)\cdot\sigma\rangle \to M$, and $x\langle(y := M)\cdot\sigma\rangle \to y\langle\sigma\rangle$ if $x \not\equiv y$, respectively. Thus, variable reduction accesses the environment.

Figure 4.2 summarizes $\lambda\sigma_{cw}$. Again, we omit all typing information. We also use different names for the rules for reasons of uniformity of notation.

The reduction rules do not reduce or propagate substitutions inside function abstractions, thus all substitutions occur at outer levels. Also, note that reduction is not allowed in the term M of a closure $M\langle\sigma\rangle$. This last restriction is required not to break the condition that turns (b') into an applicable rule! The resulting system is confluent[3]; see [13], Proposition 2.1.1.

4.3 Krivine's abstract machine

A leftmost-outermost strategy can be obtained very easily by dropping the (ν?) rules (which also discard the (sub-?) rules), thus, allowing reductions on the left only. Recall the definition of call-by-name of chapter 3 (Definition 3.2.4). We will write this relation as $\xrightarrow{\text{n}}$.

4.3.1 Remark (restricted syntax) As we said before, the reduction rules cannot reduce any term with substitutions in arbitrary places. This is not a serious restriction. After all, the reason why we wanted to add explicit substitutions on the first place was to be able to get closer to abstract machines, our main concern; particularly, environment machines. The simplicity of the KAM, provides a good starting point to understand more complicated, and sometimes more *ad hoc*, abstract machines.

It is easy to prove by induction that if the syntax is restricted to terms of the form $M\langle\sigma\rangle$ and $M_1\langle\sigma_1\rangle \ldots M_n\langle\sigma_n\rangle$ (where all the M's are pure terms) then the above rules are powerful enough to compute the WHNF's ([13], Proposition 2.2.1). □

Note that the rule (b') can also be written more specifically as

$$(\lambda x.M)\langle\sigma\rangle\, N\langle\tau\rangle \to M\langle(x := N\langle\tau\rangle)\cdot\sigma\rangle,$$

[2]This is precisely where parallel substitutions are needed, and λx is limited.

[3]This may appear a surprise since the weak λ-calculus is not confluent. However, note that redexes cannot be captured under λ's since substitutions are not allowed to cross them.

Axiom rules

$$(\lambda x.M)\langle\sigma\rangle\, N \;\to\; M\langle(x := N)\cdot\sigma\rangle \qquad \text{(b')}$$

$$x\langle(x := M\langle\sigma\rangle)\cdot\tau\rangle \;\to\; M\langle\sigma\rangle \qquad (\sigma\text{varx})$$

$$x\langle(y := M\langle\sigma\rangle)\cdot\tau\rangle \;\to\; x\langle\tau\rangle \quad \text{if } x \not\equiv y \qquad (\sigma\text{vary})$$

$$(M\ N)\langle\sigma\rangle \;\to\; M\langle\sigma\rangle\, N\langle\sigma\rangle \qquad (\sigma\text{app})$$

Contextual rules for terms

$$\frac{M \to M'}{M\ N \to M'\ N}\,(\mu) \qquad \frac{N \to N'}{M\ N \to M\ N'}\,(\nu)$$

$$\frac{\sigma \to \sigma'}{M\langle\sigma\rangle \to M\langle\sigma'\rangle}\,(\nu\text{sub})$$

Contextual rules for substitutions

$$\frac{M \to M'}{(x := M)\cdot\sigma \to (x := M')\cdot\sigma}\,\text{(sub-id)} \qquad \frac{\sigma \to \sigma'}{(x := M)\cdot\sigma \to (x := M)\cdot\sigma'}\,\text{(sub-cons)}$$

Figure 4.2: The conditional weak theory $\lambda\sigma_{cw}$

$$
\begin{array}{rcl}
(\sigma, M\ N, S) & \to & (\sigma, M, N\langle\sigma\rangle :: S) \\
(\sigma, \lambda x.M, N\langle\sigma\rangle :: S) & \to & ((x := N\langle\tau\rangle) \cdot \sigma, M, S) \\
((x := M\langle\sigma\rangle) \cdot \tau, x, S) & \to & (\sigma, M, S) \\
((y := M\langle\sigma\rangle) \cdot \tau, x, S) & \to & (\tau, n, S) \quad \text{if } x \not\equiv y
\end{array}
$$

Figure 4.3: Krivine's abstract machine with names

by the above remark (the same for (μ)). The transition rules of KAM, following the $\xrightarrow{\text{n}}$ strategy are shown in Figure 4.3. The configurations of this machine are triples (σ, M, S), where σ is an environment, M is the term being evaluated, and S is the stack of pending computations. The following lemma makes (even more) explicit the close connection between the calculus and KAM.

4.3.2 Lemma (correctness) *KAM implements the $\xrightarrow{\text{n}}$ strategy; namely, whenever $M \xrightarrow{\text{n}} N$, then $M^\sharp \xrightarrow{KAM}\!\!\!\!\!\rightarrow N^\sharp$.*

Proof. The $(\cdot)^\sharp$ translation encodes terms as machine states, and is defined as follows:

$$
\begin{array}{rl}
(M\langle\sigma\rangle)^\sharp = (\sigma, M, \epsilon) & (*) \\
(M\langle\sigma\rangle\, N_1\langle\tau_1\rangle \ldots N_m\langle\tau_m\rangle)^\sharp = (\sigma, M, N_1\langle\tau_1\rangle :: \cdots :: N_m\langle\tau_m\rangle :: \epsilon) &
\end{array}
$$

From $(*)$, we obtain the initial configuration of the KAM, $(\mathrm{id}, M, \epsilon)$ from a closure $M\langle\mathrm{id}\rangle$. Because we defined the translation on the cases of the restricted syntax, $M^\sharp$ will always be defined. Note also that the final configurations of the machine are given by the schema $(\sigma, \lambda x.M, \epsilon)$, which correspond precisely to the WHNF's $(\lambda x.M)\langle\sigma\rangle$ by the obvious inverse translation. The implication follows easily now. □

The machine above presented, is not the 'real' KAM, but a version with variable names. It is straightfoward to change the representation of variables to De Bruijn indices and give a nameless version. The syntax we'll use for preterms (in an unrestricted setting) is

$$M, N ::= n \mid \lambda A.M \mid M\ N \mid M\langle\sigma\rangle,$$

where n is an index (see §3.4), and where environments are more effectively represented as lists of nameless terms, instead of bindings; the binding is implicit in the position that the term occupies. We assume that the first term in the environment has index 0. For the rules, we will write indices using the syntax $n ::= 0 \mid \mathrm{succ}\, n$. In an implementation, an index $\mathrm{succ}^i\, 0$ is encoded, naturally, as the internal representation of the natural i.

The typing and call-by-name reduction rules, as well as the (nameless) KAM are given in Figure 4.4.

4.3.3 Remark (sharing) A call-by-need KAM can be easily obtained from the call-by-name transition system as follows. Recall that call-by-need requires sharing, in some form, of argument evaluation. KAM accesses the environment with the (Access_0) rule. The term accessed is then evaluated, and it will evaluated as many times as it is actually accessed. This is undesirable in practice.

Typing rules for nameless $\lambda\sigma_{cw}$-preterms

$$\frac{}{\Gamma, A \triangleright 0 : A}\ \text{Axiom} \qquad \frac{\Gamma \triangleright n : B}{\Gamma, A \triangleright \operatorname{succ} n : B}\ \text{Succ}$$

$$\frac{\Gamma, A \triangleright M : B}{\Gamma \triangleright \lambda A.M : A \to B}\ \to\text{-I} \qquad \frac{\Gamma \triangleright M : A \to B \qquad \Gamma \triangleright N : A}{M\ N : B}\ \to\text{-E}$$

$$\frac{\Gamma \triangleright M : A}{\Gamma \triangleright M\langle \mathrm{id}\rangle}\ \text{Id} \qquad \frac{\Gamma, A \triangleright M\langle\sigma\rangle : B \qquad \Gamma \triangleright N : A}{\Gamma \triangleright M\langle N \cdot \sigma\rangle}\ \text{Cons}$$

Reduction rules for call-by-name

$$\begin{aligned}
(\lambda M)\langle\sigma\rangle\, N &\to M\langle N \cdot \sigma\rangle \\
(M\ N)\langle\sigma\rangle &\to M\langle\sigma\rangle\, N\langle\sigma\rangle \\
0\langle M \cdot \sigma\rangle &\to M \\
(\operatorname{succ} n)\langle M \cdot \sigma\rangle &\to n\langle\sigma\rangle
\end{aligned}$$

$$\frac{M \to M'}{M\ N \to M'\ N}$$

Transition rules

$$\begin{aligned}
(\sigma, M\ N, S) &\to (\sigma, M, N\langle\sigma\rangle :: S) && \text{(Push)} \\
(\sigma, \lambda M, N\langle\sigma\rangle :: S) &\to (N\langle\tau\rangle \cdot \sigma, M, S) && \text{(Grab)} \\
(M\langle\sigma\rangle \cdot \tau, 0, S) &\to (\sigma, M, S) && (\text{Access}_0) \\
(M\langle\sigma\rangle \cdot \tau, \operatorname{succ} n, S) &\to (\tau, n, S) && (\text{Access}_n)
\end{aligned}$$

Figure 4.4: Namefree typing rules, call-by-name reduction, and KAM

The 'trick' to adapt KAM to a call-by-need strategy is very simple. When the (Access_0) rule is used, a special mark is pushed into the stack with the address of the closure accessed. When the control, i.e., the term under evaluation, is in WHNF, and the stack contains an update marker, the closure reference is updated with the current value, and evaluation proceeds normally.

Note that in the same way that a term was evaluated as many times as it was accessed, a marker will be pushed in the stack. To overcome this situation, we can attach to a closure in the environment a 'value bit', initially set to 0 (for example) to indicate that the closure in not in WHNF (unless it is). □

Chapter 5

Intuitionistic Linear Logic

The formulation of intuitionistic linear logic in which our work is based is the Dual Intuitionistic Linear Logic (DILL) of Barber [4]. Unlike some of its predecessors, DILL separates syntactically assumptions into two classes: intuitionistic, which may be used any number of times in a derivation (including zero), and linear, which must be used exactly once. Note that in Benton, Bierman, and de Paiva's ILL, for example, all assumptions are linear, and the term 'intuitionistic' is reserved to assumptions of the form $!A$ which can be contracted and weakened like assumptions in intuitionistic logic.

The idea is due to Plotkin, but it seems to have been inspired by Girard [19], who introduced this idea in his Logic of Unity (LU). The explicit distinction between linear and intuitionistic assumptions is realized in DILL by separating assumptions into two contexts[1] (the so called, *dual contexts*). The main visible features, besides the dual contexts (and the fact that we will have two axiom rules and two cut rules), is that we regain the symmetry for the logical rules for the ! connective, which now come naturally as the usual (introduction,elimination) pair. The reader should contrast the simplicity of this presentation with, for example, that of ILL.

Having two rules for ! means that we will not have term constructs for copying and discarding values, i.e., Contraction and Weakening, in the term calculus. There is no need to include them since, as we will see shortly, they are implicit in the structure of the rules, which share the intuitionistic subcontexts.

It should not come as a surprise that, although ILL and DILL are equivalent, they have different operational interpretations. A term calculus for ILL corresponds to a more verbose linear language where the programmer explicitly states where resources will need to be shared (in the presence of duplication) or discarded[2]. Presentations with Contraction and Weakening turn out to be important for models of reference-counting, as it was noted several times. In DILL, we do not have such a fine-grained record of resource usage. For this reason, if we regard DILL as a functional programming language (which is not the view the author wants to advocate), we may say that, compared to its predecessors, it is a language at a higher-level of abstraction: the programmer is not obliged to explicitly write programs using such primitives as copy[3] and discard. The syntax in DILL is less awkward.

[1] Wadler [39] offered a different approach without splitting contexts. Instead, he labels assumptions using the convention that $[A]$ refers to an intuitionistic assumption and $\langle A \rangle$ refers to a linear assumption.

[2] See for example Ian Mackie's LILAC [28].

[3] The author personally prefers share.

This reduction of complexity, greatly helps to gain a deeper understanding of the interplay between intuitionistic and linear assumptions, which is made evident in DILL.

We will now present the syntax of DILL, the natural deduction rules and the term-assignment obtained by application of the Curry-Howard correspondence. We will focus exclusively in the implicational fragment of the logic. We state the necessary properties and then focus our attention on reduction. The dual context give rise to two cut rules, which suggest two notions of substitution: linear and intuitionistic. By the 'single occurrence' property of linear variables we observe that linear substitution suggests a direct and much more efficient implementation. We will restrict our reduction relation as we did for the simple typed λ-calculus of chapter 3, to obtain a rewriting subcalculus for call-by-name, and present the machine suggested by this relation. For completeness we will state the operational semantics of the language.

Much of §5.1 and §5.2 is an excerpt from Barber's original paper.

