54

Explicit Substitutions

Martín Abadi, Luca Cardelli,
Pierre-Louis Curien, Jean-Jacques Lévy

February 6, 1990

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Explicit Substitutions

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, Jean-Jacques Lévy

February 6th, 1990

Publication history

This paper was presented at the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, January 1990.

Affiliations

Pierre-Louis Curien is a member of the faculty of Ecole Normale Supérieure, Paris.

Jean-Jacques Lévy is at INRIA Rocquencourt.

Authors' abstract

The $\lambda\sigma$-calculus is a refinement of the λ-calculus where substitutions are manipulated explicitly. The $\lambda\sigma$-calculus provides a setting for studying the theory of substitutions, with pleasant mathematical properties. It is also a useful bridge between the classical λ-calculus and concrete implementations.

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, Jean-Jacques Lévy

Contents

1 Introduction

Substitution is the *éminence grise* of the λ-calculus. The classical β rule,

$$(\lambda x.a)b \rightarrow_{\beta} a\{b/x\}$$

uses substitution crucially though informally. Here a and b denote two terms, and $a\{b/x\}$ represents the term a where all free occurrences of x are replaced with b. This substitution does not belong in the calculus proper, but rather in an informal meta-level. Similar situations arise in dealing with all binding constructs, from universal quantifiers to type abstractions.

A naive reading of the β rule suggests that the substitution of b for x should happen at once, when the rule is applied. In implementations, substitutions invariably happen in a more controlled way. This is due to practical considerations, relevant in the implementation of both logics and programming languages. The term $a\{b/x\}$ may contain many copies of b (for instance, if $a = xxxx$); without sophisticated structure-sharing mechanisms [15], performing substitutions immediately causes a size explosion.

Therefore, in practice, substitutions are delayed and explicitly recorded; the application of substitutions is independent, and not coupled with the β rule. The correspondence between the theory and its implementations becomes highly nontrivial, and the correctness of the implementations can be compromised.

In this paper we study the $\lambda\sigma$-calculus, a refinement of the λ-calculus [1] where substitutions are manipulated explicitly. Substitutions have syntactic representations, and if a is a term and s is a substitution then the term $a[s]$ represents a with the substitution s. We can now express a β rule with delayed substitution, called *Beta*:

$$(\lambda x.a)b \rightarrow_{Beta} a[(b/x) \cdot id]$$

where $(b/x) \cdot id$ is syntax for the substitution that replaces x with b and affects no other variable ("$\cdot$" represents extension and *id* the identity substitution). Of course, additional rules are needed to distribute the substitution later on.

The $\lambda\sigma$-calculus is a suitable setting for studying the theory of substitutions, where we can express and prove desirable mathematical properties. For example, the calculus is Church-Rosser and is a conservative extension of the λ-calculus. Moreover, the $\lambda\sigma$-calculus is strongly connected with the categorical understanding of the λ-calculus, where a substitution is interpreted as a composition [5].

We propose the $\lambda\sigma$-calculus as a step in closing the gap between the classical λ-calculus and concrete implementations. The calculus is a vehicle in designing, understanding, verifying, and comparing implementations of the λ-calculus, from interpreters to machines. Other applications are in the analysis of typechecking algorithms for higher-order languages and, potentially, in the mechanization of logical systems.

When one considers weak reduction strategies, the treatment of substitutions can remain quite simple—and then our approach may seem overly general. Weak reduction strategies do not compute in the scope of λ's. Then, there arise neither nested substitutions nor substitutions in the scope of λ's. All substitutions are at the top level, as simple environments. An ancestor of the $\lambda\sigma$-calculus, the $\lambda\rho$-calculus, suffices in this setting [5].

However, strong reduction strategies are useful in general, both in logics and in the typechecking of higher-order programming languages. In fact, strong reduction strategies are useful in all situations where symbolic matching has to be conducted in the scope of binders. Thus, a general treatment of substitutions is required, where substitutions may occur at the top level and deep inside terms.

In some respects, the $\lambda\sigma$-calculus resembles the calculi of combinators, including those of categorical combinators [4]. The $\lambda\sigma$-calculus and the combinator calculi all give full formal accounts of the process of computation, without suffering from unpleasant complications in the (informal) handling of variables. They all make it easy to derive machines for the λ-calculus and to show the correctness of these machines. From our perspective, the advantage of the $\lambda\sigma$-calculus over combinator calculi is that it remains closer to the original λ-calculus.

There are actually several versions of the calculus of substitutions. We start out by discussing an untyped calculus. The main value of the untyped calculus is for studying evaluation methods. We give reduction rules that extend those of the classical λ-calculus and investigate their confluence. We concentrate on a presentation that relies on De Bruijn's numbering for variables [2], and briefly discuss presentations with more traditional variable names.

Then we proceed to consider typed calculi of substitutions, in De Bruijn notation. We discuss typing rules for a first-order system and for a higher-order system; we prove some of their central properties. The typing rules are meant to serve in designing typechecking algorithms. In particular, their study has been of help for both soundness and efficiency in the design of the Quest typechecking algorithm [3].

We postpone discussion of the untyped calculi to section 3 and of the typed calculi to sections 4 and 5. We now proceed with a general technical overview.

2 Overview

The technical details of the $\lambda\sigma$-calculus can be quite intricate, and hence a gentle informal introduction seems in order. We start with a brief review of De Bruijn notation, since most of our calculi rely on it. Then we preview untyped, first-order, and second-order calculi of substitutions.

2.1 De Bruijn notation

In De Bruijn notation, variable occurrences are replaced with positive integers (called De Bruijn indices); binding occurrences of variables become unnecessary. The positive integer $\mathtt{n}$ refers to the variable bound by the $\mathtt{n}$-th surrounding λ binder, for example:

$$\lambda x.\lambda y.xy \qquad \text{becomes} \qquad \lambda\lambda \mathtt{2}\,\mathtt{1}$$

In first-order typed systems, the binder types must be preserved, for example:

$$\lambda x{:}A.\lambda y{:}B.xy \qquad \text{becomes} \qquad \lambda A.\lambda B.\,\mathtt{2}\,\mathtt{1}$$

In second-order systems, type variables too are replaced with De Bruijn indices:

$$\Lambda A.\lambda x{:}A.x \qquad \text{becomes} \qquad \Lambda\lambda \mathtt{1}.\mathtt{1}$$

Although De Bruijn notation is unreadable, it leads to simple formal systems. Therefore, we use indices in inference rules, but variable names in examples.

Classical β reduction and substitution must be adapted for De Bruijn notation. In order to reduce $(\lambda a)b$, it does not suffice to substitute b into a in the appropriate places. If there are occurrences of $\mathtt{2}$, $\mathtt{3}$, $\mathtt{4}$, $\ldots$ in a, these become "one off," since one of the λ binders surrounding a has been removed. Hence, all the remaining free indices in a must be decremented; the desired effect is obtained with an infinite substitution:

$$(\lambda x.a)b \rightarrow_\beta a\{b/x\} \qquad \text{becomes} \qquad (\lambda a)b \rightarrow_\beta a\{b/1, 1/2, 2/3, \ldots\}$$

When pushing this substitution inside a, we may come across a λ term $(\lambda c)\{b/1, 1/2, 2/3, \ldots\}$. In this case, we must be careful to avoid replacing the occurrences of 1 in c with b, since these occurrences correspond to a bound variable and the substitution should not affect them. Hence, we must "shift" the substitution. Thus, we may try:

$$(\lambda c)\{b/1, 1/2, 2/3, \ldots\} \overset{?}{=} \lambda c\{1/1, b/2, 2/3, 3/4, \ldots\}$$

But this is not yet correct: now b has an additional surrounding binder, and we must prevent capture of free indices of b. Suppose b contains the index 1, for example. We do not want the λ of (λc) to capture this index. Hence we must "lift" all the indices of b:

$$(\lambda c)\{b/1, 1/2, 2/3, \ldots\} = \lambda c\{1/1, b\{2/1, 3/2, \ldots\}/2, 2/3, \ldots\}$$

This informal introduction to De Bruijn notation should suffice to give the flavor of things to come.

2.2 An untyped calculus

We shall study a simple set of algebraic operators that perform all these index manipulations—without ...'s, even though we treat infinite substitutions that replace all indexes. If s represents the infinite substitution $\{a_1/1, a_2/2, a_3/3, \ldots\}$, we write $a[s]$ for a with the substitution s. A term of the form $a[s]$ is called a *closure*. The change from { }'s to [ ]'s emphasizes that the substitution is no longer a meta-level operation.

The syntax of the untyped $\lambda\sigma$-calculus is:

Terms	$a ::= 1 \mid ab \mid \lambda a \mid a[s]$
Substitutions	$s ::= id \mid \uparrow \mid a \cdot s \mid s \circ t$

This syntax corresponds to the index manipulations described in the previous section, as follows:

- id is the identity substitution $\{1/1, 2/2, \ldots\}$, which we may write $\{i/i\}$.

- $\uparrow$ (shift) is the substitution $\{(i+1)/i\}$; for example, $1[\uparrow] = 2$. We need only the index 1 in the syntax of terms; De Bruijn's n+1 is coded as $1[\uparrow^n]$, where $\uparrow^n$ is the composition of n shifts, $\uparrow \circ \ldots \circ \uparrow$. Sometimes we write $\uparrow^0$ for id.

- $i[s]$ is the value of the De Bruijn index i in the substitution s, also informally written $s(i)$ when s is viewed as a function.

- $a \cdot s$ (the cons of a onto s) is the substitution $\{a/1, s(i)/(i+1)\}$; for example,
$$a \cdot id = \{a/1, 1/2, 2/3, \ldots\}$$
and
$$1 \cdot \uparrow = \{1/1, \uparrow(1)/2, \uparrow(2)/3, \ldots\} = id$$

- $s \circ t$ (the composition of s and t) is the substitution such that
$$a[s \circ t] = a[s][t]$$
hence
$$s \circ t = \{s(i)/i\} \circ t = \{s(i)[t]/i\}$$
and, for example,
$$\begin{aligned} id \circ t &= \{id(i)[t]/i\} = \{t(i)/i\} = t \\ \uparrow \circ (a \cdot s) &= \{\uparrow(i)[a \cdot s]/i\} \\ &= \{(i+1)\{a/1, s(i)/(i+1)\}/i\} = \{s(i)/i\} = s \end{aligned}$$

At this point, we have shown most of the algebraic properties of the substitution operations. In addition, composition is associative and distributes over cons (that is, $(a \cdot s) \circ t = a[t] \cdot (s \circ t)$). Moreover, the last example above indicates that $\uparrow \circ s$ is the "rest" of s, without the first component of s; thus, $1[s] \cdot (\uparrow \circ s) = s$.

Using this new notation, we can write the *Beta* rule as
$$(\lambda a)b \rightarrow_{Beta} a[b \cdot id]$$
To complement this rule, we can write rules to evaluate 1, for instance
$$1[c \cdot s] \rightarrow c$$

and rules to push substitution inwards, for instance

$$(cd)[s] \rightarrow (c[s])(d[s])$$

In particular, we can derive an intriguing law for the distribution of substitution over λ:

$$\begin{array}{rcll}
(\lambda c)[s] & = & (\lambda c)\{s(i)/i\} & \\
& = & \lambda c\{1/1, s(i)\{(i+1)/i\}/(i+1)\} & \text{(by previous discussion)} \\
& = & \lambda c\{1/1, s(i)[\uparrow]/(i+1)\} & \text{(by definition of } \uparrow) \\
& = & \lambda c[1 \cdot \{s(i)[\uparrow]/i\}] & \text{(by definition of } \cdot) \\
& = & \lambda c[1 \cdot (s \circ \uparrow)] & \text{(by definition of } \circ)
\end{array}$$

that is,

$$(\lambda c)[s] \rightarrow \lambda c[1 \cdot (s \circ \uparrow)]$$

This last rule uses all the operators (except *id*), and suggests that this choice of operators is natural, perhaps inevitable. In fact, there are many possible variations, but we shall not discuss them here.

Explicit substitutions complicate the structure of bindings somewhat. For example, consider the term

$$(\lambda(1[2 \cdot id]))[a \cdot id]$$

We may be tempted to think that 1 is bound by λ, as it would be in a standard De Bruijn reading. However, the substitution $[2 \cdot id]$ intercepts the index, giving the value 2 to 1. Then, after crossing over λ, the index 2 is renamed to 1 and receives the value a. One should keep these complications in mind in examining $\lambda\sigma$ formulas—for example, in deciding whether a formula is closed, in the usual sense. A precise definition of bindings is as follows.

First, we associate statically (without reduction) a length with each substitution. The length is actually a pair of two integers (m, n). For a substitution of the form $a_1 \cdot \ldots \cdot a_m \cdot (\uparrow \circ \ldots \circ \uparrow)$, we have that m is the number of consed terms and n is the number of $\uparrow$'s. The full definition of the length is:

$$\begin{array}{rcll}
\mid id \mid & = & (0,0) & \\
\mid \uparrow \mid & = & (0,1) & \\
\mid a \cdot s \mid & = & (m+1, n) & \text{where } \mid s \mid = (m,n) \\
\mid s \circ t \mid & = & (m+p-n, q) & \text{where } \mid s \mid = (m,n), \mid t \mid = (p,q), p \geq n \\
\mid s \circ t \mid & = & (m, q+n-p) & \text{where } \mid s \mid = (m,n), \mid t \mid = (p,q), p < n
\end{array}$$

Then, in order to find where a variable n is bound in an expression, we go towards the root of the expression parse tree. We initialize a counter p to n. We decrement it when we cross a λ. If it becomes 0, the λ is the wanted binder. When we reach an a in a closure $a[s]$, with $\mid s \mid = (m_s, n_s)$, we compare p with m_s. If $p \leq m_s$, the variable is bound in s. Otherwise, we continue upwards, setting the counter to $p - m_s + n_s$.