5.1 The syntax of DILL

5.1.1 Definitions (formulas, contexts, and rules) A logical sequent of DILL,

$$\Gamma\,;\Delta \vdash A,$$

contains the *dual context* $\Gamma\,;\Delta$, where Γ is called the *intuitionistic subcontext*, which contains the intuitionistic assumptions, and Δ is called the *linear subcontext*, which contains the linear assumptions. We will refer to subcontexts also as contexts for brevity. We will understand Γ as a set and Δ as a multiset of formulas.

The syntax of the formulas, ranged over by the letters A,B,C is inductively generated by

$$A, B ::= P \mid A \multimap B \mid\, !A,$$

where P ranges over a set of propositional constants. Linear implication $\multimap$ is right-associative and binds weaker than the exponential !. For example, $A \multimap\, !B \multimap A$ is read as $A \multimap ((!B) \multimap A)$.

The natural deduction rules are given in Figure 5.1[4]. □

Note that linear contexts are non-shared, and that there are no rules for Contraction and Weakening. In this way, assumptions have to be used exactly once (refer to §2.5). On the other hand, subderivations share the intuitionistic contexts, allowing contraction and weakening to occur implicitly. The linear context in I-Axiom is empty, indicated with the bar '$-$', since it does not consume any linear resources.

Intuitively, an intuitionistic formula A can be regarded as equivalent to a linear formula $!A$, where we refer to A as the *base* of the exponential; the following two-way rule is a derived rule in DILL:

$$\frac{\Gamma, A\,;\Delta \vdash B}{\Gamma\,;\Delta, !A \vdash B}$$

The exponential mediates between both contexts. The 'Promotion' rule, !-I, states that if a formula A is proved only from intuitionistic assumptions, then $!A$ is proved, since the sequent may be used any number of times. The 'Dereliction' rule, !-E, establishes that a linear formula $!A$ may be used non-linearly to prove B, i.e., as an intuitionistic assumption A.

[4]The reader should note that we just included the implicational fragment of the logic as is usual in the rest of this thesis. Extensions will be briefly mentioned at the end of the chapter.

$$\frac{}{\Gamma\,;A\vdash A}\,\text{L-Axiom} \qquad \frac{}{\Gamma,A\,;-\vdash A}\,\text{I-Axiom}$$

$$\frac{\Gamma\,;\Delta,A\vdash B}{\Gamma\,;\Delta\vdash A\multimap B}\multimap\text{-I} \qquad \frac{\Gamma\,;\Delta_1\vdash A\multimap B \quad \Gamma;\Delta_2\vdash A}{\Gamma\,;\Delta_1,\Delta_2\vdash B}\multimap\text{-E}$$

$$\frac{\Gamma\,;-\vdash A}{\Gamma\,;-\vdash\,!A}\,!\text{-I} \qquad \frac{\Gamma,A\,;\Delta_1\vdash B \quad \Gamma\,;\Delta_2\vdash\,!A}{\Gamma\,;\Delta_1,\Delta_2\vdash B}\,!\text{-E}$$

Figure 5.1: The natural deduction rules of DILL

5.2 The linear term calculus

We now present the linear λ-calculus obtained by application of the Curry-Howard correspondence. In the same way that the λ-calculus is the model for (intuitionistic) functional programming languages, DILL may be regarded as a model for *linear* functional languages. In summary, DILL is a conservative extension of the λ-calculus with special *control* constructs (Promotion and Dereliction) that indicate the non-linear usage of variables.

This minimal linear language was implemented as the input language of the xLIN abstract machine, presented in the next chapter. With this in mind, we will be quite informal and refer to programming intuitions when this would help to understand DILL better.

5.2.1 Definition (basic notions) The preterms are constructed by the following inductive definition:

$$M,N ::= x \mid \lambda x{:}A.M \mid\, !M \mid \mathsf{let}\ !x = M\ \mathsf{in}\ N$$

The letters x,y,z,u,v,w,f,g and h range over an infinite set of variables and the letters M,N,P are used to denote preterms. The Promotion operator, i.e., the exponential, binds the strongest, and Dereliction, the weakest. For example, $\mathsf{let}\ !x = M\ \mathsf{in}\ N\,P$ reads as $\mathsf{let}\ !x = M\ \mathsf{in}\ (N\,P)$.

The set of types corresponds to DILL formulas; in this context, P should be interpreted as a set of given ground types.

The free and bound variable sets are defined as before plus the new

$$\begin{aligned}\mathrm{FV}(\mathsf{let}\ !x = M\ \mathsf{in}\ N) &= \mathrm{FV}(M)\cup(\mathrm{FV}(N)\setminus\{x\})\\ \mathrm{BV}(\mathsf{let}\ !x = M\ \mathsf{in}\ N) &= \mathrm{BV}(M)\cup\mathrm{BV}(N)\cup\{x\}\end{aligned}$$

where it is clear that dereliction binds the x the same way as a function abstraction.

The definition of substitution adds the two new equivalences

$$\begin{aligned}(\mathsf{let}\ !x = M_1\ \mathsf{in}\ M_2)[x := N] &\equiv \mathsf{let}\ !x = M_1\ \mathsf{in}\ M_2\\ (\mathsf{let}\ !y = M_1\ \mathsf{in}\ M_2)[x := N] &\equiv \mathsf{let}\ !z = M_1[x := N]\ \mathsf{in}\ M_2[x := N]\\ &\qquad\qquad \text{if } x\not\equiv y \text{ and } z\notin \mathrm{FV}(M_2)\cup\{x,y\}\end{aligned}$$

$$\frac{}{\Gamma\,;x:A\vartriangleright x:A}\text{ L-Axiom}\qquad\frac{}{\Gamma,x:A\,;-\vartriangleright x:A}\text{ I-Axiom}$$

$$\frac{\Gamma\,;\Delta,x:A\vartriangleright M:B}{\Gamma\,;\Delta\vartriangleright\lambda x{:}A.M:A\multimap B}\multimap\text{-I}\qquad\frac{\Gamma\,;\Delta_1\vartriangleright M:A\multimap B\quad\Gamma\,;\Delta_2\vartriangleright N:A}{\Gamma\,;\Delta_1,\Delta_2\vartriangleright M\,N:B}\multimap\text{-E}$$

$$\frac{\Gamma\,;-\vartriangleright M:A}{\Gamma\,;-\vartriangleright\,!M:\,!A}\text{ Promotion}\qquad\frac{\Gamma,x:A\,;\Delta_2\vartriangleright N:B\quad\Gamma\,;\Delta_1\vartriangleright\,!M:A}{\Gamma\,;\Delta_1,\Delta_2\vartriangleright\mathsf{let}\;!x=M\;\mathsf{in}\;N:B}\text{ Dereliction}$$

Figure 5.2: The type system of DILL

In the sequel we are going to adopt Barendregt's variable convention to avoid variable capture and renaming.

A preterm M is a λ_{DILL}-term if there exists a derivation with M as conclusion, as before. The type system is given in Figure 5.2.

A dual context $\Gamma\,;\Delta$ is well-formed if Γ,Δ is well-formed. If we write $\Gamma,x:A\,;\Delta$ or $\Gamma\,;\Delta,x:A$, then x should not occur in Γ or Δ. □

5.2.2 Remark (linear types) When DILL formulas are interpreted as types, $A\multimap B$ is the type of linear functions, i.e., functions that use their arguments only once, and $!A$ is the type of objects that may be used any number of times. Thus, intuitionistic functions have type $!A\multimap B$.

Also, the function $!(A\multimap B)$ is a linear function that may be used any number of times. Note that $!A\multimap B$ is itself and intuitionistic function, but its result is linear, so it can only be used once; this is contrast with $!(!A\multimap B)$ where the intuitionistic function may be used again freely, and so on.

In linear logic, $!A$ is *provably* equivalent to $!!A$; in fact, we have that $!A\multimap !!A$ and viceversa. This means that there is no need of ever writing two !'s together (or more). In practice, we will rarely find ourselves in the urge of typing or promoting something using more than one ! in a row. In any case, $!A$ and $!!A$ are not *semantically* equivalent. □

Functional programming is about function abstraction and application (or some 'sugared' version of it). Linear functional programming is also about creating and composing functions, but with the extra requirements imposed by the *linearity constraints* which must be looked after. We not only must ensure that types 'match up', but also that we are not violating the linearity constraints.

To intuitively understand the meaning of the two new constructs, Promotion and Dereliction, we will mention a few 'programming tips'. We first observe that functions are linear by default, so in order to write an intuitionistic function we must resort to Dereliction. A classical example is the combinator **S**, which in DILL is written

$$\lambda f{:}A\multimap B\multimap C.\lambda g{:}A\multimap B.\lambda x{:}!A.\mathsf{let}\;!z=x\;\mathsf{in}\;f\,z\,(g\,z).$$

In this example, we 'derelicted' the bound variable x which appears non-linearly (twice) in the term. In the following derivation for the combinator **K**, we applied dereliction to y, which

is not used at all. (Note that the linear variable $y : !A$ becomes an intuitionistic variable $z : A$.)

$$\dfrac{\dfrac{\dfrac{z : B\,;\, x : A \triangleright x : A \qquad -\,;\, y : !B \triangleright y : !B}{-\,;\, x : A, y : !B \triangleright \mathsf{let}\ !z = y\ \mathsf{in}\ x : A \multimap !B \multimap A : A}\ \text{Dereliction}}{-\,;\, x : A \triangleright \lambda y{:}!B.\mathsf{let}\ !z = y\ \mathsf{in}\ x : A \multimap !B \multimap A : !B \multimap A}\ \text{Promotion}}{-\,;- \triangleright \lambda x{:}A.\lambda y{:}!B.\mathsf{let}\ !z = y\ \mathsf{in}\ x : A \multimap !B \multimap A}\ \text{Promotion}$$

The programming 'idiom' used to write an intuitionistic function $\lambda x{:}A.M$ of type $A \to B$ is clearly

$$\lambda x{:}!A.\mathsf{let}\ !y = x\ \mathsf{in}\ M[x := y] \quad \text{for } y \text{ fresh.}$$

We may want to derelict other terms besides variables. For example, a function, as in this (not very interesting) example: $\lambda x{:}A.\mathsf{let}\ !f = !(\lambda y{:}A.y)\ \mathsf{in}\ f\ (f\ x)$.

For **K** and **S** it is easy to spot whether dereliction should be used or not: the derelicted terms syntactically occur non-linearly. Unfortunately, linearity is not so obvious to detect, although one may develop the programming skills to do it quickly. An object that occurs once is not necessarily linear. Linearity is concerned not with syntactical aspects, like *occurrence*, but with semantical aspects, like the *usage* of resources.

Application is also linear by default. In the term $M\,N$, where M is a function of intuitionistic type $!A \multimap B$, N needs to be promoted. Hence, we will write $M\,!N$. This is our next programming idiom. Note that for $!N$ to make sense, N must not contain any linear variables. If there were any, these must be derelicted and the types promoted accordingly. For example, let us consider the following λ-term

$$\lambda x{:}B.\lambda y{:}A.\mathbf{K}\,y\,((\lambda u{:}B.u)\,x).$$

To translate this term into DILL we would need to promote the second argument of **K**; also x should be derelicted, as follows

$$\lambda x{:}!B.\mathsf{let}\ !z = x\ \mathsf{in}\ \lambda y{:}A.\mathbf{K}\,x\,!((\lambda u{:}B.u)\,z).$$

For the syntactic results of the typing system the reader is referred to [4], §§3.2. Besides weakening and contraction for the intuitionistic context, we will need the following result:

$$\Gamma\,;\,\Delta, x : A \triangleright M : B \quad \text{implies} \quad x \text{ occurs } \textit{only once} \text{ in } M \qquad (**)$$

Note that this property can be easily violated by consider a very simple extension like the conditional[5],

$$\dfrac{\Gamma\,;\,\Delta_1 \triangleright M : \mathsf{Bool} \quad \Gamma\,;\,\Delta_2 \triangleright N : A \quad \Gamma\,;\,\Delta_2 \triangleright P : A}{\Gamma\,;\,\Delta_1, \Delta_2 \triangleright \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P : A}$$

where a linear variable x can occur in both subderivations N and P, although, during reduction, only one of the branches will apply. Here it is clear how linearity depends on usage rather than occurrence. We will come back to this problem later when we consider reduction in the context of explicit substitutions in the following chapter.

[5] A more logic-oriented example is the sum.

5.3 Type-checking

The notation for the typing rules of the calculi we have seen so far have an obvious algorithmic reading, which immediately suggests a recursive type reconstruction (or type-checking) algorithm. This algorithm may be regarded as a function that takes a program and a context containing the declarations of the free variables and returns its type, if the program is well-typed, or an error. For example, considering first the $\to$-E rule of the simple typed λ-calculus,

$$\frac{\Gamma \triangleright M : A \to B \quad \Gamma \triangleright N : A}{\Gamma \triangleright M\,N : B} \to\text{-E}$$

the type of $M\,N$ will be B, if the result of (recursively) applying the type-checking function to M is $A \to B$, and the result of applying this function to N is the same type as A. In both cases, the context passed to the new calls of the type-checking function is the context of the current call, Γ above. Renaming of bound variables may occur as part of this process to avoid name clash, while ensuring local scope.

Now, let's consider the $\multimap$-E rule:

$$\frac{\Gamma\,;\Delta_1 \triangleright M : A \multimap B \quad \Gamma\,;\Delta_2 \triangleright N : A}{\Gamma\,;\Delta_1,\Delta_2 \triangleright M\,N : B} \multimap\text{-E}$$

As it stands, the rule suggests 'splitting up' the contexts into two subcontexts, each containing the linear variables of M and N, respectively. Assuming that the program is well-typed, a recursive syntactic analysis of occurrence of free variables would be enough. Although computers nowadays are fast enough to let us have more leeway for indulging ourselves with some inefficient solutions, we will give a more efficient and simpler one, which emphasizes the view of linear logic as a logic of resources.

An alternative type-checking algorithm that does not rely on calculations with variable sets was naturally noticed elsewhere; for example, Mackie [28] gave a type assignment algorithm for his linear language LILAC, which is based on Abramsky's linear term calculus (in Abramsky [3]). Although the problem he considered was much harder, and more realistic, than that we are considering here, i.e., assigning a principal type for a (type-free) linear program, we will repeat the idea here. We also borrowed his alternative notation for the typing rules.

This idea consists in passing the whole context to one of the derivations. We 'implement' the fact that linear variables are consumed during type-checking by *removing* linear variables that are accessed. If a linear variable appears more than once in a derivation, the first access will cause the variable to be removed from the linear context, while the second one will cause an error. If it is not consumed, it will appear as a 'left-over' of the whole process of type-checking and an error will be reported. This is the approach we followed in our implementation; see §§7.2.3.

We may wish to reflect the new approach explicitly by altering a bit the syntax of the typing rules. Making the syntax closer to the implementation is useful in order to simplify the proof of correctness of the type-checking algorithm. For a linear context Δ we will write $\Delta \,|\, \Delta'$, meaning that "Δ' is the result of consuming the variables in Δ" in some derivation. Modifying the rules is straightforward, as Figure 5.3 shows. The set before the '$|$' is called, naturally, *before* set and the one after, *after* set. Note that $\Delta, x : A \,|\, \Delta$ easily captures the idea of 'consuming' a linear variable; $\Delta \,|\, \Delta$ means that no linear variables should be consumed (as in the Promotion rule).

$$\frac{}{\Gamma\,;\Delta, x:A \,|\, \Delta \triangleright x : A}\ \text{L-Axiom} \qquad \frac{}{\Gamma, x:A\,;\Delta \,|\, \Delta \triangleright x : A}\ \text{I-Axiom}$$

$$\frac{\Gamma\,;\Delta, x:A \,|\, \Delta' \triangleright M : B}{\Gamma\,;\Delta \,|\, \Delta' \triangleright \lambda x{:}A.M : A \multimap B}\ \multimap\text{-I} \qquad \frac{\Gamma\,;\Delta \,|\, \Delta' \triangleright M : A \multimap B \quad \Gamma\,;\Delta' \,|\, \Delta'' \triangleright N : A}{\Gamma\,;\Delta \,|\, \Delta'' \triangleright M\,N : B}\ \multimap\text{-E}$$

$$\frac{\Gamma\,;\Delta \,|\, \Delta \triangleright M : A}{\Gamma\,;\Delta \,|\, \Delta \triangleright\, !M : !A}\ \text{Promotion} \qquad \frac{\Gamma, x:A\,;\Delta \,|\, \Delta' \triangleright N : B \quad \Gamma\,;\Delta' \,|\, \Delta'' \triangleright M : !A}{\Gamma\,;\Delta \,|\, \Delta'' \triangleright \mathsf{let}\ !x = M\ \mathsf{in}\ N : B}\ \text{Dereliction}$$

Figure 5.3: The type system of DILL using before/after sets

It can be easily checked that, indeed, if $\Gamma\,;\Delta \,|\, \Delta' \triangleright M : A$, then $\Gamma\,;\Delta\backslash\Delta'$, i.e., the difference of the before and after sets is exactly the set of linear variables needed to type-check M. We say that a term M type-checks in the modified typing system whenever $\Gamma\,;\Delta \,|\, - \triangleright M$ (for some Γ and Δ).

5.4 Reduction, call-by-name, and linear substitution

We now turn to reduction in DILL. Figure 5.4 shows the two normalization cases. Besides linear function abstraction, we have to consider the detour for the exponential. The β-reduction relation will be the contextual closure of the union of the two subrelations

$$(\lambda x{:}A.M)\,N \to M[x := N] \qquad (\beta{\multimap})$$

and

$$\mathsf{let}\ !x = !M\ \mathsf{in}\ N \to N[x := M] \qquad (\beta!)$$

The correctness of these transformations is evident from the structure of the proofs. In the corresponding proof for $(\beta!)$, M must not contain any linear variables since the sequent may be used any number of times as a consequence of substitution in N. On the other hand, no constraints are imposed on the contexts for $(\beta{\multimap})$ since x occurs linearly in the derivation of N, and thus can be freely substituted by a term containing other linear variables.

We observe here that by the above transformations we can distinguish two notions of substitution, a *linear substitution* occurring when we contract a $\beta{\multimap}$-redex, and an *intuitionistic*, or *non-linear*, *substitution*, when we contract a $\beta!$-redex. By property $(**)$ a linear variable will syntactically occur exactly once, thus linear substitution can be implemented as efficiently as an in-place update (see Remark 5.4.2 below).

However, this distinction does not have only pragmatic grounds. Since DILL has two-sided sequents, we will have two Cut rules, one in which the cut-formula occurs in the intuitionistic context and another one where it occurs in the linear context:

$$\frac{\Gamma, A\,;\Delta \vdash B \qquad \Gamma\,;- \vdash A}{\Gamma\,;\Delta \vdash B}\ \text{I-Cut} \qquad \frac{\Gamma\,;\Delta_1, A \vdash B \qquad \Gamma\,;\Delta_2 \vdash A}{\Gamma\,;\Delta_1, \Delta_2 \vdash B}\ \text{L-Cut}$$

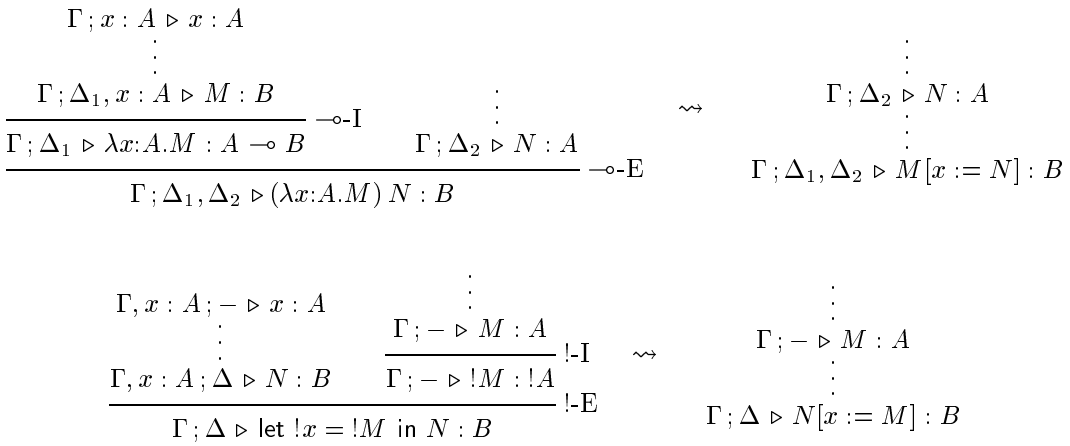

Figure 5.4: Proof normalization in DILL

These are derived rules in DILL, hence we read the annotated versions as substitution lemmas in the calculus.

5.4.1 Lemmas (substitutions) *DILL satisfy the following intuitionistic and linear substitution properties:*

$$\Gamma, x : A\,;\Delta \triangleright M : B \text{ and } \Gamma\,;- \triangleright N : A \text{ implies } \Gamma\,;\Delta \triangleright M[x := N] : B$$
$$\Gamma\,;\Delta_1, x : A \triangleright M : B \text{ and } \Gamma\,;\Delta_2 \triangleright N : A \text{ implies } \Gamma\,;\Delta_1, \Delta_2 \triangleright M[x := N] : B$$

Proof. Induction on M; see [4]. □

This explicit separation is important from a practical viewpoint. We can study these two notions of substitution, which have different properties, separately, and derive linear abstract machines that incorporate both versions in a uniform fashion. It will be clear what we mean when we discuss explicit substitutions in DILL, which is our next step towards the derivation of the xLIN machine.

5.4.2 Remark (pragmatics of linear substitution) If we know that a variable occurs *just once* in a term, we might compile substitution for that variable in such a way that the propagation over the term structure in not needed. However, although the observation is valid, addressing this optimization in environment machines, where environments appear always at the outer level, is not so trivial, as we will see in the next chapter. For the moment, let us focus on the way in which linear substitution might be implemented.

A solution consists in simply updating in-place the location of the variable by the substituted term. This is perfectly safe since linear variables are used only once, the very first time they are accesed. The xLIN machine compiles function abstractions in such a way that the location of the cell that holds the linear variable in the code graph, i.e., the internal representation of the term that the xLIN machine understands (see §7.3), is included. Moreover, since linear variables do not access any environments, the translator does not include any information in their cells; in the implementation, we call these 'empty' cells *holes*. □

5.4.3 Proposition (properties of β-reduction) *In DILL, β-reduction, i.e., the union of β—$\circ$- and $\beta!$-reduction relations, satisfy subject reduction, confluence, and strong normalization.*

Proof. See Ritter et al. [16] □

It is now appropriate to suggest a slight variation in the syntax of variables that becomes necessary to avoid confusion in the context of reduction and explicit substitutions in the following chapter. We will use the letters a,b, and c to denote linear variables to explicitly distinguish them from intuitionistic variables. This convention saves us from having to introduce a separate syntax for linear and intuitionistic substitutions.

Now, we extend the call-by-name strategy of Definition 3.2.4 to DILL. Note that it makes sense to discuss $\beta!$-reduction as an 'extension' since it is independent from β—$\circ$-reduction.

We have to consider the new contextual rules

$$\frac{M \to M'}{!M \to !M'} \qquad (\xi!)$$

$$\frac{M \to M'}{\mathsf{let}\ !x = M\ \mathsf{in}\ N \to \mathsf{let}\ !x = M'\ \mathsf{in}\ N} \qquad (\mu!)$$

$$\frac{N \to N'}{\mathsf{let}\ !x = M\ \mathsf{in}\ N \to \mathsf{let}\ !x = M\ \mathsf{in}\ N'} \qquad (\nu!)$$

We will follow the principles that we used for functional abstractions. Recall that we are considering only closed terms. By the same principle that function abstractions where reduced in the context of an application, promotion terms will only be reduced in the context of dereliction (see Remark 3.2.5). Hence we drop $(\xi!)$. The rules $(\mu!)$ and $(\nu!)$ introduce non-determinism. Observ that $(\mu!)$ corresponds to a normal order reduction strategy; the rule $(\nu!)$ first reduces N which is the argument of the substitution in a $\beta!$-contraction.

5.4.4 Definition (call-by-name) The call-by-name strategy in DILL is characterized by extending the call-by-name rules of Definition 3.2.4 with the rules $(\beta!)$ and $(\mu!)$. □

Following the same delevolpment of chapter 3, we show in Figure 5.5 the resulting call-by-name reduction rules, the suggested operational semantics in Plotkin style, and the derived naïve abstract machine that implements the strategy.

Note that in this version with names, we need to push the pair (x, N) in the stack since the variable x is needed later for the substitution; this will not be necessary in a version with De Bruijn indices. Besides the final configuration $(\lambda a.M, \epsilon)$ we will also have $(!M, \epsilon)$, which is a value.

A promoted term, $!M$, is often called a *!-closure*, since it can be thought of as encapsulating a computation, that is only revealed in the contraction of a $\beta!$-redex.

5.4.5 Remark (intuitionistic function space) In practice it would of help to add an abbreviation for intuitionistic functions. The programmer of linear programs might get tired

Transition rules

$$\begin{aligned}(M\ N, S) &\to (M, N :: S)\\ (\mathsf{let}\ !x = M\ \mathsf{in}\ N, S) &\to (M, (x, N) :: S)\\ (\lambda a.M, S) &\to (M[a := N], S)\\ (!M, (x, N) :: S) &\to (N[x := M], S)\end{aligned}$$

Operational semantics

$$\frac{}{\mathsf{let}\ !x = !M\ \mathsf{in}\ N \to N[x := M]}$$

$$\frac{}{\lambda x.M \Downarrow \lambda x.M} \qquad \frac{}{!M \Downarrow !M}$$

$$\frac{M \Downarrow \lambda x.M' \quad M'[x := N] \Downarrow V}{M\ N \Downarrow V} \qquad \frac{M \Downarrow !M' \quad N[x := M'] \Downarrow V}{\mathsf{let}\ !x = M\ \mathsf{in}\ N \Downarrow V}$$

Figure 5.5: Call-by-name evaluation for DILL

typing and renaming variables, as required by the syntax of Dereliction. This could be done by extending the typing system with the following derived rules:

$$\frac{\Gamma, x : A\,;\Delta \triangleright M : B}{\lambda x{:}!A.M : A \to B}\ {\to}\text{-I} \qquad \frac{\Gamma\,;\Delta \triangleright M : A \to B \qquad \Gamma\,;- \triangleright N : A}{\Gamma\,;\Delta \triangleright M\ N : B}\ {\to}\text{-E}$$

The second rule (intuitionistic application) saves the programmer of having to promote the term himself, but he still must derelict any linear variables occurring in the operand of the application; this can be simply corrected by promoting the type of the binding function abstraction[6]. □

5.5 Considering some extensions

Real functional programming languages are based in an extended λ-calculus which normally includes basic types like booleans, integers, reals, etc., together with their primitive operations, and structured types, like pairs and sums. A general recursion construct is implicit in the possibility offered by these languages of allowing (mutually) recursive definitions.