2.3 A first-order calculus

When we move to a typed calculus, we introduce types both in terms and in substitutions. For the typed first-order $\lambda\sigma$-calculus, the syntax becomes:

Types	$A ::= K \mid A \rightarrow B$
Environments	$E ::= nil \mid A, E$
Terms	$a ::= 1 \mid ab \mid \lambda A.a \mid a[s]$
Substitutions	$s ::= id \mid \uparrow \mid a{:}A \cdot s \mid s \circ t$

The environments are used in the type inference rules, as is commonly done, to record the types of the free variables of terms. Naturally, in this setting, environments are indexed by De Bruijn indices. The environment $A_1, A_2, \ldots, A_n, nil$ associates type A_i with index i. For example, the typing axiom for 1 is:

$$A, E \vdash 1 : A$$

and the typing rule for λ abstraction is:

$$\frac{A, E \vdash b : B}{E \vdash \lambda A.b : A \rightarrow B}$$

In the $\lambda\sigma$-calculus, environments have a further function: they serve as the "types" of substitutions. We write $s \triangleright E$ to say that the substitution s "has" the environment E. For example, the typing rule for cons is:

$$\frac{E \vdash a : A \qquad E \vdash s \triangleright E'}{E \vdash (a{:}A \cdot s) \triangleright A, E'}$$

The main use of this new notion is in typing closures. Since s provides the context in which a should be understood, the approach is to compute the environment E' of s, and then type a in that environment:

$$\frac{E \vdash s \triangleright E' \qquad E' \vdash a : A}{E \vdash a[s] : A}$$

An instance of this rule is:

$$\frac{nil \ \vdash \ a{:}A \cdot id \triangleright A, nil \qquad A, nil \ \vdash \ 1 : A}{nil \ \vdash \ 1[a{:}A \cdot id] : A}$$

2.4 A second-order calculus

When we move to a second-order system, new subtleties appear, because substitutions may contain types, and environments may contain placeholders for types; for example,

$$(Bool{::}\mathrm{Ty} \cdot id) \ \triangleright \ \mathrm{Ty}, nil$$

The typing rules become more complex because types may contain type variables, which must be looked up in the appropriate environments. (The problem arises in full generality with dependent types [14], and some readers may find it helpful to think about calculi of substitutions with dependent types.) In particular, the typing axiom for 1 shown above becomes the rule:

$$\frac{E \ \vdash \ A :: \mathrm{Ty}}{A, E \ \vdash \ 1 : A[\uparrow]}$$

The extra shift is required because A is understood in the environment E in the hypothesis, while it is understood in A, E in the conclusion. An alternative (but heavy) solution would be to have separate index sets for ordinary term variables and for type variables, and to manipulate separate term and type environments as well.

Another instance of this phenomenon is in the rule for λ abstraction, which we have also seen above:

$$\frac{A, E \ \vdash \ b : B}{E \ \vdash \ \lambda A.b : A \rightarrow B}$$

Notice that previously A must have been proved to be a type in the environment E, while B is understood in A, E in the assumption. Then $A \rightarrow B$ is understood in E in the conclusion. This means that the indices of B are "one off" in $A \rightarrow B$. The rule for application takes this into account; a substitution is applied to B to "unshift" its indices:

$$\frac{E \vdash b : A \rightarrow B \qquad E \ \vdash \ a : A}{E \ \vdash \ b(a) : B[a{:}A \cdot id]}$$

The $B[a{:}A \cdot id]$ part is reminiscent of the rule found in calculi for dependent types, and this is the correct technique for the version of such calculi with explicit substitutions. However, since here we do not deal with dependent types, B will never contain the index 1, and hence a will never be substituted in B. The substitution is still necessary to shift the other indices in B.

The main difficulty in our second-order calculus arises in typing closures. The approach described for the first order, while still viable, is not sufficient. For example, if *not* is the usual negation on *Bool*, we certainly want to be able to type the term

$$(\lambda 1.not(1))[Bool \cdot id]$$

or, in a more familiar notation,

$$Let\ X = Bool\ in\ \lambda x{:}X.not(x)$$

(We interpret *Let* via a substitution, not via a λ.) Our strategy for the first-order calculus was to type the substitution, obtaining an environment $(X :: \mathrm{Ty}) \cdot id$, and then type the term $\lambda x{:}X.not(x)$ in this environment. Unfortunately, to type this term, it does not suffice to know that X is a type; we must know that X is *Bool*. To solve this difficulty in the second-order system, we have rules to push a substitution inside a term and then type the result. As in calculi with dependent types, the tasks of deriving types and applying substitutions are inseparable.

Finally, as discussed below, surprises arise in writing down the precise rules; for example the rule for typing conses has to be modified. Even the form of the judgement $E \vdash s \triangleright E'$ must be reconsidered.

Higher-order systems, possibly with dependent constructions, are also of theoretical and practical importance. We do not discuss them formally below, however, for we believe that the main complications arise already at the second order.

3 The untyped $\lambda\sigma$-calculus

In this section we present the untyped $\lambda\sigma$-calculus. We propose a basic set of equational axioms for the $\lambda\sigma$-calculus in De Bruijn notation. The equations induce a rewriting system; this rewriting system suffices for the purposes of computation. We show that the rewriting system is confluent, and thus provides a convenient theoretical basis for more deterministic implementations of the $\lambda\sigma$-calculus.

We also consider some variants of the axiom system. Restrictions bring us closer to implementations, as they make evaluation more deterministic. An extension of the system is suggested by Knuth-Bendix computations. Finally, we discuss a $\lambda\sigma$-calculus using variable names.

As in the classical λ-calculus, actual implementations would resort to particular rewriting strategies. We discuss a normal-order strategy for $\lambda\sigma$ evaluation. Then we focus on a more specialized reduction system, still based on normal order, which provides a suitable basis for abstract $\lambda\sigma$ machines. We describe one machine, which extends Krivine's weak reduction machine [13] with strong reduction.

In her study of categorical combinators, Hardin proposed systems similar to ours [8]. In particular, Hardin's system $\mathcal{E} + (Beta)$ is the homomorphic image of our basic system. We rely on some of her techniques to prove our results, and not surprisingly we find confluence properties similar, but not equivalent, to those she found. (We come back to this point below.)

The main difference between the approaches is that in Hardin's work there is a unique sort for terms and substitutions. The distinction between terms and substitutions is central in our work. This distinction is important to a simple understanding of confluence properties and to the practicality of the $\lambda\sigma$-calculus.

Simultaneously with our work, Field developed a system almost identical to our basic system, too, and claimed some of the same results [7]. Thus, we share a starting point. However, Field's paper is an investigation of optimality properties of reduction schemes, so for example Field went on to consider a labelled calculus. In contrast, we are more concerned with questions of confluence and with typechecking issues.

3.1 The basic rewriting system

The syntax of the untyped $\lambda\sigma$-calculus is the one given in the informal overview,

$$\begin{array}{ll} \textbf{Terms} & a ::= 1 \mid ab \mid \lambda a \mid a[s] \\ \textbf{Substitutions} & s ::= id \mid \uparrow \mid a \cdot s \mid s \circ t \end{array}$$

Notice that we have not included metavariables over the sorts of terms and substitutions—we consider only closed terms, and this suffices for our purposes. (In De Bruijn notation, the variables $1, 2, \ldots$ are constants rather than metavariables.)

In this notation, we now define an equational theory for the $\lambda\sigma$-calculus, by proposing a set of equations as axioms. When they are all oriented from left to right, the equations become rewrite rules and give rise to a rewriting system. The equations fall into two subsets: a singleton *Beta*, which is the equivalent of the classical β rule, and ten rules for manipulating substitutions, which we call σ collectively.

Beta	$(\lambda a)b = a[b \cdot id]$
VarId	$1[id] = 1$
VarCons	$1[a \cdot s] = a$
App	$(ab)[s] = (a[s])(b[s])$
Abs	$(\lambda a)[s] = \lambda(a[1 \cdot (s \circ \uparrow)])$
Clos	$a[s][t] = a[s \circ t]$
IdL	$id \circ s = s$
ShiftId	$\uparrow \circ id = \uparrow$
ShiftCons	$\uparrow \circ (a \cdot s) = s$
Map	$(a \cdot s) \circ t = a[t] \cdot (s \circ t)$
Ass	$(s_1 \circ s_2) \circ s_3 = s_1 \circ (s_2 \circ s_3)$

As usual, the equational theory follows from these axioms together with the inference rules for replacing equals for equals.

Our choice of presentation is guided by the structure of terms and substitutions. The *Beta* rule eliminates λ's and creates substitutions; the function of the other rules is to eliminate substitutions. Two rules deal with the evaluation of 1. The next three deal with pushing substitutions inwards. The remaining five express substitution computations. We prove below that the substitution rules always produce unique normal forms; we denote the σ normal form of a by $\sigma(a)$.

The classical β rule is not directly included, but it can be simulated, as we now argue. The precise definition of β reduction, in the style of De Bruijn [2], is as follows:

$$(\lambda a)b \rightarrow_\beta a\{b/1, 1/2, \ldots n/n{+}1, \ldots\}$$

where the meta-level substitution $\{\ldots\}$ is defined inductively by using the rules:

$$\mathrm{n}\{a_1/1,\ldots,a_n/\mathrm{n},\ldots\} = a_n$$

$$\frac{a\{a_1/1,\ldots,a_n/\mathrm{n},\ldots\} = a' \quad b\{a_1/1,\ldots,a_n/\mathrm{n},\ldots\} = b'}{(ab)\{a_1/1,\ldots,a_n/\mathrm{n},\ldots\} = a'b'}$$

$$\frac{a_i\{2/1,\ldots,\mathrm{n{+}1}/\mathrm{n},\ldots\} = a_i' \quad a\{1/1,a_1'/2,\ldots,a_n'/\mathrm{n{+}1},\ldots\} = a'}{(\lambda a)\{a_1/1,\ldots,a_n/\mathrm{n},\ldots\} = \lambda a'}$$

If $a_1,\ldots,a_n,\ldots$ is a sequence of consecutive integers after some point (the only useful case), then the meta-level substitution $\{a_1/1,\ldots,a_n/\mathrm{n},\ldots\}$ corresponds closely to an explicit substitution:

Proposition 3.1 *If there exist m and p such that $a_{m+q} = \mathrm{p}+\mathrm{q}$ for all $q \geq 1$, and $a\{a_1/1,\ldots,a_n/\mathrm{n},\ldots\} = b$ is provable in the formal system presented above, then $\sigma(a[a_1 \cdot a_2 \cdot \ldots \cdot a_m \cdot \uparrow^p]) = b$.*

Proof The argument is by induction on the length of the proof of $a\{a_1/1,\ldots,a_n/\mathrm{n},\ldots\} = b$; we strengthen the claim, and argue that all intermediate terms in the proof satisfy the hypothesis. We omit the easy application case.

Case $\mathrm{n}\{a_1/1,\ldots,a_n/\mathrm{n},\ldots\} = a_n$: If $n \leq m$, then $\mathrm{n}[a_1 \cdot a_2 \cdot \ldots \cdot a_m \cdot \uparrow^p] \rightarrow_\sigma^* a_n$; if $n > m$, then $\mathrm{n}[a_1 \cdot a_2 \cdot \ldots \cdot a_m \cdot \uparrow^p] \rightarrow_\sigma^* n - m + p$. But by hypothesis $a_n = a_{n-m+m} = n - m + p$.

Case $(\lambda a)\{a_1/1,\ldots,a_n/\mathrm{n},\ldots\} = \lambda a'$: By induction on the a_i's (choosing m and p to be 0 and 1), we get by induction $\sigma(a_i[\uparrow]) = a_i'$. This allows us to apply induction on a for $m+1$ and $p+1$:

$$\sigma(a[1 \cdot a_1' \cdot \ldots \cdot a_m' \cdot \uparrow^{p+1}]) = a'$$

On the other hand our desired conclusion reduces to showing

$$\sigma(a[1 \cdot ((a_1 \cdot \ldots \cdot a_m \cdot \uparrow^p) \circ \uparrow)]) = a'$$

which holds since

$$(a_1 \cdot \ldots \cdot a_m \cdot \uparrow^p) \circ \uparrow \rightarrow_\sigma^* a_1[\uparrow] \cdot \ldots \cdot a_m[\uparrow] \cdot \uparrow^{p+1}$$

□

Therefore, the simulation of the β rule consists in first applying *Beta* and then σ until a σ normal form is reached.

As usual, we want a confluence theorem for the calculus. This theorem will guarantee that all rewrite sequences yield identical results, and thus that the strategies used by different implementations are equivalent:

Theorem 3.2 *Beta* + σ *is confluent.*

The proof does *not* rely on standard rewriting techniques, as *Beta* + σ does not pass the Knuth-Bendix test (but σ does). We come back to this subtle point below.

Instead, the proof relies on the termination and confluence of σ, the confluence of the classical λ-calculus, and Hardin's interpretation technique [8].

First we show that σ is noetherian (that is, σ reductions always terminate) and confluent.

Proposition 3.3 σ *is noetherian and confluent.*

Proof We have an indirect proof of noetherianity, as follows. The $\lambda\sigma$-calculus translates into categorical combinators [6], by merging the two sorts of terms and substitutions and collapsing the operations [ ] and $\circ$ into one. Under this translation, a one-step rewriting in σ is mapped to a one-step rewriting of a system *SUBST* of categorical rewriting rules (the exact translation of the largest variant considered in 3.2). Hardin and Laville have established the termination of *SUBST* [9].

Noetherianity simplifies the proof of confluence. By a well-known lemma, local confluence suffices [11]; it can be checked by examining critical pairs, according to the Knuth-Bendix test. For example, for the critical pair

$$(1[id])[s] \rightarrow 1[s] \text{ and } (1[id])[s] \rightarrow 1[id \circ s]$$

local confluence is ensured through the *IdL* rule. □

Since σ is noetherian, let us examine the form of σ normal forms. A substitution in normal form is necessarily in the form

$$a_1 \cdot (a_2 \cdot (\ldots(a_m \cdot U)\ldots))$$

where U is either *id* or a composition $\uparrow \circ (\ldots(\uparrow \circ \uparrow)\ldots)$. A term in normal form is entirely free of substitutions, except in subterms such as $1[\uparrow^n]$, which codes the De Bruijn index n+1. Thus, a term in normal form is a classical λ-calculus term (modulo the equivalence of $1[\uparrow^n]$ and n+1).

In summary, the syntax of σ normal forms is:

Terms	$a ::= 1 \mid 1[\uparrow^n] \mid ab \mid \lambda a$
Substitutions	$s ::= id \mid \uparrow^n \mid a \cdot s$

After these remarks on σ, we can apply Hardin's interpretation technique to show that the full $\lambda\sigma$ system is confluent.

First, we review Hardin's method. Let X be a set equipped with two relations R and S. Suppose that R is noetherian and confluent, and denote by $R(x)$ the R normal form of x; that S_R is a relation included in $(R \cup S)^*$ on the set of R normal forms; and that, for any x and y in X, if $S(x,y)$ then $S_R^*(R(x), R(y))$. An easy diagram chase yields that if S_R is confluent then so is $(R \cup S)^*$.

In our case, we take R to be the relation induced by the σ rules; that is, $R(x,y)$ holds if x reduces to y with the σ rules. We take S_R to be classical β conversion; that is, $S_R(x,y)$ holds if y is obtained from x by replacing a subterm of the form $(\lambda a)b$ with $\sigma(a[b \cdot id])$.

Thus the proof of confluence reduces to the two following lemmas:

Lemma 3.4 *β is confluent on σ normal forms.*

Proof Notice that, on terms, β reduction is the original β reduction, by Proposition 3.1. As for substitutions, since only normal forms are involved, the β reductions are independent β reductions on the components of the substitutions. □

Lemma 3.5

1. *If $a \rightarrow_{Beta} b$ then $\sigma(a) \rightarrow_{\beta}^{*} \sigma(b)$.*
2. *If $s \rightarrow_{Beta} t$ then $\sigma(s) \rightarrow_{\beta}^{*} \sigma(t)$.*

Proof We proceed by induction on the structure of a and s, together.

If a is an application $a_1 a_2$ and if the *Beta* redex is in a_1 or a_2, then the result follows easily from the induction hypothesis, since $\sigma(a_1 a_2) = \sigma(a_1)\sigma(a_2)$. We proceed likewise if a is an abstraction λa_1.

If the *Beta* redex is $a = (\lambda a_1)a_2$, then $b = a_1[a_2 \cdot id]$, and then $\sigma(a) = (\lambda\sigma(a_1))\sigma(a_2)$ By definition of β, we have

$$\sigma(a) \rightarrow_{\beta} \sigma(\sigma(a_1)[\sigma(a_2) \cdot id])$$

that is,

$$\sigma(a) \rightarrow_{\beta} \sigma(b)$$

The last case for terms is $a = a_1[s_1]$. Since $\sigma(a_1[s_1]) = \sigma(\sigma(a_1)[\sigma(s_1)])$, the induction hypothesis reduces our problem to the familiar substitution lemma. De Bruijn proved the following substitution lemma:

> If $a \rightarrow_\beta a'$ then $a\{a_1, a_2, \ldots\} \rightarrow_\beta a'\{a_1, a_2, \ldots\}$. If $a_i \rightarrow_\beta a'_i$ then $a\{a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots\} \rightarrow^*_\beta a\{a_1, \ldots, a_{i-1}, a'_i, a_{i+1}, \ldots\}$.

By Proposition 3.1, this lemma reads, in our notation,

> Suppose a and s are in σ normal form. If $a \rightarrow_\beta a'$, then $\sigma(a[s]) \rightarrow_\beta \sigma(a'[s])$. If $s \rightarrow_\beta s'$, then $\sigma(a[s]) \rightarrow^*_\beta \sigma(a[s'])$.

Thus the lemma settles this case.

The cases for β reductions in substitutions are analogous to those for terms. The case of $a_1 \cdot s_1$ is identical to the one for λa_1. The case of $s_1 \circ s_2$ is similar to the one for $a_1[s_1]$. It suffices to consider the normal forms of s_1 and s_2 for reducing our problem to the substitution lemma, once more. □

3.2 Variants

Some subsystems of σ are reasonable first steps to deterministic evaluation algorithms. We can restrict σ in three different ways. The rule *Clos* can be removed. The inference rule

$$\frac{s = s' \quad t = t'}{s \circ t = s' \circ t'}$$

can be removed, and the inference rule for the closure operator can be restricted to

$$\frac{s = s'}{1[s] = 1[s']}$$

These restrictions (even cumulated) do not prevent us from obtaining σ normal forms and confluence. A general result enables us to derive confluence for these subsystems:

Lemma 3.6 *If S is a subrelation of a noetherian and confluent relation R, and if S normal forms are R normal forms, then S is also confluent. Moreover, the smallest equivalence relations containing R and S coincide.*

Proof If $S^*(a, b)$ and $S^*(a, c)$ then b and c have the same R normal form d, since $S \subseteq R$. However, an S normal form of b (or c) is also an R normal form of b, and thus coincides with d. An almost identical argument establishes the second claim. □

Here we take R and S to be the relations induced by σ and by σ's restriction, respectively. Thus, we easily obtain that the restricted substitution rules are noetherian and confluent, and we can apply the interpretation technique, through exactly the same steps as before. (In fact, the lemmas proved above apply directly, with no modification.)

Confluence properties suggest a second kind of variant. Although $Beta + \sigma$ is confluent, when we view it as a standard rewriting system on first-order terms it is not even locally confluent. The subtle point is that we have proved confluence on closed $\lambda\sigma$ terms, that is, on terms exclusively constructed from the operators of the $\lambda\sigma$-calculus. In contrast, checking critical pairs involves considering open terms over this signature, with metavariables (that is, variables **x** and **u** ranging over terms and substitutions, different from De Bruijn indexes $1, 2, \ldots$).

Consider, for example, the critical pair:

$$\begin{array}{lll} ((\lambda a)b)[u] & \rightarrow^{\star} & a[b[u] \cdot u] \\ ((\lambda a)b)[u] & \rightarrow^{\star} & a[b[u] \cdot (u \circ id)] \end{array}$$

For local confluence, we would want the equation $(s \circ id) = s$, but this equation is not a theorem of σ. Similar critical pair considerations suggest the addition of four new rules:

Id	$a[id] = a$
IdR	$s \circ id = s$
VarShift	$1 \cdot \uparrow = id$
SCons	$1[s] \cdot (\uparrow \circ s) = s$

These additional rules are well justified from a theoretical point of view. However, confluence on closed terms can be established without them, and they are not computationally significant. Moreover, some of them are admissible (that is, every closed instance is provable). More precisely *Id* and *IdR* are admissible in σ, and *SCons* is admissible in σ + *VarShift*.

We should particularly draw attention to the last rule, *SCons*. It expresses that a substitution is equal to its first element appended in front of the rest. This rule is reminiscent of the surjective-pairing rule, which deserved much attention in the classical λ-calculus. Klop has shown that surjective pairing destroys confluence for the λ-calculus [12].

Similarly, we conjecture that the system σ + *Id* + *IdR* + *VarShift* + *SCons* is not confluent when we have metavariables for both terms and substitutions, although it is locally confluent. The following term, inspired by Klop's counterexample [12], seems to work as a counterexample to confluence:

$$Y(Y(\lambda\lambda\mathbf{x}[1[\mathrm{u} \circ (1 \cdot id)] \cdot (\uparrow \circ (\mathrm{u} \circ ((21) \cdot id)))]))$$

where Y is a fixpoint combinator, $\mathbf{x}$ is a term metavariable, and $\mathbf{u}$ is a substitution metavariable. Some work has to be done to check the details. Let us just recall the informal argument. Call $b = Y(c)$ the term above. It reduces to both $\mathbf{x}[\mathrm{u} \circ ((cb) \cdot id)]$ and $c(\mathbf{x}[\mathrm{u} \circ ((cb) \cdot id)])$. To get a common reduct of these two terms, we need to apply *SCons* at some stage, and this requires finding a common reduct of the very same terms. Klop uses standardization to turn this informal circularity argument into a reductio ad absurdum, starting with a minimal length standard reduction to such a common reduct.

The reader may wonder what thwarts the techniques used in the last subsection. The point is that in Lemma 3.5, our reduction to the classical substitution lemma depended crucially on the syntax of substitutions in normal form, which is not so simple any more. (The syntax allows in particular expressions of the form $\mathrm{u} \circ (1 \cdot id)$, as in the suggested counterexample.)

We can go half way in adding metavariables. If we add only term metavariables, the syntax of substitution σ normal forms is unchanged. This protects us from the claimed counterexample. There are two additional cases for term σ normal forms, the cases for metavariables:

Terms $\quad a ::= 1 \mid 1[\uparrow^n] \mid ab \mid \lambda a \mid \mathbf{x} \mid \mathbf{x}[s]$

We believe that confluence can be proved in this case by the interpretation technique. Confluence on normal forms would be obtained through an encoding of the normal forms in the λ-calculus extended with constants, which is known to be confluent ($\mathbf{x}$ becomes a constant; $\mathbf{x}[s]$ becomes a constant applied to the elements of s).

Hardin's results on confluence bear some similarity with ours. In [8], Hardin has shown that various systems are confluent on a set $\mathcal{D}$ of closed terms, which includes the representation of all the usual λ expressions; she found problems with confluence for non-closed terms, too. However, her difficulties and ours differ somewhat, and in particular the counterexamples to confluence differ.

Recently, Hardin and Lévy have succeeded in obtaining confluence with metavariables for both terms and substitutions, by slightly changing the syntax and the set of equations. Their results are reported in [10].

3.3 The $\lambda\sigma$-calculus with names

Let us discuss a more traditional formulation of the calculus, with variable names $x, y, z, \ldots$, as a small digression. Two ways seem viable.

In one approach, we consider the following syntax:

Terms	$a ::= x \mid ab \mid \lambda x.a \mid a[s]$
Substitutions	$s ::= id \mid (a/x) \cdot s \mid s \circ t$

The corresponding theory includes equations such as:

Beta	$(\lambda x.a)b = a[(b/x) \cdot id]$	
Var1	$x[(a/x) \cdot s] = a$	
Var2	$x[(a/y) \cdot s] = x[s]$	$(x \neq y)$
Var3	$x[id] = x$	
App	$(ab)[s] = (a[s])(b[s])$	
Abs	$(\lambda x.a)[s] = \lambda y.(a[(y/x) \cdot s])$	(y occurs in neither a nor s)

The rules correspond closely to the basic ones presented in De Bruijn notation. The *Abs* rule does not require a shift operator, but involves a condition on variable occurrences. (The side condition could be weakened, from y not occurring *at all* in a and s, to y not occurring *free*, in a precise technical sense that we do not define here.) The consideration of the critical pairs generated by the previous rules immediately suggests new rules, such as:

OccT	$a[(b/x) \cdot t] = a[t]$	(x does not occur in a)
OccS	$s \circ ((a/x) \cdot t) = (a/x) \cdot (s \circ t)$	(x does not occur in s)
Comm	$(a/x) \cdot ((b/y) \cdot s) = (b/y) \cdot ((a/x) \cdot s)$	$(x \neq y)$
Alpha	$\lambda x.a = \lambda y.(a[(y/x) \cdot id])$	(y does not occur in a)

This is an unpleasant set of rules. The *Comm* rule destroys the existence of substitution normal forms and the *Alpha* rule expresses renaming of bound variables. Intuitively, we may take this as a hint that this calculus with names does not really enjoy nice confluence features. In this respect, the calculus in De Bruijn notation seems preferable.

There is an alternative solution, with the shift operator. The syntax is now:

Terms	$a ::= x \mid ab \mid \lambda x.a \mid a[s]$
Substitutions	$s ::= id \mid \uparrow \mid (a/x) \cdot s \mid s \circ t$

In this notation, intuitively, $x[\uparrow]$ refers to x after the first binder. The equations are the ones of the $\lambda\sigma$-calculus in De Bruijn notation except for:

Beta	$(\lambda x.a)b = a[(b/x) \cdot id]$	
Var1	$x[(a/x) \cdot s] = a$	
Var2	$x[(a/y) \cdot s] = x[s]$	$(x \neq y)$
Var3	$x[id] = x$	
Abs	$(\lambda x.a)[s] = \lambda x.(a[(x/x) \cdot (s \circ \uparrow)])$	

This framework may be useful for showing the differences between dynamic and lexical scopes in programming languages. The rules here correspond to lexical binding, but dynamic binding is obtained by erasing the shift operator in rule *Abs*.

3.4 A normal-order strategy

As usual, we want a complete rewriting strategy—a deterministic method for finding a normal form whenever one exists. Here we study normal-order strategies, that is, the leftmost-outermost redex is chosen at each step. Completeness is established via the completeness of the normal-order strategy for the λ-calculus.

The normal-order algorithm naturally decomposes into two parts: a routine for obtaining weak head normal forms, and recursive calls on this routine. In our setting, weak head normal forms are defined as follows:

Definition 3.7 *A weak head normal form (whnf for short) is a $\lambda\sigma$ term of the form λa or* $\mathrm{n}a_1 \cdots a_m$.

As a starting point, we take the classical definition of (one step) weak normal-order β reduction $\xrightarrow{n}_\beta$ in the λ-calculus:

$$(\lambda a)b \xrightarrow{n}_\beta \sigma(a[b \cdot id])$$

$$\frac{a \xrightarrow{n}_\beta a'}{ab \xrightarrow{n}_\beta a'b}$$

There are several possibilities for implementing recursive calls, in order to obtain full normal forms; the simplest one consists in adding two rules:

$$\frac{a_i \xrightarrow{n}_\beta a'_i \quad (a_j \text{ in normal form for } j < i)}{\mathbf{n}a_1 \ldots a_i \ldots a_m \xrightarrow{n}_\beta \mathbf{n}a_1 \ldots a'_i \ldots a_m}$$

$$\frac{a \xrightarrow{n}_\beta a'}{\lambda a \xrightarrow{n}_\beta \lambda a'}$$

We do not include these rules, and from now on focus on weak head normal forms—though it is routine to extend the results below to normal forms.