We will not seriously address issues like recursion, but we will indicate instead how some of these extensions may be typed in DILL, as well as the suggested reduction rules. The xLIN machine can be extended to include the extensions, but we will not do this here.

[6] Dereliction will still be needed in practice, since an expression may contain variables that are bound in a top-level environment.

The language considered adds the following new types and constructs to DILL:

$$\begin{aligned} A, B &::= \mathsf{Unit} \mid \mathsf{Bool} \mid A \otimes B \mid A \oplus B \\ M, N, P &::= c^k \mid * \mid \mathsf{let}\ * = M\ \mathsf{in}\ N \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P \mid \langle M, N\rangle \\ &\quad \mid \mathsf{let}\ \langle x, y\rangle = M\ \mathsf{in}\ N \mid \mathsf{inl}\, M \mid \mathsf{inr}\, M \mid \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\, M \to y\,;\mathsf{inr}\, N \to P \\ &\quad \mid \mathsf{fix}\ x{:}A.M \end{aligned}$$

We consider Unit, the 'unit' type with a unique value, $*$, and Bool, the type of boolean values, $\{\mathsf{true}, \mathsf{false}\}$, as ground types. We also have a set of constants c^k of arity k that we use to model constants of some basic type (with arity 0) and primitive functions. The types $A \otimes B$ and $A \oplus B$ correspond to the linear pairs and sums. Intuitionistic pairs may be added with type $!A \otimes !B$, in the sense of Remark 5.4.5. Linear pairs are constructed, as usual, with $\langle M, N\rangle$, and destructed using the pattern-matching form $\mathsf{let}\ \langle a, b\rangle = M\ \mathsf{in}\ N$, instead of the usual projection functions; the case constructor corresponds to additive disjunction in the logic, so it behaves as in the λ-calculus. Finally, the construct 'ununit', $\mathsf{let}\ * = M\ \mathsf{in}\ N$, destructs a value of Unit type, and fix corresponds to a fixed-point combinator construct.

Figure 5.6 gives the typing rules for the extended calculus. Note that all the constants of basic type are typed in an empty linear context, since they do not consume any linear resources. Recursion is inherently non-linear, hence the typing rules ensures that the body of the fixed-point does not contain any linear variables.

The notions of reduction arising are shown below. The first rule corresponds to a family of δ-rules for applying primitive functions. The notation on the right-hand side $c^k(M_1, \ldots, M_k)$ denotes primitive application. The other rules are β-rules, except for the last one, which corresponds to the obvious rewriting rule for fixed-points. The resulting reduction system, because of this rule, is no more strongly normalizing.

$$\begin{aligned} c^k M_1 \ldots M_k &\to c^k(M_1, \ldots, M_k) \\ \mathsf{let}\ * = *\ \mathsf{in}\ M &\to M \\ \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ M\ \mathsf{else}\ N &\to M \\ \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ M\ \mathsf{else}\ N &\to N \\ \mathsf{let}\ \langle a, b\rangle = \langle M, N\rangle\ \mathsf{in}\ P &\to P[a := M][b := N] \\ \mathsf{case}\ \mathsf{inl} M\ \mathsf{of}\ \mathsf{inl}\, x \to N\,;\mathsf{inr}\, y \to P &\to N[x := M] \\ \mathsf{case}\ \mathsf{inr} M\ \mathsf{of}\ \mathsf{inl}\, x \to N\,;\mathsf{inr}\, y \to P &\to P[y := M] \\ \mathsf{fix}\ x{:}A.M &\to M\ (\mathsf{fix}\ x{:}A.M) \end{aligned}$$

A call-by-name set of reduction rules may be obtained by extending the one of Definition 5.4.4 with the above rules, plus contextual rules for reducing the leftmost subterm of the destructors for objects of unit and boolean types, pairs and sums. The set of values is then characterized by adding the constructors (see Remark 3.2.5). Note that primitive functions that are 'partially applied' ($\ell < k$) are also in WHNF:

$$V ::= c^k M_1 \ldots M_\ell \mid * \mid \mathsf{true} \mid \mathsf{false} \mid \langle M, N\rangle \mid \mathsf{inl}\, M \mid \mathsf{inr}\, M$$

$$\frac{}{\Gamma\,;-\,\triangleright\, c : A} \qquad \frac{}{\Gamma\,;-\,\triangleright\, * : \mathsf{Unit}} \qquad \frac{}{\Gamma\,;-\,\triangleright\, \mathsf{true} : \mathsf{Bool}} \qquad \frac{}{\Gamma\,;-\,\triangleright\, \mathsf{false} : \mathsf{Bool}}$$

$$\frac{\Gamma\,;\Delta_1 \triangleright M : \mathsf{Unit} \quad \Gamma\,;\Delta_2 \triangleright N : A}{\Gamma\,;\Delta_1,\Delta_2 \triangleright \mathsf{let}\ * = M\ \mathsf{in}\ N : A} \qquad \frac{\Gamma\,;\Delta_1 \triangleright M : \mathsf{Bool} \quad \Gamma\,;\Delta_2 \triangleright N : A \quad \Gamma\,;\Delta_2 \triangleright P : A}{\Gamma\,;\Delta_1,\Delta_2 \triangleright \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P : A}$$

$$\frac{\Gamma\,;\Delta_1 \triangleright M : A \quad \Gamma\,;\Delta_2 \triangleright N : B}{\langle M, N\rangle : A \otimes B}\otimes\text{-I} \qquad \frac{\Gamma\,;\Delta_1 \triangleright M : A \otimes B \qquad \Gamma\,;\Delta_2, a : A, b : B \triangleright N : C}{\Gamma\,;\Delta_1,\Delta_2 \triangleright \mathsf{let}\ \langle x, y\rangle = M\ \mathsf{in}\ N : C}\otimes\text{-E}$$

$$\frac{\Gamma\,;\Delta \triangleright M : A}{\Gamma\,;\Delta \triangleright \mathsf{inr}\, M : A \oplus B}\oplus\text{-I}_1 \qquad \frac{\Gamma\,;\Delta \triangleright M : B}{\Gamma\,;\Delta \triangleright \mathsf{inl}\, M : A \oplus B}\oplus\text{-I}_2$$

$$\frac{\Gamma\,;\Delta_1 \triangleright M : A \oplus B \quad \Gamma\,;\Delta_2, a : A \triangleright N : C \quad \Gamma\,;\Delta_2, b : B \triangleright P : C}{\Gamma\,;\Delta_1,\Delta_2 \triangleright \mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\, x \to N\,;\mathsf{inr}\, y \to P : C}\oplus\text{-E}$$

$$\frac{\Gamma, x : A\,;- \triangleright M : A}{\mathsf{fix}\ x{:}A.M : A}$$

Figure 5.6: Typing the extensions

Chapter 6

Adding explicit substitutions to DILL

In this chapter we will add explicit substitutions to DILL with the aim of deriving a linear abstract machine in the spirit of Krivine's machine, in which substitutions are carried out using environments.

We will proceed as follows.

Firstly, we will look at the naïve approach, which will consist in extending the type system of DILL with the two cut rules, and the reduction calculus with the rewriting rules obtained by looking at the cases of cut-elimination. This will lead us to a linear version of λx. Since we will have two explicit substitution constructs, we will use the convention of the previous chapter and write intuitionistic and linear variables with distinct letters.

Recalling the fact that linear substitution can be implemented efficiently as an in-place update, it will not be necessary (or even sensible) to consider a refined subcalculus for propagating linear substitution, unless we consider commonplace extensions in functional programming like the conditional, or the sum. But in any case, the calculus can be extended so that explicit linear substitution is confined to the special cases only, and continues to be an implicit operation for the others.

Secondly, following the same development of chapter 4 we will consider parallel (intuitionistic) substitution and restrict our reduction rules in the sense of the weak conditional calculus $\lambda\sigma_{cw}$. The resulting 'closure calculus', xDILL$_w$, is suitable for reasoning about weak strategies for linear abstract machines.

Thirdly, and finally, we will consider a call-by-name strategy for xDILL$_w$ and present the xLIN machine. We will look at some obvious optimizations, and draw some conclusions.

6.1 The naïve substitutions

In this approach we simply extend DILL's calculus with two explicit substitution constructs, a linear substitution, which we will write $\langle a := N\rangle$, and an intuitionistic substitution, written $\langle x := N\rangle$. The two cut rules introduced in the previous chapter are the new typing rules, and the cases of cut-elimination will give the (minimal) reduction rules, in the sense of λx. We will not consider composition of substitutions; the reader might want to check that there are four cases to consider.

Let us begin with linear substitution, typed by the following (annotated) cut rule

$$\frac{\Gamma\,;\Delta_1, a : A \triangleright M : B \qquad \Gamma\,;\Delta_2 \triangleright N : A}{\Gamma\,;\Delta_1,\Delta_2 \triangleright M\langle a := N\rangle : B}\ \text{L-Sub}$$

and eliminated by the following rewriting rules

$$\begin{array}{rcll}
a\langle a := N\rangle & \to & N & \\
(\lambda b{:}A.M)\langle a := N\rangle & \to & \lambda b{:}A.M\langle a := N\rangle & \\
(M_1\, M_2)\langle a := N\rangle & \to & (M_1\langle a := N\rangle)\, M_2 & \text{if } a \in \mathrm{FV}(M_1) \\
(M_1\, M_2)\langle a := N\rangle & \to & M_1\, (M_2\langle a := N\rangle) & \text{if } a \in \mathrm{FV}(M_2) \\
(\mathsf{let}\ !x = M_1\ \mathsf{in}\ M_2)\langle a := N\rangle & \to & \mathsf{let}\ !x = M_1\langle a := N\rangle\ \mathsf{in}\ M_2 & \text{if } a \in \mathrm{FV}(M_1) \\
(\mathsf{let}\ !x = M_1\ \mathsf{in}\ M_2)\langle a := N\rangle & \to & \mathsf{let}\ !x = M_1\ \mathsf{in}\ M_2\langle a := N\rangle & \text{if } a \in \mathrm{FV}(M_2)
\end{array}$$

Note that the rules come in pairs for those term constructs with two subterms. This is because a linear variable will occur exactly once in only one of these subterms. Propagating substitution in both sides, as in the intuitionistic case, would break the linearity constraints, since N may contain free linear variables. Linear substitution, then, propagates N down the structure of the term by following the appropriate branch at each step. It is pointless to take an approach based on an implementation of linear substitution using the above rewriting rules, where a 'decision' need to be taken based on information about the location of the linear variable. We will prefer the more reasonable alternative of leaving linear substitution as a meta-operation, since we observed that it admits a more efficient implementation as an in-place update. Indeed, this was the case we wanted to identify and implement more efficiently on the first place! In other words, our old rule from DILL remains unchanged:

$$(\lambda a{:}A.M)\, N \to M[a := N]. \qquad (\beta{\multimap})$$

The extra bit of complexity involved in analyzing the structure of the term to locate the linear variable and compile the information in the binding function abstraction goes to the translation phase. (We will see in the next chapter that no 'forward' anaysis is require in practice; for the details check §7.2.4.)

However, as we observed before, if we consider conditionals or sums a linear variable may occur more than once in a term. For example, for the case of the conditional we will have the rule

$$(\mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3)\langle a := N\rangle \to \mathsf{if}\ M_1\ \mathsf{then}\ M_2\langle a := N\rangle\ \mathsf{else}\ M_3\langle a := N\rangle$$

if $a \in \mathrm{FV}(M_2\, M_3)$. For a reduction calculus that include extensions like these we will have to introduce again linear substitution explicitly and reduction rules like the above. For the other cases, it will remain an 'instantaneous' one-step operation. We will have more to say about these problematic cases in future work.