The analogous reduction mechanism for the $\lambda\sigma$-calculus is:

$$(\lambda a)b \xrightarrow{n} a[b \cdot id]$$

$$\frac{a \xrightarrow{n} a'}{ab \xrightarrow{n} a'b}$$

$$1[id] \xrightarrow{n} 1$$

$$1[a \cdot s] \xrightarrow{n} a$$

$$\frac{s \xrightarrow{n} s'}{1[s] \xrightarrow{n} 1[s']}$$

$$(ab)[s] \xrightarrow{n} (a[s])(b[s])$$

$$(\lambda a)[s] \xrightarrow{n} \lambda(a[1 \cdot (s \circ \uparrow)])$$

$$a[s][t] \xrightarrow{n} a[s \circ t]$$

$$id \circ s \xrightarrow{n} s$$

$$\uparrow \circ id \xrightarrow{n} \uparrow$$

$$\uparrow \circ (a \cdot s) \xrightarrow{n} s$$

$$\frac{s \xrightarrow{n} s'}{\uparrow \circ s \xrightarrow{n} \uparrow \circ s'}$$

$$(a \cdot s) \circ t \xrightarrow{n} a[t] \cdot (s \circ t)$$

$$(s \circ s') \circ s'' \xrightarrow{n} s \circ (s' \circ s'')$$

Clearly, $\xrightarrow{n}_\beta$ and $\xrightarrow{n}$ are closely related:

Proposition 3.8 *If $a \xrightarrow{n} b$ then either $\sigma(a) \xrightarrow{n}_\beta \sigma(b)$ or $\sigma(a)$ and $\sigma(b)$ are identical. The $\xrightarrow{n}$ reduction of a terminates (with a weak head normal form) iff the $\xrightarrow{n}_\beta$ reduction of $\sigma(a)$ terminates.*

Proof As for the first part, let $a \xrightarrow{n} b$. If the underlying redex is a σ redex, then obviously $\sigma(a) = \sigma(b)$. If the underlying redex is a *Beta* redex, then a is of the form $(\lambda a_1)a_2 \ldots a_n$, and from $\sigma((\lambda a_1)a_2 \ldots a_n) = (\lambda\sigma(a_1))\sigma(a_2)\ldots\sigma(a_n)$ we can derive $\sigma(a) \xrightarrow{n}_\beta \sigma(b)$.

As for the second part, notice that a $\xrightarrow{n}$ reduction stops exactly when a weak head normal form is reached. Thus, for the "if" part of the claim, it suffices to check that the $\xrightarrow{n}$ reduction of a terminates. We define $\xrightarrow{n}{}^1_\beta$ as the reflexive closure of $\xrightarrow{n}_\beta$. Let

$$a \xrightarrow{n} a_1 \xrightarrow{n} \ldots a_k \xrightarrow{n} \ldots$$

be a $\xrightarrow{n}$ reduction sequence. Then

$$\sigma(a) \xrightarrow{n}{}^1_\beta \sigma(a_1) \xrightarrow{n}{}^1_\beta \ldots \xrightarrow{n}{}^1_\beta \sigma(a_k) \xrightarrow{n}{}^1_\beta \ldots$$

is a $\xrightarrow{n}{}^1_\beta$ reduction sequence, which cannot have infinitely many consecutive reflexive steps because these reflexive steps correspond to σ reductions.

Conversely, suppose that b is a weak head normal form, then $\sigma(b)$ is a weak head normal form. □

Corollary 3.9 *$\xrightarrow{n}$ is a complete strategy.*

Proof This follows from the completeness of the $\xrightarrow{n}_\beta$ strategy. (See [1] for a proof in the classical notation.) □

With the same approach, we can also define a system $\overset{wn}{\rightarrow}$, which incorporates some slight optimizations (present also in our abstract machine, below). In $\overset{wn}{\rightarrow}$, the rule

$$((\lambda a)[s])b \overset{wn}{\rightarrow} a[b \cdot s]$$

replaces the rules

$$(\lambda a)b \overset{n}{\rightarrow} a[b \cdot id]$$

$$(\lambda a)[s] \overset{n}{\rightarrow} \lambda(a[1 \cdot (s \circ \uparrow)])$$

The new rule is an optimization justified by the σ + *IdR* reduction steps

$$\begin{aligned} ((\lambda a)[s])b &\rightarrow (\lambda(a[1 \cdot (s \circ \uparrow)]))b \rightarrow a[1 \cdot (s \circ \uparrow)][b \cdot id] \\ &\rightarrow a[(1 \cdot (s \circ \uparrow)) \circ (b \cdot id)] \rightarrow^* a[b \cdot s] \end{aligned}$$

which is not allowed in $\overset{n}{\rightarrow}$.

Both $\overset{n}{\rightarrow}$ and $\overset{wn}{\rightarrow}$ are weak in the sense that they do not reduce under λ's. In addition, $\overset{wn}{\rightarrow}$ is also weak in the sense that substitutions are not pushed under λ's. In this respect, $\overset{wn}{\rightarrow}$ models environment machines, while $\overset{n}{\rightarrow}$ is closer to combinator reduction machines.

We do not exactly obtain weak head normal forms—in particular, $\overset{wn}{\rightarrow}$ does not reduce even $(\lambda 11)(\lambda 11)$ or $(1[(\lambda 11) \cdot id])(\lambda 11)$. This motivates a syntactic restriction which entails no loss of generality: we start with closures, and all conses have the form $a[s] \cdot t$. Under this restriction, we cannot start with $(\lambda 11)(\lambda 11)$, but instead have to write $((\lambda 11)(\lambda 11))[id]$, which has the expected, nonterminating behavior. The correctness of $\overset{wn}{\rightarrow}$ with respect to normal-order weak head normal form reduction in the λ-calculus can now be proved as in Proposition 3.8.

Proposition 3.10 *If $a \overset{wn}{\rightarrow} b$ then either $\sigma(a) \xrightarrow{n}_\beta \sigma(b)$ or $\sigma(a)$ and $\sigma(b)$ are identical. The $\overset{wn}{\rightarrow}$ reduction terminates (with a term of the form $(\lambda a)[s]$ or $\mathrm{n}a_1 \ldots a_m$) iff the $\xrightarrow{n}_\beta$ reduction of $\sigma(a)$ terminates.*

Proof The proof goes exactly as in Proposition 3.8. The only slight difficulty is in establishing that the $\overset{wn}{\rightarrow}$ reduction terminates exactly on the terms of the form indicated in the statement. The following invariant of the $\overset{wn}{\rightarrow}$ reduction is useful:

For each term b in the $\overset{wn}{\rightarrow}$ reduction sequence starting from $a[s]$,

1. b is a term of the restricted syntax, that is, all subexpressions b'' in contexts $b'' \cdot s''$ are closures;

2. the first node on the spine of b (the leftmost branch of the tree representation of b) that is not an application can only be a closure $b'[s]$ or 1, and all the right arguments of the application nodes above are closures.

We first prove this invariant. We show that if the properties stated hold for b and $b \overset{wn}{\rightarrow} c$ then they hold for c. Notice that the properties are proved together. If the node mentioned in the claim is 1, then the $\overset{wn}{\rightarrow}$ reduction is terminated. If it is a closure $b'[s]$, the proof goes by cases on the structure of b', and if b' is 1 by cases on the structure of s. We detail only two crucial cases, one for each part of the claim. When $b'[s]$ has the form $(\lambda a')[s]$ and is not the root of b, then its immediate context in b has the form $((\lambda a')[s])(a''[s''])$ (by induction hypothesis), and becomes $a'[a''[s'']\cdot s]$. When $b'[s]$ has the form $1[a'[s'] \cdot t]$, then c is b where $b'[s]$ is replaced with $a'[s']$, another closure. (The restriction on the syntax is crucial here.)

Now we derive the claim about $\overset{wn}{\rightarrow}$ normal forms. Suppose that b and $b'[s]$ are as in the statement of the invariant, and that moreover b is not reducible by $\overset{wn}{\rightarrow}$. An easy checking of the rules allows us to exclude the possibility that b' be an application or a closure. It can be 1 only if s' is not further $\overset{wn}{\rightarrow}$ reducible and is not a cons, which forces s' to have the form $\uparrow^k$. Finally, b' can be an abstraction only if $b = b'[s]$. □

3.5 Towards an implementation

As a further refinement towards an implementation, we adapt $\overset{wn}{\rightarrow}$, to manipulate only expressions of the forms $a[t]$ and $s \circ t$. The substitution t corresponds to the "global environment," whereas substitutions deeper in a or s correspond to "local declarations." In defining our machine, we take the view that the linear representation of a can be read as a sequence of machine instructions acting on the graph representation of t.

In this approach, some of the original rules are no longer acceptable, since they do not yield expressions of the desired forms. For example, the reduct of the *App* rule, $(a[s])(b[s])$, is not a closure. In order to reduce $(ab)[s]$, we have to reduce $a[s]$ to a weak head normal form first. In the machine discussed below, we use a stack for storing $b[s]$.

The following reducer *whnf*() embodies these modifications to $\overset{wn}{\rightarrow}$. The reducer takes a pair of arguments, the term a and the substitution s of a closure, and returns another pair, of the form $(\mathrm{n}a_1 \cdots a_m, id)$ or $(\lambda a', s')$. To compute *whnf*(), the following axioms and rules should be applied, in the order of their listing. We proceed by cases on the structure of a, and when a is n by cases on the structure of s, and when s is a composition $t \circ t'$ by cases on the structure of t.

$$whnf(\lambda a, s) = (\lambda a, s)$$

$$\frac{whnf(a, s) = (\lambda a', s')}{whnf(ab, s) = whnf(a', b[s] \cdot s')}$$

$$\frac{whnf(a, s) = (a', id) \quad (a' \text{ not an abstraction})}{whnf(ab, s) = (a'(b[s]), id)}$$

$$whnf(\mathrm{n}, id) = (\mathrm{n}, id)$$

$$whnf(\mathrm{n}, \uparrow) = (\mathrm{n+1}, id)$$

$$whnf(1, a[s] \cdot t) = whnf(a, s)$$

$$whnf(\mathrm{n+1}, a \cdot s) = whnf(\mathrm{n}, s)$$

$$whnf(\mathrm{n}, s \circ s') = whnf(\mathrm{n}[s], s')$$

$$whnf(\mathrm{n}[id], s) = whnf(\mathrm{n}, s)$$

$$whnf(\mathrm{n}[\uparrow], s) = whnf(\mathrm{n+1}, s)$$

$$whnf(1[a \cdot s], s') = whnf(a, s')$$

$$whnf(\mathrm{n+1}[a \cdot s], s') = whnf(\mathrm{n}[s], s')$$

$$whnf(\mathrm{n}[s \circ s'], s'') = whnf(\mathrm{n}[s], s' \circ s'')$$

$$whnf(a[s], s') = whnf(a, s \circ s')$$

A simple extension of these rules yields full normal forms:

$$\frac{whnf(a, s) = (\lambda a', t)}{nf(a, s) = \lambda(nf(a', 1 \cdot (t \circ \uparrow)))}$$

$$\frac{whnf(a, s) = (\mathrm{n}(a_1[s_1]) \ldots (a_m[s_m]), id)}{nf(a, s) = \mathrm{n}(nf(a_1, s_1)) \ldots (nf(a_m, s_m))}$$

The precise soundness property of *whnf*() is:

Proposition 3.11 *The equation $whnf(a, s) = (a', s')$ is provable if and only if $\sigma(a'[s'])$ is the weak head normal form of $\sigma(a[s])$.*

Proof It is routine to check the correctness of *whnf*() with respect to $\overset{wn}{\rightarrow}$. Specifically, $whnf(\mathrm{n}, s) = (a', s')$ is provable iff $a'[s']$ is the $\overset{wn}{\rightarrow}$ normal form of $1[(\uparrow \circ (\ldots(\uparrow \circ s)\ldots))]$ (with $n-1$ $\uparrow$'s); $whnf(\mathrm{n}[t], s) = (a', s')$ is provable iff $a'[s']$ is the $\overset{wn}{\rightarrow}$ normal form of $1[(\uparrow \circ (\ldots(\uparrow \circ (t \circ s))\ldots))]$ (with $n-1$ $\uparrow$'s); in all other cases, $whnf(a, s) = (a', s')$ is provable iff $a'[s']$ is the $\overset{wn}{\rightarrow}$ normal form of $a[s]$. □

The last step we consider is the derivation of a transition machine from the rules for *whnf*(). One basic idea is to implement the recursive call on $a[s]$ during the evaluation of $(ab)[s]$ by using a stack to store the argument $b[s]$. Thus, the stack contains closures.

The following table represents an extension of Krivine's abstract machine [13, 5]. The first column represents the "current state," the second one represents the "next state." Each line has to be read as a transition from a triplet (Subst, Term, Stack) to a triplet of the same nature. To evaluate a program a in the global environment s, the machine is started in state $(s, a, \langle\,\rangle)$, where $\langle\,\rangle$ is the empty stack. The machine repeatedly uses the first applicable rule. The machine stops when no transition is applicable any more. These termination states have one of the forms $(id, \mathrm{n}, a_1 \cdot \cdots \cdot a_m)$ and $(s, \lambda a, \langle\,\rangle)$, which represent $\mathrm{n}a_1 \cdots a_m$ and $(\lambda a)[s]$, respectively.