An explicit intuitionistic substitution is introduced as expected by the rule

$$\mathsf{let}\ !x = !M\ \mathsf{in}\ N \to N\langle x := M\rangle \qquad (\mathrm{b}!)$$

and propagated with the following σ-rules:

$$\begin{array}{rcl}
a\langle x := N\rangle & \to & a \\
x\langle x := N\rangle & \to & M \\
y\langle x := N\rangle & \to & y \quad \text{if } x \not\equiv y \\
(\lambda a{:}A.M)\langle x := N\rangle & \to & \lambda a{:}A.M\langle x := N\rangle \\
(M_1\, M_2)\langle x := N\rangle & \to & M_1\langle x := N\rangle\, M_2\langle x := N\rangle \\
(!M)\langle x := N\rangle & \to & !M\langle x := N\rangle \\
(\mathsf{let}\ !x = M_1\ \mathsf{in}\ M_2)\langle x := N\rangle & \to & \mathsf{let}\ !x = M_1\langle x := N\rangle\ \mathsf{in}\ M_2\langle x := N\rangle
\end{array}$$

The typing system is extended accordingly with DILL's (annotated) intuitionistic cut:

$$\frac{\Gamma, x : A\,;\Delta \triangleright M : B \qquad \Gamma\,;- \triangleright N : A}{\Gamma\,;\Delta \triangleright M\langle x := N\rangle}\ \text{I-Sub}$$

Reduction in our (typed) linear version of λx with explicit intuitionistic substitution is defined by taking the contextual closure of the notion of reduction obtained by taking the union of ($\beta\multimap$), (b!), and the σ-rules.

6.1.1 Remark (commuting conversions) The above reduction is incomplete if we do not consider the following *commuting conversion* rules

$$\begin{array}{c}
(\mathsf{let}\ !x = M\ \mathsf{in}\ N)\, P \to \mathsf{let}\ !x = M\ \mathsf{in}\ N\, P \\
\mathsf{let}\ !x = \mathsf{let}\ !y = M\ \mathsf{in}\ N\ \mathsf{in}\ P \to \mathsf{let}\ !y = M\ \mathsf{in}\ \mathsf{let}\ !x = N\ \mathsf{in}\ P
\end{array}$$

To motivate why these rules are necessary, consider the term $\mathsf{let}\ !x = y\ \mathsf{in}\ \lambda u{:}B.u$, where $y : !A$. Note that although it has type $B \multimap B$, it cannot be applied to a term, say, $M : B$, because $(\mathsf{let}\ !x = y\ \mathsf{in}\ \lambda u{:}B.u)\, M$ is simply not a redex! However, note that the equivalent term $\mathsf{let}\ !x = y\ \mathsf{in}\ (\lambda u{:}B.u)\, M$ can now be reduced. This is an undesirable characteristic of Dereliction. Fortunately, we will not have to deal with commuting conversions if we consider only closed terms. □

6.1.2 Proposition (properties) *The typed linear λx calculus with explicit intuitionistic substitution is confluent and strongly normalizing.*

Proof. For $\beta!$ the argument can be adapted from previous work on λx. Note that $\beta\multimap$ does not introduce any explicit substitutions, and that it is basically independent from $\beta!$. The reader may consult [16] for a stronger result. □

6.2 A weak xDILL calculus

A weak DILL with explicit substitutions, in the spirit of $\lambda\sigma_{cw}$ (or with minor differences of presentation $\lambda\rho$), can be easily obtained by considering environments, i.e., parallel substitutions, and reduction only in the context of closures, again, terms like $M\langle\sigma\rangle$, for σ an environment (see §4.2). Note that the environments we will consider are a generalization of $\langle x := M\rangle$, hence they refer only to intuitionistic variables. This is in constract with xDILL, proposed by Ritter et al. [16], which may contain both linear and intuitionistic variables.

We do not pretend to propose xDILL$_w$ to reason about linear abstract machines in a general setting, only as a starting point. The calculus is still quite restricted. The natural step is to consider composition of substitutions in the sense of $\lambda\sigma_w$ as well as explicit linear substitutions, for some of the extensions.

As before we will write environments as lists of bindings $(x_1 := M_1) \cdot \cdots \cdot (x_n := M_n) \cdot \mathrm{id}$, which are well-formed if they can be typed using the following typing rules:

$$\frac{\Gamma\,;\Delta \triangleright M : A}{\Gamma\,;\Delta \triangleright M\langle \mathrm{id}\rangle : A}\,\mathrm{Id} \qquad \frac{\Gamma, x : A\,;\Delta : M\langle\sigma\rangle : B \qquad \Gamma\,;- \triangleright N : A}{\Gamma\,;\Delta \triangleright M\langle (x := N)\cdot\sigma\rangle : B}\,\mathrm{Cons}$$

Figure 6.1 shows xDILL$_w$, our version of a closure calculus for DILL. Note that we have, again, restricted ourselves to closed terms. If we were to include open terms we would be lead to add composition of substitutions. This is not the case for the intuitionistic fragment only, i.e., as is the case of $\lambda\sigma_{cw}$. To illustrate where composition of substitutions come into play, even in this simple setting, consider the following closed term

$$(\lambda a.a)\langle\tau\rangle\, M\langle\sigma\rangle.$$

Applying (b—∘), we get $M\langle\sigma\rangle\langle\tau\rangle$. The characteristic of the closure calculus is that environments and terms are separate; (b—∘) breaks this implicit rule. The in-place update 'injects' a closure inside the body of a function abstraction. This problem can be remedied easily by adding the (σsub) rule, $M\langle\sigma\rangle\langle\tau\rangle \to M\langle\sigma\rangle$, which simply discards the outer environment τ. The correctness of this reduction is given by the imposed condition that the term we started with, i.e., in practice usually the closure $M\langle\mathrm{id}\rangle$, was already closed; hence, no free variables could ever be bound by τ.

It is evident that if we wanted to support reduction for open terms, we would be obliged to include a composition operator. The usefulness of this calculus relies on the fact that, indeed, at least for the case of closed terms, composition is not needed in a linear framework. We state without proof that xDILL$_w$ is confluent and strongly normalizing. At least, we will show that the call-by-name strategy, from which the xLIN machine will be derived, reduces terms to WHNF.

6.3 The xLIN linear abstract machine

We are now in a position to consider a leftmost-outermost strategy for xDILL$_w$ and 'derive' a linear machine that implements the strategy. As we showed before for the KAM, we forbid reduction inside substitutions and leave only the contextual rules that evaluate in head-position; hence we drop the (ν?) and the (sub-?) rules. The resulting reduction system, if represented appropriately, yield the named version of xLIN, shown in Figure 6.2. Again, we will write $\xrightarrow{\mathrm{n}}$ for call-by-name reduction.

6.3.1 Lemma *If M is a closed irreducible term w.r.t. the call-by-name strategy, then M is in WHNF, namely, M is either $(\lambda a.M')\langle\sigma\rangle$ or $(!M')\langle\sigma\rangle$.*

Proof. First, note that we are assuming that we start with a closure $M\langle\sigma\rangle$, M a DILL term, then, during reduction, we can only encounter terms of the restricted syntax:

$$R ::= M\langle\sigma\rangle \mid R\,M\langle\sigma\rangle \mid \mathsf{let}\ !x = R\ \mathsf{in}\ M\langle\sigma\rangle.$$

Axiom rules

$$
\begin{array}{rcll}
(\lambda a.M)\langle\sigma\rangle\, N\langle\tau\rangle & \to & M[a := N\langle\tau\rangle]\langle\sigma\rangle & (\mathrm{b}{\multimap}) \\
\mathsf{let}\ !x = (!M)\langle\sigma\rangle\ \mathsf{in}\ N\langle\tau\rangle & \to & N\langle(x := M\langle\sigma\rangle)\cdot\tau\rangle & (\mathrm{b}!) \\
x\langle(x := M\langle\sigma\rangle)\cdot\tau\rangle & \to & M\langle\sigma\rangle & (\sigma\mathrm{varx}) \\
x\langle(y := M\langle\sigma\rangle)\cdot\tau\rangle & \to & x\langle\tau\rangle \quad \text{if } x \not\equiv y & (\sigma\mathrm{vary}) \\
(M\,N)\langle\sigma\rangle & \to & M\langle\sigma\rangle\, N\langle\sigma\rangle & (\sigma\mathrm{app}) \\
(\mathsf{let}\ !x = M\ \mathsf{in}\ N)\langle\sigma\rangle & \to & \mathsf{let}\ !x = M\langle\sigma\rangle\ \mathsf{in}\ N\langle\sigma\rangle & (\sigma\mathrm{der}) \\
M\langle\sigma\rangle\langle\tau\rangle & \to & M\langle\sigma\rangle & (\sigma\mathrm{sub})
\end{array}
$$

Contextual rules for terms

$$\frac{M \to M'}{M\,N \to M'\,N}\,(\mu) \qquad \frac{N \to N'}{M\,N \to M\,N'}\,(\nu)$$

$$\frac{M \to M'}{\mathsf{let}\ !x = M\ \mathsf{in}\ N \to \mathsf{let}\ !x = M'\ \mathsf{in}\ N}\,(\mu!) \qquad \frac{N \to N'}{\mathsf{let}\ !x = M\ \mathsf{in}\ N \to \mathsf{let}\ !x = M\ \mathsf{in}\ N'}\,(\nu!)$$

$$\frac{\sigma \to \sigma'}{M\langle\sigma\rangle \to M\langle\sigma'\rangle}\,(\nu\mathrm{sub})$$

Contextual rules for substitutions

$$\frac{M \to M'}{(x := M)\cdot\sigma \to (x := M')\cdot\sigma}\,(\text{sub-id}) \qquad \frac{\sigma \to \sigma'}{(x := M)\cdot\sigma \to (x := M)\cdot\sigma'}\,(\text{sub-cons})$$

Figure 6.1: The reduction rules for xDILL$_w$

$$
\begin{array}{rcl}
(\sigma, \mathsf{let}\ !x = M\ \mathsf{in}\ N, S) & \to & (\sigma, M, (x, N\langle\sigma\rangle) :: S) \\
(\sigma, !M, (x, N\langle\tau\rangle) :: S) & \to & ((x := M\langle\sigma\rangle)\cdot\tau, N, S) \\
(\sigma, M\,N, S) & \to & (\sigma, M, N\langle\sigma\rangle :: S) \\
(\sigma, \lambda a.M, N\langle\tau\rangle :: S) & \to & (\sigma, M[a := N\langle\tau\rangle], S) \\
(\sigma, M\langle\tau\rangle, S) & \to & (\tau, M, S) \\
((x := M\langle\sigma\rangle)\cdot\tau, x, S) & \to & (\sigma, M, S) \\
((y := M\langle\sigma\rangle)\cdot\tau, x, S) & \to & (\tau, x, S) \quad \text{if } x \not\equiv y
\end{array}
$$

Figure 6.2: The xLIN abstract machine with names

Now, we can prove the statement by induction on the size of M and by the case analysis on the restricted R terms:

- $M \equiv M'\langle\sigma\rangle$. Note that M' cannot be an application, since (σapp) would apply; it cannot be a dereliction, since (σder) would apply; and so on. The only possibilities are for M' to be an abstraction or a promotion, which are WHNF's.
- $M \equiv M'\,M''\langle\sigma\rangle$. By induction, M' must be an irreducible term. (Recall that call-by-name reduces in head position only.) But, since M is an application, it must be of type $A \multimap B$, so the only possibility is for M' to be an abstraction. But this is impossible since (b$\multimap$) would apply. Hence, if M is irreducible, it cannot be an application.
- $M \equiv \mathsf{let}\ !x = M'\ \mathsf{in}\ M''\langle\sigma\rangle$. Similarly to the reasoning above.