Subst	Term	Stack	Subst	Term	Stack
$\uparrow$	n	S	id	$\mathrm{n+1}$	S
$a[s] \cdot t$	1	S	s	a	S
$a \cdot s$	$\mathrm{n+1}$	S	s	n	S
$s \circ s'$	n	S	s'	$\mathrm{n}[s]$	S
s	ab	S	s	a	$b[s] \cdot S$
s	λa	$b[t] \cdot S$	$b[t] \cdot s$	a	S
s	$\mathrm{n}[id]$	S	s	n	S
s	$\mathrm{n}[\uparrow]$	S	s	$\mathrm{n+1}$	S
s'	$1[a \cdot s]$	S	s'	a	S
s'	$\mathrm{n+1}[a \cdot s]$	S	s'	$\mathrm{n}[s]$	S
s''	$\mathrm{n}[s \circ s']$	S	$s' \circ s''$	$\mathrm{n}[s]$	S
s'	$a[s]$	S	$s \circ s'$	a	S

The machine can be restarted when it stops, and then we have a full normal form λ reducer. Specifically, when the machine terminates with the triplet $(s, \lambda a, \langle\,\rangle)$, we restart it in the initial state $(1 \cdot (s \circ \uparrow), a, \langle\,\rangle)$, and when the machine terminates with the triplet $(id, \mathrm{n}, a_1[s_1] \cdot \ldots \cdot a_n[s_n] \cdot \langle\,\rangle)$, we restart n copies of the machine in the states $(s_1, a_1, \langle\,\rangle), \ldots, (s_n, a_n, \langle\,\rangle)$.

The correctness of the machine can be stated as follows (we omit the simple proof).

Proposition 3.12 *Starting in the state $(s, a, \langle\,\rangle)$, the machine terminates in $(id, \mathrm{n}, a_1 \cdot \ldots \cdot a_m)$ iff $whnf(a, s) = (\mathrm{n}a_1 \ldots a_m, id)$, and terminates in $(s, \lambda a, \langle\,\rangle)$ iff $whnf(a, s) = (\lambda a, s)$.*

By now, we are far away from the wildly nondeterministic basic rewriting system of Section 3.1. However, through the derivations, we have managed to keep some understanding of the successive refinements and to guarantee their correctness. This has been possible because the $\lambda\sigma$-calculus is more concrete than the λ-calculus, and hence an easier starting point.

4 First-order theories

In the previous section, we have seen how to derive a machine that can be used as a sensible implementation of the untyped $\lambda\sigma$-calculus, and in turn of the untyped λ-calculus. Different implementation issues arise in typed systems. For typed calculi, we need not just an execution machine, but also a typechecker. As will become apparent when we discuss second-order systems, explicit substitutions can also help in deriving typecheckers. Thus, we want a typechecker for the $\lambda\sigma$-calculus.

At the first order, the typechecker does not present much difficulty. In addition to the usual rules for a classical system L1, we must handle the typechecking of substitutions. Inspection of the rules of L1 shows that this can be done easily, since the rules are deterministic.

In this section we describe the first-order typed $\lambda\sigma$-calculus. We prove that it preserves types under reductions, and that it is sound with respect to the λ-calculus. We move on to the second-order calculus in the next section.

We start by recalling the syntax and the type rules of the first-order λ-calculus with De Bruijn's notation.

Types	$A ::= K \mid A \rightarrow B$
Environments	$E ::= nil \mid A, E$
Terms	$a ::= \mathrm{n} \mid \lambda A.a \mid ab$

Definition 4.1 (Theory L1)

(L1-var) $A, E \vdash 1 : A$

(L1-varn) $$\frac{E \vdash n : B}{A, E \vdash n{+}1 : B}$$

(L1-lambda) $$\frac{A, E \vdash b : B}{E \vdash \lambda A.b : A \to B}$$

(L1-app) $$\frac{E \vdash b : A \to B \qquad E \vdash a : A}{E \vdash ba : B}$$

We do not include the β rule, because we now focus on typechecking—rather than on evaluation.

The first-order $\lambda\sigma$-calculus has the following syntax:

Types	$A ::= K \mid A \to B$
Environments	$E ::= nil \mid A, E$
Terms	$a ::= 1 \mid ab \mid \lambda A.a \mid a[s]$
Substitutions	$s ::= id \mid \uparrow \mid a{:}A \cdot s \mid s \circ t$

The type rules come in two groups, one for giving types to terms, and one for giving environments to substitutions. The two groups interact through the rule for closures.

Definition 4.2 (Theory S1)

(S1-var) $A, E \vdash 1 : A$

(S1-lambda) $$\frac{A, E \vdash b : B}{E \vdash \lambda A.b : A \to B}$$

(S1-app) $$\frac{E \vdash b : A \to B \qquad E \vdash a : A}{E \vdash ba : B}$$

(S1-clos) $$\frac{E \vdash s \triangleright E' \qquad E' \vdash a : A}{E \vdash a[s] : A}$$

(S1-id) $\quad E \vdash id \triangleright E$

(S1-shift) $\quad A, E \vdash \uparrow \triangleright E$

(S1-cons) $$\frac{E \vdash a : A \qquad E \vdash s \triangleright E'}{E \vdash a{:}A \cdot s \triangleright A, E'}$$

(S1-comp) $$\frac{E \vdash s'' \triangleright E'' \qquad E'' \vdash s' \triangleright E'}{E \vdash s' \circ s'' \triangleright E'}$$

In S1, we include neither the *Beta* axiom nor the σ axioms.

Clearly, typechecking is decidable in S1. Furthermore, the fact that we can separate typing of terms from typing of substitutions is quite pleasant; as we have seen, this property does not extend to the second order.

We proceed to show that S1 is sound. As a preliminary, we prove two lemmas. The first lemma relies on the notion of σ normal form, which was defined in the previous section. We use a modified version of the σ rules, in order to deal with typed terms; four of the rules change.

VarCons	$1[a{:}A \cdot s] = a$
Abs	$(\lambda A.a)[s] = \lambda A.(a[1{:}A \cdot (s \circ \uparrow)])$
ShiftCons	$\uparrow \circ (a{:}A \cdot s) = s$
Map	$(a{:}A \cdot s) \circ t = a[t]{:}A \cdot (s \circ t)$

The typed version of σ enjoys the properties of the untyped version.

A term in σ normal form is typeable in S1 iff it is typeable in L1:

Lemma 4.3 (Same theory on normal forms) *Let a be in σ normal form. Then $E \vdash_{S1} a{:}A$ iff $E \vdash_{L1} a{:}A$.*

Proof The argument is an easy induction on the length of proofs. The only delicate case is the one that deals with the rules L1-varn and S1-clos.

First, we assume that $A, E \vdash_{L1}$ n+1 : B, and show that $A, E \vdash_{S1}$ n+1 : B. Since $A, E \vdash_{L1}$ n+1 : B, it must be that $E \vdash_{L1}$ n : B. By induction hypothesis, $E \vdash_{S1}$ n : B. Unless n is 1 (a trivial case), the last rule in the S1 proof could only be S1-clos, and then it must be that $E \vdash_{S1} \uparrow^{n-1} \triangleright E'$

and $E' \vdash_{S1}$ $1 : B$ for some E'. In fact, it must be that $E \vdash_{S1} \uparrow^{n-1} \triangleright B, E''$ and B, E'' $\vdash_{S1}$ $1 : B$ for some E''. Then S1-shift and S1-comp yield $A, E \vdash_{S1} \uparrow^{n} \triangleright B, E''$, and S1-clos yields $A, E \vdash_{S1} 1[\uparrow^{n}] : B$, the desired result.

For the converse, we assume that $E \vdash_{S1}$ n+1 : B, in order to show that $E \vdash_{L1}$ n+1 : B. Since $E \vdash_{S1}$ n+1 : B, it must be that $E \vdash_{S1} \uparrow^{n} \triangleright E'$ and $E' \vdash_{S1}$ $1 : B$ for some E' (unless n is 1, a trivial case). Further analysis shows that E must be of the form C, E'' and that $E'' \vdash_{S1} \uparrow^{n-1} : B, E_0$, and hence $E'' \vdash_{S1}$ n : B. The proof of this last theorem is shorter than the proof of $E \vdash_{S1}$ n+1 : B. By induction hypothesis, it follows that $E'' \vdash_{L1}$ n : B, and then $C, E'' \vdash_{L1}$ n+1 : B, that is, $E \vdash_{L1}$ n+1 : B. □

Let $\rightarrow_\sigma$ denote one-step reduction with the σ rules; σ reductions preserve typings in S1.

Lemma 4.4 (Subject reduction) *If $a \rightarrow_\sigma a'$ and $E \vdash_{S1} a : A$, then $E \vdash_{S1} a' : A$. Similarly, if $s \rightarrow_\sigma s'$ and $E' \vdash_{S1} s \triangleright E''$, then $E' \vdash_{S1} s' \triangleright E''$.*

Proof We inspect the σ rules one by one; we abbreviate $\vdash_{S1}$ as $\vdash$.

Var: Let $1[b{:}B \cdot s] \rightarrow_\sigma b$. Suppose $E \vdash 1[b{:}B \cdot s] : A$. By S1-clos, $E \vdash b{:}B \cdot s \triangleright E1$ and $E1 \vdash 1{:}A$, for some $E1$. Furthermore, by S1-cons, $E \vdash b{:}B \cdot s \triangleright B, E2$, with $E1 = B, E2$, with $E \vdash b{:}B$, and with $E \vdash s \triangleright E2$. By S1-var, $B, E2 \vdash 1{:}A$ implies $B = A$, and thus $E \vdash b{:}A$.

App: Let $ba[s] \rightarrow_\sigma (b[s])(a[s])$. Suppose $(ba)[s] : B$. By S1-clos, $E \vdash s \triangleright E1$ and $E1 \vdash ba : B$, and hence $E1 \vdash b : A \rightarrow B$ and $E1 \vdash a : A$. By S1-clos, moreover, $E \vdash b[s] : A \rightarrow B$ and $E \vdash a[s] : A$. Therefore, $E \vdash (b[s])(a[s]) : B$.

Abs: Let $(\lambda A.b)[s] \rightarrow_\sigma \lambda A.(b[1 : A \cdot (s \circ \uparrow)])$. Suppose $(\lambda A.b)[s] : C$. By S1-clos, $E \vdash s \triangleright E1$ and $E1 \vdash \lambda A.b : C$. By S1-lambda, $C = A \rightarrow B$ and $A, E1 \vdash b : B$. Now, we apply S1-shift and S1-comp to obtain $A, E \vdash \uparrow \triangleright E$. and then $A, E \vdash s \circ \uparrow \triangleright E1$. Since $A, E \vdash 1 : A$ by S1-var, S1-cons gives us $A, E \vdash 1{:}A \cdot s \circ \uparrow \triangleright A, E1$. Finally, since $A, E1 \vdash b : B$, S1-clos yields $A, E \vdash b[1{:}A \cdot s \circ \uparrow] : B$, and therefore, $\lambda A.(b[1{:}A \cdot (s \circ \uparrow)]) : A \rightarrow B$ by S1-lambda.

Clos: Let $(b[s])[t] \rightarrow_\sigma b[s \circ t]$. Suppose $E \vdash (b[s])[t] : B$. Then $E \vdash t \triangleright E1$ and $E1 \vdash b[s] : B$, that is, $E1 \vdash s \triangleright E2$ and $E2 \vdash b : B$. S1-comp tells us $E \vdash s \circ t \triangleright E2$, and then $E \vdash b[s \circ t] : B$ by S1-clos.

IdL: Let $id \circ s \rightarrow_\sigma s$. Suppose $E \vdash id \circ s \triangleright E'$. Then $E \vdash s \triangleright E''$ and $E'' \vdash id \triangleright E'$, by S1-comp, and $E'' = E'$ by S1-id. Finally, $E \vdash s \triangleright E'$.

ShiftCons: Let $\uparrow \circ (a{:}A \cdot s) \rightarrow_\sigma s$. Suppose $E \vdash \uparrow \circ (a{:}A \cdot s) \triangleright E'$. Then $E \vdash a{:}A \cdot s \triangleright E''$ and $E'' \vdash \uparrow \triangleright E'$, by S1-comp. S1-cons says $E \vdash a{:}A$ and $E \vdash s \triangleright E1$, with $E'' = A, E1$. By S1-shift, we have $E'' = A, E'$. Therefore, $E1 = E'$ and $E \vdash s \triangleright E'$.

Ass: Let $(s_1 \circ s_2) \circ s_3 \rightarrow_\sigma s_1 \circ (s_2 \circ s_3)$. To solve this case, we simply use S1-comp twice.

Map: Let $(a{:}A \cdot s) \circ t \rightarrow_\sigma A[t] \cdot (s \circ t)$. Suppose $E \vdash (a{:}A \cdot s) \circ t \triangleright E'$. Then $E \vdash t \triangleright E''$ and $E'' \vdash a{:}A \cdot s \triangleright E'$, by S1-comp. Hence, by S1-cons, $E'' \vdash a : A$ and $E'' \vdash s \triangleright E1$, with $E' = A, E1$. Then $E \vdash s \circ t \triangleright E1$ by S1-comp, and $E \vdash a[t] : A$ by S1-clos. Finally, $E \vdash a[t]{:}A \cdot (s \circ t) \triangleright A, E1$, by S1-cons.

IdR: Let $s \circ id \rightarrow_\sigma s$. This case is similar to the case for *IdL*.

Id: Let $a[id] \rightarrow_\sigma a$. Suppose $E \vdash a[id] : A$. Then $E \vdash id \triangleright E'$ and $E' \vdash a{:}A$ by S1-clos. S1-id implies $E' = E$. Thus, $E \vdash a{:}A$.

VarShift: Let $1{:}A \cdot \uparrow \rightarrow_\sigma id$. Suppose $E \vdash 1{:}A \cdot \uparrow \triangleright E'$. By S1-cons, $E \vdash 1{:}A$ and $E \vdash \uparrow \triangleright E''$, with $E' = A, E''$. S1-var yields $E = A, E1$, and S1-shift yields $E1 = E''$. Finally, by S1-id, $A, E1 \vdash id : A, E1$, that is, $E : id \triangleright E'$.

SCons: Let $(1{:}A)[s] \cdot (\uparrow \circ s) \rightarrow_\sigma s$. This case is similar to the previous one.
□

Together, the two lemmas immediately give us soundness:

Proposition 4.5 (Soundness) *If* $E \vdash_{S1} a : A$, *then* $E \vdash_{L1} \sigma(a) : A$.