□

6.3.2 Lemma *The xLIN machine implements the call-by-name strategy; namely, $M \xrightarrow{n} N$, then $M^\sharp \xrightarrow{xLIN} N^\sharp$.*

Proof. As for the KAM, the important step is to define the tranlation into machine steps; the implication then follows easily. Besides the translation rules for the KAM

$$(M\langle\sigma\rangle)^\sharp = (\sigma, M, \epsilon)$$
$$(M\langle\sigma\rangle\, N_1\langle\tau_1\rangle \ldots N_m\langle\tau_m\rangle)^\sharp = (\sigma, M, N_1\langle\tau_1\rangle :: \cdots :: N_m\langle\tau_m\rangle :: \epsilon)$$

we will have the (new) rule

$$(\mathsf{let}\ !x_1 = (\mathsf{let}\ !x_2 = (\cdots (\mathsf{let}\ !x_m = M\langle\sigma\rangle\ \mathsf{in}\ N_m\langle\tau_m\rangle)\ldots)\ \mathsf{in}\ N_2)\ \mathsf{in}\ N_1)^\sharp$$
$$= (\sigma, M, (x_m, N_m) :: \cdots :: (x_1, N_1) :: \epsilon)$$

Note that xLIN will stop at the configurations $(\sigma, \lambda a.M, \epsilon)$ and $(\sigma, !M, \epsilon)$, which map (by the obvious inverse translation) to WHNF's. □

6.3.3 Remark (xLIN refines KAM) It is fairly easy to see how xLIN appears as a refinement of KAM if we look at the sequence of steps necessary to evaluate function application in KAM and intuitionistic function application in xLIN. If we start with the closure $((\lambda a.\mathsf{let}\ !x = a\ \mathsf{in}\ M)\,!N)\langle\sigma\rangle$ evaluation using the above transition rules would proceed as follows

$$\begin{aligned}
&(\sigma, (\lambda a.\mathsf{let}\ !x = a\ \mathsf{in}\ M)\,!N, S)\\
&\quad\to (\sigma, \lambda a.\mathsf{let}\ !x = a\ \mathsf{in}\ M, (!N)\langle\sigma\rangle :: S)\\
&\quad\to (\sigma, \mathsf{let}\ !x = (!N)\langle\sigma\rangle\ \mathsf{in}\ M, S)\\
&\quad\to (\sigma, (!N)\langle\sigma\rangle, (x, M\langle\sigma\rangle) :: S)\\
&\quad\to (\sigma, !N, (x, M\langle\sigma\rangle) :: S)\\
&\quad\to ((x := N\langle\sigma\rangle)\cdot\sigma, M, S)
\end{aligned}$$

Note that the final configuration in this sequence is precisely the result of the reduction sequence below, using the rules for the KAM of §4.3.

$$(\sigma, (\lambda x.M)\,N, S) \to (\sigma, \lambda x.M, N\langle\sigma\rangle :: S) \to ((x := N\langle\sigma\rangle)\cdot\sigma, M, S)$$

Now, it is clear in which sense our machine is a refinement of KAM; it also points out an optimization which would be necessary in practice to have KAM as a *subset* of xLIN, so that KAM is derived as a special case if we restrict ourselves only to intuitionistic terms[1]. Theoretically, we could add the rule

$$(\sigma, \lambda a.\mathsf{let}\ !x = a\ \mathsf{in}\ M, (!N)\langle\tau\rangle :: S) \to ((x := N\langle\tau\rangle) \cdot \sigma, M, S),$$

but this is not a good idea in practice. The best idea seems to follow the suggestion of Remark 5.4.5 and add intutionistic function abstraction and application; then, we simply add the usual rule of the KAM

$$(\sigma, \lambda x.M, N\langle\tau\rangle :: S) \to ((x := N\langle\tau\rangle) \cdot \sigma, M, S)$$

□

Figure 6.3 shows the recast reduction and transition rules of a De Bruijn version of xLIN. The syntax of the target language is inductively defined by

$$M, N ::= n \mid \lambda a.M \mid M\,N \mid !M \mid \mathsf{derelict}\ M\ \mathsf{in}\ N,$$

where n is, again, an index, and a stands for a 'hole' that will be updated by linear substitution. Note that Dereliction is written as '$\mathsf{derelict}\ M\ \mathsf{in}\ N$' since the bound variable that appears in the named version will occur as an index 0 in the De Bruijn's version.

For future reference, let us refer to this version of xLIN as the *pure* xLIN machine. The prototype interpreter described in the following chapter implements this machine, modified as shown in §7.3 to obtain a call-by-need version, suitable for lazy (linear) functional programming languages.

[1] For example, if we consider Girard's embedding of intuitionistic logic into intuitionistic linear logic, which maps all intuitionistic functions of type $A \to B$ to linear functions of type $!A \multimap B$.

Reduction rules

$$
\begin{array}{rcl}
(\lambda a.M)\langle\sigma\rangle\, N\langle\tau\rangle & \to & M[a := N\langle\tau\rangle]\langle\sigma\rangle \\
(\mathsf{derelict}\ (!M)\langle\sigma\rangle\ \mathsf{in}\ N\langle\tau\rangle & \to & N\langle M\langle\sigma\rangle\cdot\tau\rangle \\
0\langle M\langle\sigma\rangle\cdot\tau\rangle & \to & M\langle\sigma\rangle \\
(\mathrm{succ}\, n)\langle M\langle\sigma\rangle\cdot\tau\rangle & \to & n\langle\tau\rangle \\
(M\ N)\langle\sigma\rangle & \to & M\langle\sigma\rangle\, N\langle\sigma\rangle \\
(\mathsf{derelict}\ M\ \mathsf{in}\ N)\langle\sigma\rangle & \to & \mathsf{derelict}\ M\langle\sigma\rangle\ \mathsf{in}\ N\langle\sigma\rangle \\
M\langle\sigma\rangle\langle\tau\rangle & \to & M\langle\sigma\rangle
\end{array}
$$

$$\frac{M \to M'}{M\ N \to M'\ N}$$

$$\frac{M \to M'}{\mathsf{derelict}\ M\ \mathsf{in}\ N \to \mathsf{derelict}\ M'\ \mathsf{in}\ N}$$

Transition rules

$$
\begin{array}{rcl}
(\sigma, \mathsf{derelict}\ M\ \mathsf{in}\ N, S) & \to & (\sigma, M, N\langle\sigma\rangle :: S) \\
(\sigma, !M, N\langle\tau\rangle :: S) & \to & (M\langle\sigma\rangle\cdot\tau, N, S) \\
(\sigma, M\ N, S) & \to & (\sigma, M, N\langle\sigma\rangle :: S) \\
(\sigma, \lambda a.M, N\langle\tau\rangle :: S) & \to & (\sigma, M[a := N\langle\tau\rangle], S) \\
(\sigma, M\langle\tau\rangle, S) & \to & (\tau, M, S) \\
(M\langle\sigma\rangle\cdot\tau, 0, S) & \to & (\sigma, M, S) \\
(M\langle\sigma\rangle\cdot\tau, \mathrm{succ}\, n, S) & \to & (\tau, n, S)
\end{array}
$$

Figure 6.3: The De Bruijn's version of the xLIN machine

Chapter 7

The Interpreter

In this chapter we describe in some detail the interpreter implemented as an interface to the xLIN abstract machine. We will explain the different design decisions that lead to the current version of the prototype interpreter (version 0.9α). We try to discuss the different techniques used without getting into the realistic level of complexity characteristic of an object-oriented language like C++; for example, we do not hint at the organization of the program in terms of modules and classes. Our primary concern here is to give a flavour of the practical issues involved in a 'real' implementation. Anyway, the techniques used follow standard lines, and are simple; the data structures are more or less suggested by the syntax of the target language, i.e., xDILL_w.

The reader may wonder why we used a (rather complex) language like C++ [36] instead of using the style of programming we advocate. Well, our original idea consisted in having a basic framework, i.e., an interpreter, based on the λ-calculus in a language that would let us have more control on low-level aspects. These become crucial if one wishes to experiment with different efficiency techniques that rely on the underlying arquitecture of the target machine. On the other hand, C++ provides traditional high-level features as well as supporting programming in the object-oriented style, which can be thought of as programming with Abstract Data Types (ADT)[1].

We have to stress that the current implementation is still in its infancy. Actually we should refer to it as a 'pre-interpreter', since in an interpreter we should be able to write some 'programs'. This is not the case yet[2]; for example, there is no notion of *top-level environment* where function definitions may be stored for later reference in a program. Everything the user wants xLIN to reduce, needs to be typed in a single expression.

This chapter is organized in four sections that can be read separately: §7.1 gives an overview of the organization of the interpreter; §7.2 and §7.3 describe the "back stage" of the implementation; and §7.4 provides a brief user's guide.

7.1 General overview

The interpreter program may be viewed as composed of two modules:

The Front-end mediates between the user and the xLIN machine, translating input expressions, i.e., terms of an extended DILL, into the internal representation, which we

[1] This is an unfair approximation, unless one considers *inheritance*.

[2] but it will be soon.

termed *code graph*, that xLIN understands. It includes lexical and syntactic analysis, type-checking, and translation (compilation), which we will look in more detail in the following section.

The front-end calls xLIN with the code graph as input, which then outputs in a readable form along with the returned type by the type-checking phase. Any errors in any of the stages are reported back to the user.

Alternatively, the front-end may accept commands; for example, typing '`:load`' followed by the name of a file, tells the interpreter to process the contents of that file. (More of this in §7.4.)

The xLIN Abstract Machine reduces a compiled input expression, which is an internal representation of an (extended) xDILL$_w$ term, to WHNF.

The version implemented here corresponds to the 'pure' abstract machine of §6.3. In the future, other versions that will include several optimizations will follow. The details are given in §7.3.

Although preliminary, this version includes a trace feature, activated by typing '`:trace`' in input mode[3]. This command causes the abstract machine to execute the code in trace mode. In this mode, the abstract machine stops evaluation and displays the internal registers, i.e., the current configuration, and allows the user to resume evaluation in several ways.

Now, we will look at the composing submodules of the front-end, which we briefly introduce as follows:

The Lexer and Parser or the lexical and syntactic analyzer, respectively, construct an Abstract Syntax Tree (AST) from an input expression which is input to the program as a string of characters.

An AST represents a term as a tree, as shown in §§7.2.2. This AST is then input to the type-checking and translation phases. The latter will 'degenerate' this tree into a graph, i.e., the resulting code from the translation will contain in the function abstraction cells explicit references to their bound linear variables cells.

The Type-checker takes an AST as input and returns a type expression as result. This type is itself encoded as an AST. It uses the type reconstruction rules of §5.3, using two contexts to keep track of intuitionistic and linear variables. Since we do not impose any naming conventions for variables, we will address the issue in §§7.2.3 of ensuring local scope for the case of two contexts.

The Translator returns a graph code from an input AST. The graph code resembles an AST, except that (a) variable names are compiled into De Bruijn indices, and (b) function abstractions are compiled along with a pointer to its bound linear variable, which is encoded as a 'hole' (see Remark 5.4.2).

We will briefly explain how this translation was implemented in §§7.2.4.

[3]We have not said anything about 'modes'. In general, the interpreter is in *input mode*, indicated by the input prompt '`<`', expecting to read in an expression or a command. Another example mode is *trace mode* which is indicated by the trace prompt '`?`'.

7.2 The front-end

7.2.1 Lexical and syntactic analysis

It is traditional to use some automatic lexer and parser generator, like Lex and Yacc, which then generate code in the object language that can easily be linked with the rest of the application program.

The author has some experience in writing lexers and parsers, and the structure of DILL as a programming language is so simple that the author decided not to use the existing tools and write a recursive-descent parser by hand. The approach has some advantages, though.

First of all, the parser is nicely integrated as part of the object-oriented framework of the application. Secondly, it allows for a better error handling; it is quite tricky and error prone to extend a grammar with error production rules in order to achieve a more or less reasonable result.

7.2.2 Data Representation

Inductive structures like terms[4] and type expressions, as well as code for abstract machines, are usually represented in a computer as *trees*. For example, the output of the parser is an AST, a term tree which has as *nodes*, (tag, data)-pairs, where the *tag* is usually some code to be able to identify the 'type' of the node, i.e., variable, function abstraction, application, etc., and *data* contains tag-dependent information. For instance, a node of variable type will contain the name encoded (for example) as a sequence of ASCII characters; an application node will contain two pointers to its subterms (encoded as addresses in the computer store); and so on. In the literature, nodes of this sort are often called *boxed* values, as opposed to tagless nodes or *unboxed* values. The reader is referred to Peyton Jones [30] for a discussion on this topic. In our implementation, all the objects are boxed.

7.2.3 Type-checking

The type-checker takes the AST of an input expression and returns another AST that represents the type of the expression; if the type-checker fails to find a type, an error is returned.

We will not give the type-checking algorithm here, which is a straightfoward implementation of the type-checking rules of DILL, as presented and discussed in §5.3. Instead, we will address the point of how local scope is ensured in the case of two contexts.

In the typed λ-calculus a term like $\lambda x{:}A.\lambda x{:}B.x$ would not be accepted as well-formed: the second bound occurence of x would clash with the first occurence, when trying to type $x : A \triangleright \lambda x{:}B.x$ (see Remark 3.1.6). We might think, even, that it would not be a good idea to allow programmers to type terms like this one, since if we consider it in isolation, it looks 'ambiguous', in the sense that it is not clear whether x is bound by the first or the second abstraction[5]. In functional programming the meaning of the function $\lambda x{:}A.\lambda x{:}B.x$ is given traditionally by what is known as *local scope*: x is bound by the second λ, which 'hides' the first one. A variable is always bound by the innermost operator with a bound variable of the same name.

[4] We have silently shifted our terminology a little bit and used the word 'expression' instead of 'term' which is more common in functional programming.

[5] Note that by the way substitution is defined, for untyped terms, $(\lambda x.\lambda x.x)\, M \to \lambda x.x$.

This is easy to implement by regarding contexts as stacks. Function abstractions extend contexts by pushing their bound variables. In this way, the innermost variable bound would appear at the top of the stack. Typing a variable is reduced to scanning the stack from top to bottom, so that the first variable that is found is the one that will be taken into account. This method can also be used to translate terms to their De Bruijn counterparts; see Proposition 3.4.1.

For the case with two contexts, consider the term $\lambda x{:}A.\mathsf{let}\ !x = M\ \mathsf{in}\ x$ (for some $M : B$). We are lead to a point where we have to type the variable x: $x : B\,; x : A \triangleright x$. Which x was the innermost? A solution consists in associating with each entry in the contexts (which are stacks) an integer value that corresponds to the value of a global count of entries (initially set to 0); each time a variable is pushed in any of the contexts, it is assigned the value of the global count, which then is incremented. Typing a variable consists in scanning both contexts and taking the variable with the maximum count. Thus, the typing situation is easily solved, since we will have something like $x_1 : B\,; x_0 : A \triangleright x_1 : B$.

A context is itself implemented as a linked list; each node being a quadruple containing a variable name, a pointer to its type AST, a global count value, and a pointer to the next node in the list. There are four basic operations on contexts: find, extend, remove, and length. The first two suggest themselves; the other two are used to implement the linearity constraints. When a linear variable is successfully typed, it is ('destructively') removed so that later accesses will fail (as intended). Verifying that no linear variables where consumed in the type-checking of a subterm is easily done by using length to compare the size of the linear context before typing the subterm and after; this is safe since linear contexts can only shrink (by removing linear variables), but they never grow, i.e., context extension is not a destructive operation, it always returns a new context and the original context is left intact.

7.2.4 Translation

The translation (or compilation) phase takes an expression AST as input and returns a code graph as result. As a precondition, the term is assumed to be well-typed.

Although it can be implemented as a separate phase, it is best to merge it with type-checking, since translation is defined recursively and, again, variables need to be interpreted correctly according to local scope. The code graph is a 'degenerated' case of an AST, as explained shortly after. In this case the syntax represented is that of the target language, xDILL_w with De Bruijn indices. We will refer to the nodes of the code graph as *cells*, which are special nodes that are allocated in the *pool*, a special area in the memory of the computer administered by the xLIN machine, and the *garbage collector*.

7.2.1 Remark (garbage collection) Since the pool is a finite collection of cells, garbage collection is necessary in order to proceed evaluation, if allocation happens to fail. Typically, the pool is inspected in order to identify and reclaim cells that were used, but which are not required any longer. This is a time-consuming process, but is the price to pay for leaving all memory-management duties to the language.

We will not discuss garbage collection here, since we are interested in having a prototype that does not require it in a future release. In the meantime, we plan to use a simple algorithm; for example, mark-and-sweep. □

The compile algorithm also takes two contexts (used here in a quite liberal sense) as arguments. Intuitionistic variables are compiled to their De Bruijn indices, using the translation

of Proposition 3.4.1. No information is needed, besides the variable names and their global count values; the index assigned to each variable is implicit in the location that it occupies in the stack.

Recall that linear variables are compiled as holes, i.e., empty cells, which only stand as markers for linear substitution. All the information that linear substitution needs at run-time is the address (a pointer) to the hole, so that the hole can be *overwritten*[6] by the argument of the function, which is popped up from the machine stack (see next §). Each entry in the linear context contains, besides a variable name and its global count, a pointer to its binding abstraction cell. When a function abstraction is compiled, a cell is created with an empty hole-pointer field, and its address is saved in the linear context. When a linear variable is compiled, the hole-pointer field is updated with the address of the newly created hole.

7.3 Implementation of the xLIN machine

xLIN is a machine, so at any given time it is in some *state* (or configuration), and it has a *control*, the reduction algorithm, that implements the transition function. Naturally, since xLIN is a transition system, the transition from each state to the next is uniquely determined by the *current state*, which is characterized by the following three registers[7]:

- E, or *environment register*, is a pointer to an environment, represented as an AST. This AST may be regarded as a sequence, where each node can be either of type id (if it is an empty sequence), or cons, in which case the node contains two fields: a pointer to a closure, and a pointer to an environment. (Recall that environments in the nameless call-by-name subcalculus will always be of the form $M_1\langle\sigma_1\rangle \cdot \cdots \cdot M_n\langle\sigma_n\rangle \cdot \mathrm{id}$.)

- C, or *code register* (also referred to as *control register*), is a pointer to the root of the code graph. Note that the transition rules are defined by cases on the structure of the code register, so it is actually very easy to determine which rule to apply by inspecting the tag of the root.

- S, or *stack register*, is an array containing closures (and update markers). xLIN defines the internal primitive operations push, pop, and top (with their common meanings) that it uses for implementing the different transitions. Note that final states are characterized by an empty stack, i.e., the state schemas $(\sigma, \lambda a.M, \epsilon)$ and $(\sigma, !M, \epsilon)$, so an extra primitive, empty, is also needed.

The machine is 'loaded' for execution by the front-end, which actives its reduce algorithm, passing the compiled graph code and an environment with the bindings of the free variables. (This version does support top-level environments, so the environment passed is always empty.)

The control proceeds as follows:

[6] This operation is implemented efficiently as bit-pattern copy; for example, in our prototype a cell occupies 6 machine words (or 24 bytes), so the contents of the root cell of the argument are word-wise copied to the address saved in the function cell.

[7] xLIN also contains the registers trace and steps; the former is activated in trace mode, and the latter, which has significance only in this mode, is used by xLIN so that it will stop and 'dump' the values of the internal registers when this count reaches 0 (by default set to 1).

S1. [WHNF reached?] If S is empty and C points to a value (an integer or boolean, an abstraction, or a !-closure) return the contents of C; otherwise, go to the next step.

S2. [Update environment?] If, again, C points to a value and the top of S contains an update marker, pop out the update marker and use the pointer to the closure contained in it to update the closure in the environment with the current values of E and C.

S3. [Compute.] Inspect the tag of the root at C and, according to its value, modify the current state according to the xLIN rules. For example, if the tag is abs (abstraction), overwrite the cell at the address contained in its hole-address field, and set C to point to the value of its body field, i.e., the root of the abstraction body.

S4. [Repeat.] Increment the internal transition count and go back to S1.

There are two notions of sharing that we deal with in this implementation. To illustrate the first one, consider the transition rule for application

$$(\sigma, M\ N, S) \to (\sigma, M,\ N\langle\sigma\rangle :: S).$$

From an implementation viewpoint, there are two alternatives, which are not differentiated in the calculus (naturally). We could either duplicate the environment σ, which would be quite an expensive operation, or we could share it. In fact, this is the reasonable choice. This is very easily achieved since we only manipulate pointers to cells of objects, whatever they might be (e.g. environments, closures, holes, etc.) In S we just push, along with a pointer to the root of N, the contents of E, which is itself a pointer to the head of the environment list. The correctness of this relies on the fact that we are not *discarding* any objects; that is the task of the garbage collector, that takes sharing into consideration.

The other notion of sharing is related with call-by-need and the update markers already informally discussed. It is not difficult to extend the calculus to explicit sharing strategies.

7.4 Pragmatics

Here we take a user-oriented viewpoint[8] and explain the rules to input and evaluate DILL expressions with xLIN.

7.4.1 Getting started

We begin by assuming, once more, that the user knows how to start the interpreter application program; usually, typing something like '`xlin`' is enough. The interpreter will display a start-up banner with the version of the program and the *input prompt* '`<`' immediately after, which indicates that it is waiting for an input expression or a command. We will refer to this state as *input mode*. The natural way to proceed is to give xLIN what it expects, and type in an input expression; the simplest of the expressions is a value of a basic type, like

```
< true;
> true :: Bool
```

[8] Well, the interested reader who would like to *actually* try xLIN will have to wait until the first release is made available. The prototype at the time of writing is not yet suitable for most platforms.

As a convention we will underline the text that is intended to be typed by the user. The *output prompt* indicates that what follows is the result of processing the input typed immediately before, in this case the same value, along with its type; the '`::`' symbol is read "is of type". Note that the end of an expression should be explicitly marked with a semi-colon. This is because spaces are insignificant and may be included freely to improve readability (except, naturally, when used to separate tokens). ⟨Return⟩'s are spaces too, so `xlin` will wait for more input if an expression is not properly terminated with '`;`'. Recall that application is written in DILL by juxtaposing expressions.

As a convenience, `xlin` supports the types `Bool` of boolean values and `Int` of integer values, but no primitives that operate on them[9].

The next example, shows **K** in action, and the syntax we used:

```
< (\x:Int.\y:!Int.let !z = y in x) -1 !1;
> -1 :: Int
```

The input language resembles the constructs of DILL we already used, except for '`\`' (backslash) that denotes 'λ', and '`-o`' (a minus sign followed by the letter 'o') for '$\multimap$'. The user may also want to write, for example, '`Int->Int`', as an abbreviation for '`!Int-oInt`'. Appendix A provides the exact definition of the syntax rules in BNF notation.

The example above takes 8 xLIN transitions. Actually, it is possible to see each of the steps xLIN has to follow to evaluate expressions by setting the trace option, as follows:

```
< :trace;
Trace on.
```

Now, `xlin` will stop immediately after an expression is typed in and will display its internal registers. The user is allowed to trace the evaluation step by step, or to continue reduction normally. The commands available in this mode as well as in input mode are discussed in the next §§.

Note that xLIN reduces xDILL$_w$ expressions, so the code register will contain compiled code, which takes a little time to get used to. As a tip, it is easy to learn how '`xlin`' compiles expressions simply by typing the expression into a !-closure. When the result is a function or a !-closure, `xlin` responds with something like '`[function:`⟨code⟩`]`' or '`[!-closure:`⟨code⟩`]`', respectively, dumping the code between the brackets.

Another useful programming tip concerns the fact that `xlin` does oblige the programmer to respect any naming conventions for variables. Local variables are interpreted according to local scope. Hence, it is easier to write intuitionistic functions, for example.

7.4.2 Commands

In input mode, the user can choose to type in a *command*. To get a quick grasp of the available features of the version in hand, entering '`:h`' (or the more verbose '`:help`') will output a list of all the available commands and a brief one-sentence explanation of their usage and meaning. Table 7.1 is a summary of the commands available in version 0.9$\alpha$of xLIN. The initial of the command names may be used as an abbreviation for the full name.

Note that when the interpreter is reading from a file it is not considered to be in input mode, so actions like reading a file from another file (using the '`:load`' command) are therefore

[9]Except for addition '`+`', which was added to experiment with primitive functions.

Name	Action
`help`	Display a list of the available commands
`load` ⟨filename⟩	Reads input from ⟨filename⟩
`quit`	Quits the interpreter[10]
`trace`	Activates/Deactivates the trace mode option

Table 7.1: Summary of input commmands

Syntax	Action
`c`	Cotinues evaluation without stopping
`h`	Display a list of the available commands
`n` ⟨num⟩	Skip the next ⟨num⟩ − 1 transitions
⟨Return⟩	Go to the next state (equivalent to '`n 1`', "next one")

Table 7.2: Trace commmands

not allowed. The contents of a file are expected to be a sequence of ⟨expression⟩'s (with the syntax of Appendix A). The result is as if they were typed one after the other in (normal) input mode.

7.4.3 Tracing expression reduction

Since xLIN is an experimental abstract machine, it was necessary to have some way to be able to *trace* down the evaluation of an expression to debug the machine and to test its behaviour in numerous ways. For this reason, we implemented a very simple trace feature, activated with the '`:trace`' command in input mode. This command instructs the interpreter to run the xLIN machine in trace mode.

In this mode, xLIN will stop after displaying the step (or transition) count, as well as the contents of the internal registers (E, C, and S, as described in §7.3), and will wait until the user types in a trace command after the *trace prompt* '`?`'. The user can choose to continue evaluation any number of steps, proceed normally, or stop altogether. The full list of commands is given in Table 7.2.

Note that the commands only allow control over the steps of reduction in a quite restrictive way. No stop condition can be expressed in terms of the register contents (e.g. "stop when the control register contains a value").

Chapter 8

Conclusions

8.1 Summary

The goal of this thesis was to present xLIN, a simple linear abstract machine, which might be charaterized as a refined version of Krivine's machine. As Krivine's machine, which reduces terms of the λ-calculus to weak-head normal form using a call-by-name strategy, xLIN reduces terms of a linear λ-calculus, which, following the spirit of Linear Logic, is a 'resource conscious' version of the λ-calculus. From a logical viewpoint, the analogy can be stated as follows:

$$\frac{\text{Dual Intutionistic Linear Logic}}{\text{xLIN Machine}} = \frac{\text{Intuitionistic Logic}}{\text{Krivine's Machine}}$$

In the equation, Dual Intuitionistic Linear Logic (DILL), corresponds to a presentation of Linear Logic that distinguishes between two classes of assumptions: intuitionistic and linear. This logic gives rise to a simpler calculus that does not possess explicit constructs for duplicating and discarding values. If considered as a programming language, DILL has the advantage over its predecessors of being at a higher level of abstraction.

We show how xLIN can be easily derived from a call-by-name subcalculus of xDILL_w, our version of DILL with (intuitionistic) explicit substitutions, which may be viewed as a linear version of a 'calculus of closures' in the spirit of $\lambda\rho$. The proof of correctness of the machine is a straightfoward consequence of the closeness of the calculus to the abstract machine. We informally show how to obtain a call-by-need version.

We tried to motivate the methodology applied from first principles, with the objective of being didactic. Particularly, we emphasize the role of logic through the Curry-Howard isomorphism as a methodological framework, which suggested several directions in the stages of design of the calculi. For example, the separation of substitutions in linear and intuitionistic in xDILL_w is a design decision motivated from the existence of two Cut rules in DILL.

We present the pragmatics of xLIN, by introducing the simple interpreter implemented as the interface to the machine. Without getting into unnecessary details, we briefly explain the techniques used, which are standard. Finally, a brief guide with examples shows the interpreter in action.