One may wonder whether a completeness result holds, as a converse to our soundness result. Unfortunately, the answer is no. For instance, if L1 gives a type to a but not to b, then S1 cannot give a type to $1[a{:}A \cdot (b{:}B \cdot id)]$, while L1 gives a type to $\sigma(1[a{:}A \cdot (b{:}B \cdot id)])$, that is, to a.

However, if L1 gives types to both a and b, then S1 can give a type to $1[a{:}A \cdot (b{:}B \cdot id)]$. Conversely, if S1 can give a type to $1[a{:}A \cdot (b{:}B \cdot id)]$, then L1 can give types to both a and b.

These observations suggest a reformulation of the soundness and completeness claim. Informally, one would like to show that S1 can give a type to a term iff L1 can give a type to the normal forms of the term and of some subterms that σ normalization discards.

5 Second-order theories

Type rules and typecheckers are also needed for second-order calculi. Unfortunately, the situation is more complex than at the first order, because types include binding constructs (quantifiers). These interact with substitutions in the same subtle ways in which λ interacts with substitutions. (We have no equivalent of β reduction here, but this too reappears in higher-order typed systems.)

In implementing a typechecker (or proofchecker) for the second or higher orders, we face the same concerns of efficient handling of substitution and correctness of implementation that pushed us from the untyped λ-calculus to the untyped $\lambda\sigma$-calculus. These are important concerns in typechecking Quest programs, for example. It is nice to discover that we can apply the same concept of explicit substitutions to tackle typechecking problems as well.

In order to carry out this plan, we must first obtain a second-order system with explicit substitutions, which already incurs several difficulties. Then we must refine the system, and obtain an actual typechecking algorithm. During this long enterprise, where many steps are interesting for their own sake, we should keep in mind the goal of deriving an algorithm that is correct and close to a sensible implementation by virtue of handling substitutions explicitly.

Second-order theories are considerably more complex than untyped or first-order theories, both in number of rules and in subtlety. The complication is already apparent in the De Bruijn formulation of the ordinary second-order λ-calculus (L2, below). The complication intensifies in the second-order $\lambda\sigma$-calculus (S2) because of unexpected difficulties. (We have mentioned some of them in the informal overview.)

We begin with a description of L2, then we define S2 and prove that it is sound with respect to L2. Unlike L1, L2, and even S1, the new system S2 is not deterministic. Therefore, we also define a second-order typechecking algorithm S2alg, and prove that it is sound with respect to S2.

The syntax for the second-order λ-calculus is:

Types	$A ::= \mathrm{n} \mid A \rightarrow B \mid \forall A$
Environments	$E ::= \mathit{nil} \mid A, E \mid \mathrm{Ty}, E$
Terms	$a ::= \mathrm{n} \mid \lambda A.a \mid \Lambda a \mid ab \mid aB$

The system L2 consists of the type rules for the second-order λ-calculus:

Definition 5.1 (Theory L2)

(L2-nil) $\vdash \mathit{nil}\ \ \mathrm{env}$

(L2-ext) $$\frac{\vdash E\ \ \mathrm{env} \qquad E \vdash A :: \mathrm{Ty}}{\vdash A, E\ \ \mathrm{env}}$$

(L2-ext2) $$\frac{\vdash E\ \ \mathrm{env}}{\vdash \mathrm{Ty}, E\ \ \mathrm{env}}$$

(L2-tvar) $$\frac{\vdash E\ \ \mathrm{env}}{\mathrm{Ty}, E \vdash 1 :: \mathrm{Ty}}$$

(L2-tvarn) $$\frac{E \vdash \mathrm{n} :: \mathrm{Ty} \qquad E \vdash A :: \mathrm{Ty}}{A, E \vdash \mathrm{n{+}1} :: \mathrm{Ty}}$$

(L2-tvarn2) $$\frac{E \vdash \mathrm{n} :: \mathrm{Ty}}{\mathrm{Ty}, E \vdash \mathrm{n{+}1} :: \mathrm{Ty}}$$

(L2-tfun) $$\frac{E \vdash A :: \mathrm{Ty} \qquad A, E \vdash B :: \mathrm{Ty}}{E \vdash A \rightarrow B :: \mathrm{Ty}}$$

(L2-tgen) $$\frac{\mathrm{Ty}, E \vdash B :: \mathrm{Ty}}{E \vdash \forall B :: \mathrm{Ty}}$$

(L2-var) $$\frac{E \vdash A :: \mathrm{Ty}}{A, E \vdash 1 : A\{\uparrow\}}$$

(L2-varn) $$\frac{E \vdash \mathrm{n} : B \qquad E \vdash A :: \mathrm{Ty}}{A, E \vdash \mathrm{n{+}1} : B\{\uparrow\}}$$

(L2-varn2) $$\frac{E \vdash \mathtt{n} : B}{\mathrm{Ty}, E \vdash \mathtt{n+1} : B\{\uparrow\}}$$

(L2-lambda) $$\frac{A, E \vdash b : B}{E \vdash \lambda A.b : A \rightarrow B}$$

(L2-Lambda) $$\frac{\mathrm{Ty}, E \vdash b : B}{E \vdash \Lambda b : \forall B}$$

(L2-app) $$\frac{E \vdash b : A \rightarrow B \qquad E \vdash a : A}{E \vdash b(a) : B\{a{:}A \cdot id\}}$$

(L2-App) $$\frac{E \vdash b : \forall B \qquad E \vdash A :: \mathrm{Ty}}{E \vdash b(A) : B\{A{::}\mathrm{Ty} \cdot id\}}$$

We now move on to the S2 system, with the following syntax:

Types	$A ::= 1 \mid A \rightarrow B \mid \forall A \mid A[s]$
Environments	$E ::= nil \mid A, E \mid \mathrm{Ty}, E$
Terms	$a ::= 1 \mid \lambda A.a \mid \Lambda a \mid ab \mid aB \mid a[s]$
Substitutions	$s ::= id \mid \uparrow \mid a{:}A \cdot s \mid A{::}\mathrm{Ty} \cdot s \mid s \circ t$

In the previous section, we have seen how to formulate a first-order $\lambda\sigma$-calculus (S1) by adding one closure rule and a group of substitution rules to the first-order λ-calculus (L1). Unfortunately, this approach fails for second-order systems, as it would not provide a satisfactory treatment of definitional equality. In L1, we can simply define a *let* construct in terms of either abstraction and application, or substitution:

$$let\ x{:}A = a\ in\ b \ =_{def}\ (\lambda x{:}A.b)a \text{ or } b\{a/x\}$$

In L2, we can accept this definition of *let*, and also define a *Let* construct for giving names to types, by substitution:

$$Let\ X = A\ in\ b \ =_{def}\ b\{A/X\}$$

However, it is not adequate to define *Let* as an abbreviation for abstraction and application. For instance, recall the example given in the informal overview: $Let\ X = Bool\ in\ \lambda x{:}X.not(x)$ cannot be typed if it is interpreted as $(\Lambda X.\lambda x:X.not(x))Bool$. Here the body of *Let* can only be typechecked by knowing that $X = Bool$; it does not suffice to have $X::\mathrm{Ty}$. Thus, we must interpret *Let* with a substitution.

Unfortunately, this strategy does not carry over to S2. First, we cannot define *Let* in S2 with a meta-level substitution, because the whole point of S2 is to deal with explicit substitutions. Second, if we define *Let* with an explicit substitution, we obtain:

$$Let\ X = A\ in\ b\ \ =_{def}\ \ b[(A::\mathrm{Ty}/X)\cdot id]$$

and, for example,

$$Let\ X = Bool\ in\ \lambda x:X.not(x)\ \ =_{def}\ \ (\lambda x{:}X.not(x))[(Bool::\mathrm{Ty}/X)\cdot id]$$

We still cannot type the body of *Let* independently, before pushing the substitution into it. We are in no better shape than with the encoding of *Let* via Λ. Hence, it does not suffice to deal with terms and substitutions separately, as we did in the S1-clos rule of the previous section. The task of deriving types cannot be separated from the task of applying substitutions. The rules of S2 described below are structured in such a way that substitutions are automatically pushed inside terms during typechecking, so that typing can occur as expected in the example above. The unfortunate side effect is a small explosion in the number of rules. We do not include an analogue for S1-clos (in fact, we conjecture that it is admissible).

After having settled on a general approach, let us discuss the form of judgments. The theory S2 is formulated with equivalence judgments, for example judgments of the form $E\ \vdash\ a \sim b : A$. This judgment means that in the environment E the terms a and b both have type A and are equivalent. We can recover the standard judgments, with definitions such as

$$E \vdash a : A\ \ =_{def}\ \ E \vdash a \sim a : A$$

In S2, equivalence judgments are needed because it is not always possible to prove directly $E\ \vdash\ a : A$, but only $E\ \vdash\ b : A$ for a term b that is σ-equivalent to a (as in the example above). Formally, in order to prove $E \vdash a \sim a : A$, we first prove $E \vdash a \sim b : A$, and then use symmetry and transitivity. Similarly, it is not always possible to prove directly $E\ \vdash\ a : A$, but instead we may have to prove $E\ \vdash\ a : B$ for a type B that is σ-equivalent to A, and then we need to "retype" a from B to A.

We have seen in section 2 how the typing axiom for 1 has to be modified. Similar considerations show that the rule for conses, S1-cons, needs to be modified as well, and suggest the following, tentative rule:

$$\frac{E \ \vdash \ a \sim b : A[s] \qquad E \ \vdash \ s \sim t \triangleright E' \qquad E \ \vdash \ A[s] \sim B[t] :: \mathrm{Ty}}{E \ \vdash \ (a{:}A \cdot s) \sim (b{:}B \cdot t) \triangleright A, E'}$$

Note that, in the hypothesis, we require that a have type $A[s]$ rather than A: the reason is that A is well-formed in E' rather than in E. Furthermore, we require that s and t be equivalent substitutions of type E', but in truth their type is irrelevant. This suggests a new approach: we deal with judgments of the form

$$E \ \vdash \ s \sim t \quad subst_p$$

where p records the length $\mid E' \mid$ of E'. (The precise relation between environment lengths, and substitutions sizes, as defined in section 2, obeys the invariant: if $E \ \vdash \ s \quad subst_p$ and $\mid s \mid = (m, n)$ then $p = m + \mid E \mid - n \geq 0$.)

In fact, we could hardly do more than keep track of the lengths of substitutions. As the following example illustrates, the type of a substitution cannot be determined satisfactorily. In the tentative rule above, let $E = nil$, $s = t = Bool{::}\mathrm{Ty} \cdot id$, $a = b = true$, $A = 1$, and $B = Bool$. We obtain

$$nil \ \vdash \ (true{:}1 \cdot s) \sim (true{:}Bool \cdot t) \triangleright (1{::}\mathrm{Ty}, nil)$$

where we would more naturally expect the type $Bool{::}\mathrm{Ty}, nil$. The information that 1 is $Bool$ is not found in the environment: the substitution s has to be used to check that 1 is indeed $Bool$. It seems thus that the type of a substitution cannot be intrinsically defined.

With these explanations in mind, the reader should be able to approach the rules of the theory S2 (though some may find it preferable to understand S2alg at the same time).

Definition 5.2 (Theory S2) *See appendix 7.*

We now prove the soundness of S2 with respect to L2.

Proposition 5.3 (Soundness)

1. *If* $E \ \vdash_{S2} \ a \sim b : A$
   then $\sigma(E) \ \vdash_{L2} \ \sigma(a) : \sigma(A)$ *and* $\sigma(a) = \sigma(b)$.

2. *If* $E \vdash_{S2} A \sim B :: \mathrm{Ty}$
   then $\sigma(E) \vdash_{L2} \sigma(A) :: \mathrm{Ty}$ *and* $\sigma(A) = \sigma(B)$.

3. *If* $\vdash_{S2} E \sim E'$ env
   then $\vdash_{L2} \sigma(E)$ env *and* $\sigma(E) = \sigma(E')$.

4. *If* $E \vdash_{S2} s \sim s'$ $subst_p$
   then there exist m and n such that

   - $\sigma(s) = G_1 \cdot \ \ldots \ \cdot G_m \cdot \uparrow^n$ *and* $\sigma(s') = G'_1 \cdot \ \ldots \ \cdot G'_m \cdot \uparrow^n$,
   - *for all* $q \leq m$, *either* $G_q{=}G'_q{=}A{::}\mathrm{Ty}$ *and* $\sigma(E) \vdash_{L2} A :: \mathrm{Ty}$ *for some A, or* $G_q = a{:}A$, $G'_q = a{:}A'$, $\sigma(A[\uparrow^q \circ s]) = \sigma(A'[\uparrow^q \circ s'])$, *and* $\sigma(E) \vdash_{L2} a : \sigma(A[\uparrow^q \circ s])$ *for some a, A, and A'*,
   - $p = m + \mid E \mid - n$.

Proof The proof is by induction on the rules of S2. We omit the checking of the numeric invariant in the last part of the claim. The cases for the EqReenving rules are trivial. The symmetric character of the claim settles the cases for the Symm and Trans rules, as well as that for EqRetyping. Other easy cases are those for rules that express typing through rewriting, and where one of the sides of the underlying rewrite rule appears in the premise. This concerns EqTyClosVarId, EqTyClosPi, EqTyClosClos, EqClosVarId, EqClosApp, EqClosAbs, EqClosClos, EqCompId, EqCompShiftId, EqCompShiftCons, EqCompCons, EqCompComp, and their variants (such as EqClosApp2). Now we briefly examine the remaining cases:

EqTyVar: by the induction hypothesis and L2-tvar.

EqTyPi: by the induction hypothesis, L2-tfun, and the observation that $\sigma(A \rightarrow B) = \sigma(A) \rightarrow \sigma(B)$.

EqTyPi2, EqTyClosVarShift, EqVar, EqAbs, EqApp, EqClosVarShift, EqNil, EqExt, and their variants (such as EqTyClosVarShiftN2): similar to EqTyVar and EqTyPi.

EqTyClosVarCons: by the induction hypothesis (with $q = 1$).