8.2 Further work

The present work spans many areas, giving ample space for future research. Many questions come to our mind, but there is still one obvious pragmatic question which remains open: "In every day functional programming, *how much do we gain from linearity*?" Roughly speaking, functional programming is about recursion, and recursion, as we know it, is inherently non-linear, intuitionistic. Supposing that we are very good programmers of linear programs, or that we count with an optimal embedding of intuitionistic programs into intuitionistic linear programs, how much more efficient is xLIN (for example) as an extension of Krivine's machine? In other words, how much was it worth the extension, with the trade-off of the increasing of complexity? Naturally, the experimental results will be linked to a particular style of programming, like the lazy style of programming. Variation in the programming style might increase the rate of linearity in a program, especially if connected with recursion. Without considering linear programming as such, which is exactly what the author does not want to do, in which way does the linearity constraints suggest a shift in the style of programming?

The answer to the first question would require testing realistic functional programming projects against Krivine's machine and the xLIN machine, which corresponds to its natural linear extension. The testing programs would need to be either rewritten very carefully into a linear functional programming language, a suitable extension of DILL, or be translated using an optimal embedding. We would like to address the second of the approaches, known since Schellinx [15] as the problem of *linear decoration*. Informally, with 'optimal' we mean that the translated linear program (or proof, if we address it in logical terms) be annotated with exponentials "in the right places". If we consider this translation on types rather than on terms, this problem can be restated as assigning a *maximally* linear type, thus suggesting an obvious ordering of linear types from less intuitionistic to less linear (for example, $A \multimap B \prec !A \multimap B \prec !!A \multimap B \prec \ldots$). The following translation is based on *Girard's translation*, an example of an obviously correct but extreme embedding:

$$\begin{array}{rcl} [\![x]\!] & = & x \\ [\![\lambda x{:}A.M]\!] & = & \lambda x{:}!A.\mathsf{let}\ !y = x\ \mathsf{in}\ [\![M[x := y]]\!] \\ [\![M\,N]\!] & = & [\![M]\!]\,![\![N]\!] \end{array}$$

We believe that the simplicity of DILL would certainly help in this direction.

We summarize several possible directions that may constitute improvements and extensions to the present "state of the art" of the xLIN machine:

- Several *optimizations* are possible, both to the present version of the xLIN machine, as well as to the implementation techniques used. A starting point is the machine with the two flavours of function abstraction and function application, which are necessary to achieve a parallel between our machine and Krivine's version, i.e., in order to be able to compare the performance of the two in several practical tests. As part of this project, *extensions* may be added using the available techniques.

- The thesis did not consider the issue of *garbage collection*. DILL is too a high-level language, in the sense that no explicit constructs for sharing and discarding variables, i.e., which may be viewed as references to terms, are available as part of the syntax. This has immediate consequences in the abstract machine, which is, as we desired, tightly connected with the structure of the underlying calculus (and hence, the logic). It seems

necessary, then, to recover Weakening and Contraction to have a target language for which to derive a more 'refined' version of the xLIN machine, where garbage collection can be studied.

- We focused on call-by-name, but informally showed how a call-by-need xLIN can be easily obtained by extending it with 'update markers' in the stack that overwrite the environment in order to implement *sharing*, which we did not address as formally as we should. We could easily extend xDILL$_w$ with *addresses*, following the approach of Benaissa et al. [8]. If we restrict ourselves to annotate with addresses only closures, this would be sufficient to characterize the notion of sharing of argument evaluation, and the notion of sharing noted in section 7.3, which avoids copying environments. This is a natural idea, which may be motivated from earlier work done by Launchbury (?), which introduces the notion of *heaps* that associate closures with labels. Since closures are not part of the syntax, Launchbury starts with the untyped λ-calculus, the terms need to be translated to a target language where closures are made explicit.

- The idea of adding substitutions to DILL was not our original idea, although is different from that of de Paiva and Ritter [16], who envisaged the idea. Our version differs from theirs in that we have two (syntactically) *separate versions of substitution*; instead, their environments contain both intuitionistic and linear substitutions. For this reason, their reduction rules are more complicated and the presentation more subtle; for example, the rule for propagating substitutions in application terms (with minor notational differences) suggests that the substitution be split into two, to handle correctly the (potential) occurence of linear substitutions:

  $$(M\,N)\langle\sigma\rangle \to M\langle\sigma'\rangle\,N\langle\sigma''\rangle$$

  Here σ' and σ'' are obtained from σ by 'copying' the intuitionistic substitutions and separating the linear ones in the obvious way.

  In any case, while our approach is more intuitive, theirs is more general (in the spirit of the full $\lambda\sigma_{\Uparrow}$-calculus). It was obtained from a particular categorical model, following previous work of Ritter (?) in the area of categorical abstract machines. Their calculus also has a unique characteristic, namely, that they have several 'sorts' of substitution; these are included to avoid *commuting conversion* reduction rules, which we would have had to deal with if we had considered reduction with open terms.

  We believe that the calculus presented, as well as a stronger version with composition of substitutions, deserves a more through investigation.

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. "Explicit Substitutions." *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] S. Abramsky. "The lazy λ-calculus." In *Declarative Programming.* D. Turner (ed.) Addison-Wesley, 1989.

[3] S. Abramsky. "Computational Interpretations of Linear Logic."

[4] A. Barber. "Dual Intuitionistic Linear Logic." Technical Report. LFCS, University of Edinburgh, 1996.

[5] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Second Edition. Studies in Logic and the Foundations of Mathematics, vol. 103. North-Holland, 1984.

[6] H. P. Barendregt. *Lambda Calculi with Types.* In *Handbook of Logic in Computer Science*, vol. II, 118–309. S. Abramsky, D. M. Gabbay, and T. S. E. Mailbaum (eds.). Oxford University Press, 1992.

[7] N. Benton, G. Bierman, V. de Paiva, and J. M. E. Hyland. "A Term Calculus for Intuitionistic Linear Logic." In *Lecture Notes in Theoretical Computer Science*, vol. 664. Springer Verlag, 1993.

[8] Z. Benaissa, P. Lescanne, and K. Rose. *Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution.* BRICS Research Report RS-1996-56.

[9] G. Bierman and V. de Paiva. "Intuitionistic Necessity Revisited." Technical Report CSRP-96-10, School of Computer Science, University of Birmingham. June 1996.

[10] R. Bloo and K. H. Rose. "Preservation of Strong Normalization in Named Lambda Calculi with Explicit Substitution and Garbage Collection." In *Computer Science in the Netherlands*, November, 1995.

[11] L. Cardelli. "Compiling a Functional Language." *LFP*, 1984.

[12] P. Crégut. "An Abstract Machine for the Normalization of λ-terms." *Proceedings of the 1990 ACM Conference on LISP and Functional Programming,* 333–340, Nice, 1990.

[13] P.-L. Curien, T. Hardin, and J.-J. Lèvy. "Confluence Properties of Weak and Strong Calculi of Explicit Substitutions." *JACM*, 43(2):362–397. April, 1996.

[14] P.-L. Curien. "The $\lambda\rho$-calculus: an abstract framework for environment machines." *Theoretical Computer Science*, 82:389–402, 1991.

[15] V. Danos, J.-B. Joinet and H. Schellinx. "On the linear decoration of intuitionistic derivations." *Archive for Mathematical Logic*, 33:387–412, 1995.

[16] V. de Paiva, N. Ghani, and E. Ritter. *Linear Explicit Substitutions.* Manuscript.

[17] L. Damas and R. Milner. "Principal type schemes for functional programs." In *Conference Records for the Ninth ACM Annual Symposium on Programming Languages*, 207–212, 1982.

[18] N. G. De Bruijn. "Lambda calculus with nameless dummies: a tool for automatic formula manipulation." In R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer (eds.), *Selected Papers on Automath.* Studies in Logic and the Foundations of Mathematics, vol. 133. North-Holland, 1994.

[19] J.-Y. Girard. "Linear Logic." In *Theoretical Computer Science*, 50:1–102, 1997.

[20] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science, vol. 7. Cambridge University Press, 1988.

[21] J. Hannan and D. Miller. "From Operational Semantics to Abstract Machines." *Journal of Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[22] T. Hardin, L. Maranget, and B. Pagano. "Functional Back-Ends within the Lambda-Sigma Calculus." ICFP, 1996.

[23] R. Hindley and J. Seldin. *Introduction to Combinators and λ-calculus.* London Mathematical Society Student texts, vol. 1. Cambridge University Press, 1986.

[24] W. A. Howard. "The formulæ-as-types notion of construction." In J .R .Hindley and J. P. Seldin (eds.), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, 1980.

[25] Y. Lafont. "The linear abstract machine." *Theoretical Computer Science*, 59:157–180, 1988.

[26] P. J. Landin. "The mechanical evaluation of expressions." *Computer Journal*, 6:308–320, 1964.

[27] P. Lescanne. "From $\lambda\sigma$ to $\lambda\upsilon$: a Journey through Calculi of Explicit Substitutions." *POPL*, 1994.

[28] I. Mackie. *Lilac: A Functional Language Based on Linear Logic.* Msc. Thesis, Department of Computing, Imperial College. September, 1991.

[29] R. Milner, M. Tofte, and R. W. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[30] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1990.

[31] G. Plotkin. "LCF Considered as a Programming Language" *Theoretical Computer Science*, 5:223-255, 1977.

[32] D. Prawitz. "Ideas and results in proof theory." In J. E. Fenstad (ed.), *Proceedings of the Second Scandinavian Logic Symposium*, 235–307, 1971.

[33] K. H. Rose. "Explicit cyclic substitutions." *Semantics Note D-166.* DIKU, University of Copenhagen, 1993.

[34] A. Scedrov. *A Guide to Linear Logic.* EATCS, 1990.

[35] Peter Sestoft. "Deriving a Lazy Abstract Machine." Technical Report ID-TR 1994-146. Department of Computer Science, Technical University of Denmark. September, 1994.

[36] Bjarne Stroustrup. *The C++ Programming Language.* Second Edition. Addison-Wesley, 1990.

[37] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge Tracts in Theoretical Computer Science, vol. 43. Cambridge University Press, 1996.

[38] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics: Volume I.* Studies in Logic and the Foundation of Mathematics, vol. 121. North-Holland, 1988.

[39] P. Wadler. "A taste of linear logic." In *Mathematical Foundations of Computer Science*, Gdansk, Poland. LNCS, Springer Verlag. August, 1993.

Appendix A

Formal syntax

This section gives a formal account of the syntax of the xLIN language accepted by the interpreter of chapter 7. An extended BNF notation is used—this is why we have written '$\rightarrow$' instead of '::='. The extensions are $\{\alpha\}^*$ to indicate repetition (a sequence of, possibly zero, α's), and $\{\alpha\}^+$ as a shorthand of $\alpha\ \{\alpha\}^*$ (at least one α), where α stands for anything that can occur at the right-hand side of a production. Also, $[\alpha]$ is used as an abbreviation for $\alpha \mid \langle\text{empty}\rangle$.

The syntax for term expressions, ⟨expression⟩, type expressions, ⟨type⟩, and commands, ⟨command⟩, appears in Figure A.1. The grammar presented contains many redundancies, aimed as improving its readability and serve as documentation. It is also ambiguous (see the production for ⟨application⟩). A non-ambiguous grammar would be clearly obscure, since it would have to explicit, for example, aspects like operator precedence.

The tokens recognized by the lexical analyzer are the punctuation, separator, and operator symbols that appear in the grammar, including keywords like '`let`' and '`in`'. The following are also tokens. The non-terminals ⟨lc-name⟩ and ⟨uc-name⟩ stand for lowercase and uppercase names, respectively, where a name is a sequence of letters. An ⟨integer⟩ is a sequence of digits that may or may not start with a sign ('`+`' or '`-`'). A ⟨boolean⟩ is either of the keywords '`true`' or '`false`'.

⟨input⟩ → `:` ⟨command⟩
| ⟨expression⟩ `;`

⟨expression⟩ → ⟨constant⟩
| ⟨variable⟩
| ⟨primitive⟩
| ⟨abstraction⟩
| ⟨application⟩
| ⟨promotion⟩
| ⟨dereliction⟩
| `(` ⟨expression⟩ `)`

⟨constant⟩ → ⟨integer⟩
| ⟨boolean⟩

⟨variable⟩ → ⟨lc-name⟩

⟨abstraction⟩ → `\` ⟨variable⟩ `:` ⟨type⟩ `.` ⟨expression⟩

⟨application⟩ → ⟨expression⟩ ⟨expression⟩

⟨promotion⟩ → `!` ⟨expression⟩

⟨dereliction⟩ → `let !` ⟨variable⟩ `=` ⟨expression⟩ `in` ⟨expression⟩

⟨type⟩ → ⟨base⟩
| ⟨linear function⟩
| ⟨non-linear function⟩
| ⟨bang⟩
| `(` ⟨type⟩ `)`

⟨base⟩ → ⟨uc-name⟩

⟨linear function⟩ → ⟨type⟩ `-o` ⟨type⟩

⟨non-linear function⟩ → ⟨type⟩ `->` ⟨type⟩

⟨bang⟩ → `!` ⟨type⟩

Figure A.1: The syntax of terms, types, and commands