EqTyClosVarCong: we exploit the induction hypothesis on the first premise. There are two cases. If $m = 0$, then s and s' coincide, and the conclusion is identical to the second premise. If $m > 1$ and

$\sigma(s) = G{\cdot}s_1$, then G cannot have the form $a{:}A$, because we would get a contradiction from the induction hypothesis (on the second premise). Hence, $G = A :: \mathrm{Ty}$, and the conclusion follows from the induction hypothesis on the first premise (with $q = 1$).

EqClosVarCons: similar to EqTyClosVarCons, noting that $\sigma(A[s]) = \sigma(A[\uparrow \circ (a{:}A \cdot s)])$.

EqClosVarCong: similar to EqTyClosVarCong, except that the second premise forces G to have now the other form $a{:}A$.

EqId, EqShift, EqShift2: since in these cases s and s' coincide and $m = 0$, the property holds vacuously for the conclusion.

EqCons, EqCons2: by the induction hypothesis, noting that $\sigma(a{:}A \cdot s) = \sigma(a){:}\sigma(A) \cdot \sigma(s)$.

EqCompShiftCong: we exploit the induction hypothesis on the premise. If $m = 0$, then s and s' coincide, and we can use the argument of case EqId. If $m > 0$, the conclusion follows immediately from the assumption, since $\sigma(\uparrow \circ s) = \sigma(s_1)$, where $\sigma(s) = G \cdot s_1$ for some G.

□

As for S1, we speculate that the soundness claim for S2 can be strengthened, and that a converse completeness result then holds.

We now provide a typechecking algorithm S2alg for the second-order calculus. The algorithm is formulated as a set of inference rules, for easy comparison with S2. As we will see, each rule of S2alg is an admissible rule for S2; this shows the soundness of S2alg.

For terms that are not closures, S2alg and L2 operate identically. However, these are the least interesting cases: an actual implementation would manipulate only closures (as in subsection 3.5). In order to typecheck a term $a[s]$, the basic strategy is to analyze simpler and simpler components of a while accumulating more and more complex substitutions in s. When we finally reach an index, we extract the relevant information from the substitution or from the environment.

Informally, the algorithmic flow of control for each rule is: start with the given parts of the conclusion, recursively do what the assumptions on top require, accumulate the results, and from them produce the unknown parts of the conclusion. For example, if we want to type a in the environment E, we select an inference rule of S2alg by inspecting the shape of its conclusion. Then we move on to the assumptions of this rule, recursively; we solve the typing problems presented by each of them, and collect the results to produce a type for the original term a.

Some of the rules involve tests for type equivalence; two auxiliary "reduction" judgments are used for this:

$$E \vdash s \rightsquigarrow s' \quad subst_p \quad \text{and} \quad E \vdash A \rightsquigarrow A'::\text{Ty}$$

In these judgments, s' and A' are in a sort of weak head normal form, namely: s' is never a composition and if A' is a closure then it has the form $1[\uparrow^n]$.

Definition 5.4 (Algorithm S2alg) *See appendix 8.*

To show that S2alg really defines an algorithm, we first notice that only one rule can be applied bottom-up in each situation. For the judgments $E \vdash A::\text{Ty}$ and $E \vdash A \rightsquigarrow A'::\text{Ty}$, we test applicability by cases on A; when $A = B[s]$, by cases on B; and when $B = 1$ by cases on the reduction of s. For $E \vdash a : A$, we proceed by cases on a; when $a = b[s]$, by cases on b; and when $b = 1$ by cases on the reduction of s. For $E \vdash s \quad subst_p$, we proceed by cases on s, and when $s = t \circ u$ by cases on t. For $E \vdash s \rightsquigarrow s' \quad subst_p$, we proceed by cases on s; when $s = t \circ u$, by cases on t; and when $t = \uparrow$ by cases on the reduction of u. Finally, $E \vdash A \leftrightarrow B :: \text{Ty}$ is handled by cases on the reductions of A and B.

The following invariants can be used to show that the algorithm considers all the cases that may arise when the input terms are well-typed:

If $E \vdash s \rightsquigarrow s' \quad subst_p$ then s' is one of

id	
$\uparrow^n$	$(n \geq 1)$
$a{:}A \cdot t$	(for some a, A, and t)
$A::\text{Ty} \cdot t$	(for some A and t)

If $E \vdash A \rightsquigarrow A' :: \text{Ty}$ then A' is one of

1	
$1[\uparrow^n]$	$(n \geq 1)$
$B \rightarrow C$	(for some B and C)
$\forall B$	(for some B)

Finally, the algorithm can be shown to always terminate, with success or failure, because every rule either reduces the size of terms or moves terms towards a normal form.

The algorithm S2alg is sound with respect to S2:

Proposition 5.5

1. *If* $E \vdash_{S2alg} A :: \mathrm{Ty}$ *then* $E \vdash_{S2} A \sim A :: \mathrm{Ty}$.
2. *If* $E \vdash_{S2alg} a : A$ *then* $E \vdash_{S2} a \sim a : A$.
3. *If* $E \vdash_{S2alg} s \;\; subst_p$ *then* $E \vdash_{S2} s \sim s \;\; subst_p$.
4. *If* $E \vdash_{S2alg} s \rightsquigarrow s' \;\; subst_p$ *then* $E \vdash_{S2} s \sim s' \;\; subst_p$.
5. *If* $E \vdash_{S2alg} A \rightsquigarrow A' :: \mathrm{Ty}$ *then* $E \vdash_{S2} A \sim A' :: \mathrm{Ty}$.
6. *If* $E \vdash_{S2alg} A \leftrightarrow A' :: \mathrm{Ty}$ *then* $E \vdash_{S2} A \sim A' :: \mathrm{Ty}$.
7. *If* $\vdash_{S2alg} E \;\; \mathrm{env}$ *then* $\vdash_{S2} E \sim E \;\; \mathrm{env}$.

Proof The proof is a simple case analysis, with an extensive use of the Symm and Trans rules. □

We conjecture that the algorithm is also complete, in the following sense:

Conjecture 5.6

1. *If* $E \vdash_{S2} A \sim A' :: \mathrm{Ty}$ *then* $E \vdash_{S2alg} A :: \mathrm{Ty}$.
2. *If* $E \vdash_{S2} a \sim b : A$
   then $E \vdash_{S2alg} a : A'$ *and* $E \vdash_{S2alg} A' \leftrightarrow A :: \mathrm{Ty}$ *for some* A'.
3. *If* $E \vdash_{S2} s \sim s' \;\; subst_p$ *then* $E \vdash_{S2alg} s \;\; subst_p$.
4. *If* $\vdash_{S2} E \sim E \;\; \mathrm{env}$ *then* $\vdash_{S2alg} E \;\; \mathrm{env}$.

Unfortunately, it seems unlikely that one could simply prove the conjecture by induction on proofs (for example, the presence of $A' \leftrightarrow A$ in the second part of the statement gives rise to complications).

6 Conclusion

The usual presentations of the λ-calculus discreetly play down the handling of substitutions. This helps in developing the meta-theory of the λ-calculus, at a suitable level of abstraction. We hope to have demonstrated the benefits of a more explicit treatment of substitutions, both for untyped systems and typed systems. The theory and the manipulation of explicit substitutions can be delicate, but useful for correct and efficient implementations.

Acknowledgements We have benefited from discussions with P. Crégut, T. Hardin, E. Muller, A. Ríos, and A. Suárez, and from C. Hibbard's editorial help.

7 Appendix: Theory S2

7.1 Type equivalence

(TypeSymm) $$\frac{E \vdash A \sim B :: \mathrm{Ty}}{E \vdash B \sim A :: \mathrm{Ty}}$$

(TypeTrans) $$\frac{E \vdash A \sim B :: \mathrm{Ty} \qquad E \vdash B \sim C :: \mathrm{Ty}}{E \vdash A \sim C :: \mathrm{Ty}}$$

(EqTyVar) $$\frac{\vdash E \quad env}{\mathrm{Ty}, E \vdash 1 \sim 1 :: \mathrm{Ty}}$$

(EqTyPi) $$\frac{E \vdash A \sim A' :: \mathrm{Ty} \qquad A, E \vdash B \sim B' :: \mathrm{Ty}}{E \vdash A \to B \sim A' \to B' :: \mathrm{Ty}}$$

(EqTyPi2) $$\frac{\mathrm{Ty}, E \vdash B \sim B' :: \mathrm{Ty}}{E \vdash \forall B \sim \forall B' :: \mathrm{Ty}}$$

(EqTyClosVarId) $$\frac{\vdash E \quad env}{E \vdash 1[id] \sim 1 :: \mathrm{Ty}}$$

(EqTyClosVarShift) $$\frac{E \vdash 1 :: \mathrm{Ty} \qquad E \vdash A :: \mathrm{Ty}}{A, E \vdash 1[\uparrow] \sim 1[\uparrow] :: \mathrm{Ty}}$$

(EqTyClosVarShift2) $$\frac{E \vdash 1 :: \mathrm{Ty}}{\mathrm{Ty}, E \vdash 1[\uparrow] \sim 1[\uparrow] :: \mathrm{Ty}}$$

(EqTyClosVarShiftN) $$\frac{E \vdash 1[\uparrow^n] :: \mathrm{Ty} \qquad E \vdash A :: \mathrm{Ty}}{A, E \vdash 1[\uparrow^{n+1}] \sim 1[\uparrow^{n+1}] :: \mathrm{Ty}}$$

(EqTyClosVarShiftN2) $$\frac{E \vdash 1[\uparrow^n] :: \mathrm{Ty}}{\mathrm{Ty}, E \vdash 1[\uparrow^{n+1}] \sim 1[\uparrow^{n+1}] :: \mathrm{Ty}}$$

(EqTyClosVarCons)
$$\frac{E \vdash A{::}\mathrm{Ty} \cdot s \quad \mathit{subst}_p}{E \vdash 1[A{::}\mathrm{Ty} \cdot s] \sim A :: \mathrm{Ty}}$$

(EqTyClosVarCong)
$$\frac{E \vdash s \sim s' \quad \mathit{subst}_p \qquad E \vdash 1[s'] :: \mathrm{Ty}}{E \vdash 1[s] \sim 1[s'] :: \mathrm{Ty}}$$

(EqTyClosPi)
$$\frac{E \vdash A[s] \rightarrow B[1{:}A.(s \circ \uparrow)] :: \mathrm{Ty}}{E \vdash (A \rightarrow B)[s] \sim A[s] \rightarrow B[1{:}A \cdot (s \circ \uparrow)] :: \mathrm{Ty}}$$

(EqTyClosPi2)
$$\frac{E \vdash \forall(B[1 :: \mathrm{Ty} \cdot (s \circ \uparrow)]) :: \mathrm{Ty}}{E \vdash (\forall B)[s] \sim \forall(B[1{::}\mathrm{Ty} \cdot (s \circ \uparrow)]) :: \mathrm{Ty}}$$

(EqTyClosClos)
$$\frac{E \vdash A[s \circ t] :: \mathrm{Ty}}{E \vdash A[s][t] \sim A[s \circ t] :: \mathrm{Ty}}$$

(EqTypeReenving)
$$\frac{E \vdash A \sim B :: \mathrm{Ty} \qquad \vdash E \sim E' \quad \mathit{env}}{E' \vdash A \sim B :: \mathrm{Ty}}$$

7.2 Term equivalence

(TermSymm)
$$\frac{E \vdash a \sim b : A}{E \vdash b \sim a : A}$$

(TermTrans)
$$\frac{E \vdash a \sim b : A \qquad E \vdash b \sim c : A}{E \vdash a \sim c : A}$$

(EqVar)
$$\frac{E \vdash A :: \mathrm{Ty}}{A, E \vdash 1 \sim 1 : A[\uparrow]}$$

(EqAbs)
$$\frac{E \vdash A \sim A' :: \mathrm{Ty} \qquad A, E \vdash b \sim b' : B}{E \vdash \lambda A.b \sim \lambda A'.b' : A \rightarrow B}$$

(EqAbs2)
$$\frac{\mathrm{Ty}, E \vdash b \sim b' : B}{E \vdash \Lambda b \sim \Lambda b' : \forall B}$$

(EqApp) $$\frac{E \vdash b \sim b' : A \to B \qquad E \vdash a \sim a' : A}{E \vdash b(a) \sim b'(a') : B[a{:}A \cdot id]}$$

(EqApp2) $$\frac{E \vdash b \sim b' : \forall B \qquad E \vdash A \sim A' :: \mathrm{Ty}}{E \vdash b(A) \sim b'(A') : B[A{::}\mathrm{Ty} \cdot id]}$$

(EqClosVarId) $$\frac{E \vdash 1 : A}{E \vdash 1[id] \sim 1 : A}$$

(EqClosVarShift) $$\frac{E \vdash 1 : A \qquad E \vdash B :: \mathrm{Ty}}{B, E \vdash 1[\uparrow] \sim 1[\uparrow] : A[\uparrow]}$$

(EqClosVarShift2) $$\frac{E \vdash 1 : A}{\mathrm{Ty}, E \vdash 1[\uparrow] \sim 1[\uparrow] : A[\uparrow]}$$

(EqClosVarShiftN) $$\frac{E \vdash 1[\uparrow^n] : A \qquad E \vdash B :: \mathrm{Ty}}{B, E \vdash 1[\uparrow^{n+1}] \sim 1[\uparrow^{n+1}] : A[\uparrow]}$$

(EqClosVarShiftN2) $$\frac{E \vdash 1[\uparrow^n] : A}{\mathrm{Ty}, E \vdash 1[\uparrow^{n+1}] \sim 1[\uparrow^{n+1}] : A[\uparrow]}$$

(EqClosVarCons) $$\frac{E \vdash a{:}A \cdot s \quad \mathit{subst}_p}{E \vdash 1[a{:}A \cdot s] \sim a : A[s]}$$

(EqClosVarCong) $$\frac{E \vdash s \sim s' \quad \mathit{subst}_p \qquad E \vdash 1[s'] : A}{E \vdash 1[s] \sim 1[s'] : A}$$

(EqClosAbs) $$\frac{E \vdash \lambda A[s].b[1{:}A \cdot (s \circ \uparrow)] : B}{E \vdash (\lambda A.b)[s] \sim \lambda A[s].b[1{:}A \cdot (s \circ \uparrow)] : B}$$

(EqClosAbs2) $$\frac{E \vdash \Lambda(b[1{::}\mathrm{Ty} \cdot (s \circ \uparrow)) : B}{E \vdash (\Lambda b)[s] \sim \Lambda(b[1{::}\mathrm{Ty} \cdot (s \circ \uparrow)]) : B}$$

(EqClosApp) $$\frac{E \vdash (b[s])(a[s]) : A}{E \vdash b(a)[s] \sim (b[s])(a[s]) : A}$$

(EqClosApp2) $$\frac{E \vdash (b[s])(A[s]) : B}{E \vdash b(A)[s] \sim (b[s])(A[s]) : B}$$

(EqClosClos) $$\frac{E \vdash a[s \circ t] : A}{E \vdash a[s][t] \sim a[s \circ t] : A}$$

(EqRetyping) $$\frac{E \vdash a \sim b : A \qquad E \vdash A \sim B :: \mathrm{Ty}}{E \vdash a \sim b : B}$$

(EqTermReenving) $$\frac{E \vdash a \sim b : A \qquad \vdash E \sim E' \;\; env}{E' \vdash a \sim b : A}$$

As in S1, we do *not* include Beta rules in S2:

(Beta) $$\frac{E \vdash a : A \qquad A, E \vdash b : B}{E \vdash (\lambda A.b)(a) \sim b[a{:}A \cdot id] : B[a{:}A \cdot id]}$$

(Beta2) $$\frac{E \vdash A :: \mathrm{Ty} \qquad \mathrm{Ty}, E \vdash b : B}{E \vdash (\Lambda b)(A) \sim b[A{::}\mathrm{Ty} \cdot id] : B[A{::}\mathrm{Ty} \cdot id]}$$

7.3 Substitution equivalence

(SubsSymm) $$\frac{E \vdash s \sim t \;\; subst_p}{E \vdash t \sim s \;\; subst_p}$$

(SubsTrans) $$\frac{E \vdash s \sim t \;\; subst_p \qquad E \vdash t \sim u \;\; subst_p}{E \vdash s \sim u \;\; subst_p}$$

(EqId) $$\frac{\vdash E \;\; env}{E \vdash id \sim id \;\; subst_{|E|}}$$

(EqShift) $$\frac{E \vdash A :: \mathrm{Ty}}{A, E \vdash \uparrow \sim \uparrow \;\; subst_{|E|}}$$

(EqShift2) $$\frac{\vdash E \;\; env}{\mathrm{Ty}, E \vdash \uparrow \sim \uparrow \;\; subst_{|E|}}$$

(EqCons) $$\frac{E \vdash s \sim t \quad subst_p \qquad E \vdash A[s] \sim B[t] :: \mathrm{Ty} \qquad E \vdash a \sim b{:}A[s]}{E \vdash a{:}A \cdot s \sim b{:}B \cdot t \quad subst_{p+1}}$$

(EqCons2) $$\frac{E \vdash A \sim B :: \mathrm{Ty} \qquad E \vdash s \sim t \quad subst_p}{E \vdash A{::}\mathrm{Ty} \cdot s \sim B{::}\mathrm{Ty} \cdot t \quad subst_{p+1}}$$

(EqCompId) $$\frac{E \vdash s \sim s' \quad subst_p}{E \vdash id \circ s \sim s' \quad subst_p}$$

(EqCompShiftId) $$\frac{E \vdash \uparrow \quad subst_p}{E \vdash \uparrow \circ\, id \sim \uparrow \quad subst_p}$$

(EqCompShiftCons) $$\frac{E \vdash s \sim s' \quad subst_p \qquad E \vdash a : A[s]}{E \vdash \uparrow \circ (a{:}A \cdot s) \sim s' \quad subst_p}$$

(EqCompShiftCons2) $$\frac{E \vdash s \sim s' \quad subst_p \qquad E \vdash A :: \mathrm{Ty}}{E \vdash \uparrow \circ (A{::}\mathrm{Ty} \cdot s) \sim s' \quad subst_p}$$

(EqCompShiftCong) $$\frac{E \vdash s \sim s' \quad subst_{p+1}}{E \vdash \uparrow \circ s \sim \uparrow \circ s' \quad subst_p}$$

(EqCompCons) $$\frac{E \vdash a[t]{:}A \cdot (s \circ t) \quad subst_p}{E \vdash (a{:}A \cdot s) \circ t \sim a[t]{:}A \cdot (s \circ t) \quad subst_p}$$

(EqCompCons2) $$\frac{E \vdash A[t]{::}\mathrm{Ty} \cdot (s \circ t) \quad subst_p}{E \vdash (A{::}\mathrm{Ty} \cdot s) \circ t \sim A[t]{::}\mathrm{Ty} \cdot (s \circ t) \quad subst_p}$$

(EqCompComp) $$\frac{E \vdash s \circ (t \circ u) \quad subst_p}{E \vdash (s \circ t) \circ u \sim s \circ (t \circ u) \quad subst_p}$$

(EqSubstReenving) $$\frac{E \vdash s \sim t \quad subst_p \qquad \vdash E \sim E' \quad env}{E' \vdash s \sim t \quad subst_p}$$

7.4 Environment equivalence

(EnvSymm) $$\frac{\vdash\ E \sim E'\quad \mathit{env}}{\vdash\ E' \sim E\quad \mathit{env}}$$

(EnvTrans) $$\frac{\vdash\ E \sim E'\quad \mathit{env} \qquad \vdash\ E' \sim E''\quad \mathit{env}}{\vdash\ E \sim E''\quad \mathit{env}}$$

(Eqnil) $$\vdash\ \mathit{nil} \sim \mathit{nil}\quad \mathit{env}$$

(EqExt) $$\frac{\vdash\ E \sim E'\quad \mathit{env} \qquad E\ \vdash\ A \sim B :: \mathrm{Ty}}{\vdash\ A, E \sim B, E'\quad \mathit{env}}$$

(EqExt2) $$\frac{\vdash\ E \sim E'\quad \mathit{env}}{\vdash\ \mathrm{Ty}, E \sim \mathrm{Ty}, E'\quad \mathit{env}}$$

8 Appendix: Algorithm S2alg

8.1 Inference for types

(TyVar) $$\frac{\vdash E \quad env}{\mathrm{Ty}, E \vdash 1 :: \mathrm{Ty}}$$

(TyPi) $$\frac{E \vdash A :: \mathrm{Ty} \qquad A, E \vdash B :: \mathrm{Ty}}{E \vdash A \to B :: \mathrm{Ty}}$$

(TyPi2) $$\frac{\mathrm{Ty}, E \vdash B :: \mathrm{Ty}}{E \vdash \forall B :: \mathrm{Ty}}$$

(TyClosVarId) $$\frac{\mathrm{Ty}, E \vdash s \rightsquigarrow id \quad subst_p}{\mathrm{Ty}, E \vdash 1[s] :: \mathrm{Ty}}$$

(TyClosVarShift) $$\frac{E \vdash 1 :: \mathrm{Ty} \qquad E \vdash A :: \mathrm{Ty}}{A, E \vdash 1[\uparrow] :: \mathrm{Ty}}$$

(TyClosVarShift2) $$\frac{E \vdash 1 :: \mathrm{Ty}}{\mathrm{Ty}, E \vdash 1[\uparrow] :: \mathrm{Ty}}$$

(TyClosVarShiftN) $$\frac{E \vdash 1[\uparrow^n] :: \mathrm{Ty} \qquad E \vdash A :: \mathrm{Ty}}{A, E \vdash 1[\uparrow^{n+1}] :: \mathrm{Ty}}$$

(TyClosVarShiftN2) $$\frac{E \vdash 1[\uparrow^n] :: \mathrm{Ty}}{\mathrm{Ty}, E \vdash 1[\uparrow^{n+1}] :: \mathrm{Ty}}$$

(TyClosVarCons) $$\frac{E \vdash s \rightsquigarrow A{::}\mathrm{Ty} \cdot t \quad subst_p}{E \vdash 1[s] :: \mathrm{Ty}}$$

(TyClosVarCong) $$\frac{E \vdash s \rightsquigarrow \uparrow^n \quad subst_p \qquad E \vdash 1[\uparrow^n] :: \mathrm{Ty}}{E \vdash 1[s] :: \mathrm{Ty}}$$

(TyClosPi) $$\frac{E \vdash A[s] :: \mathrm{Ty} \qquad A[s], E \vdash B[1 : A \cdot (s \circ \uparrow)] :: \mathrm{Ty}}{E \vdash (A \to B)[s] :: \mathrm{Ty}}$$

(TyClosPi2) $$\frac{\mathrm{Ty}, E \vdash B[1{::}\mathrm{Ty} \cdot (s \circ \uparrow)] :: \mathrm{Ty}}{E \vdash (\forall B)[s] :: \mathrm{Ty}}$$

(TyClosClos) $$\frac{E \vdash A[s \circ t] :: \mathrm{Ty}}{E \vdash A[s][t] :: \mathrm{Ty}}$$

8.2 Inference for terms

(Var) $$\frac{E \vdash A :: \mathrm{Ty}}{A, E \vdash 1 : A[\uparrow]}$$

(Abs) $$\frac{E \vdash A :: \mathrm{Ty} \qquad A, E \vdash b : B}{E \vdash \lambda A.b : A \to B}$$

(Abs2) $$\frac{\mathrm{Ty}, E \vdash b : B}{E \vdash \Lambda b : \forall B}$$

(App) $$\frac{E \vdash b : A \to B \qquad E \vdash a : A}{E \vdash b(a) : B[a{:}A \cdot id]}$$

(App2) $$\frac{E \vdash b : \forall B \qquad E \vdash A :: \mathrm{Ty}}{E \vdash b(A) : B[A{::}\mathrm{Ty} \cdot id]}$$

(ClosVarId) $$\frac{A, E \vdash s \rightsquigarrow id \quad subst_p}{A, E \vdash 1[s] : A[\uparrow]}$$

(ClosVarShift) $$\frac{E \vdash 1 : A \qquad E \vdash B :: \mathrm{Ty}}{B, E \vdash 1[\uparrow] : A[\uparrow]}$$

(ClosVarShift2) $$\frac{E \vdash 1 : A}{\mathrm{Ty}, E \vdash 1[\uparrow] : A[\uparrow]}$$

(ClosVarShiftN)
$$\frac{E \vdash 1[\uparrow^n] : A \qquad E \vdash B :: \mathrm{Ty}}{B, E \vdash 1[\uparrow^{n+1}] : A[\uparrow]}$$

(ClosVarShiftN2)
$$\frac{E \vdash 1[\uparrow^n] : A}{\mathrm{Ty}, E \vdash 1[\uparrow^{n+1}] : A[\uparrow]}$$

(ClosVarCons)
$$\frac{E \vdash s \rightsquigarrow a{:}A \cdot t \quad \mathit{subst}_p}{E \vdash 1[s] : A[t]}$$

(ClosVarCong)
$$\frac{E \vdash s \rightsquigarrow \uparrow^n \quad \mathit{subst}_p \qquad E \vdash 1[\uparrow^n] : A}{E \vdash 1[s] : A}$$

(ClosAbs)
$$\frac{A[s], E \vdash b[1 : A \cdot (s \circ \uparrow)] : B}{E \vdash (\lambda A.b)[s] : A[s] \to B}$$

(ClosAbs2)
$$\frac{\mathrm{Ty}, E \vdash b[1{::}\mathrm{Ty} \cdot (s \circ \uparrow)] : B}{E \vdash (\Lambda b)[s] : \forall B}$$

(ClosApp)
$$\frac{\begin{array}{c} E \vdash b[s] : A \to B \qquad E \vdash a[s] : A' \\ E \vdash A \leftrightarrow A' :: \mathrm{Ty} \end{array}}{E \vdash (b(a))[s] : B[a[s] : A \cdot id]}$$

(ClosApp2)
$$\frac{E \vdash b[s] : \forall B \qquad E \vdash A[s] :: \mathrm{Ty}}{E \vdash (b(A))[s] : B[A[s]{::}\mathrm{Ty} \cdot id]}$$

(ClosClos)
$$\frac{E \vdash a[s \circ t] : A}{E \vdash a[s][t] : A}$$

8.3 Inference for substitutions

(Id)
$$\frac{\vdash E \quad \mathit{env}}{E \vdash id \quad \mathit{subst}_{|E|}}$$

(Shift)
$$\frac{E \vdash A :: \mathrm{Ty}}{A, E \vdash \uparrow \quad \mathit{subst}_{|E|}}$$

(Shift2) $$\frac{\vdash E \quad env}{\mathrm{Ty}, E \vdash \uparrow \quad subst_{|E|}}$$

(Cons) $$\frac{E \vdash a : B \qquad E \vdash s \quad subst_p \qquad E \vdash A[s] \leftrightarrow B :: \mathrm{Ty}}{E \vdash a : A \cdot s \quad subst_{p+1}}$$

(Cons2) $$\frac{E \vdash A :: \mathrm{Ty} \qquad E \vdash s \quad subst_p}{E \vdash A{::}\mathrm{Ty} \cdot s \quad subst_{p+1}}$$

(CompId) $$\frac{E \vdash s \quad subst_p}{E \vdash id \circ s \quad subst_p}$$

(CompShift) $$\frac{E \vdash s \quad subst_{p+1}}{E \vdash \uparrow \circ s \quad subst_p}$$

(CompCons) $$\frac{E \vdash a[t] : A \cdot (s \circ t) \quad subst_p}{E \vdash (a : A \cdot s) \circ t \quad subst_p}$$

(CompCons2) $$\frac{E \vdash A[t]{::}\mathrm{Ty} \cdot (s \circ t) \quad subst_p}{E \vdash (A{::}\mathrm{Ty} \cdot s) \circ t \quad subst_p}$$

(CompComp) $$\frac{E \vdash s \circ (t \circ u) \quad subst_p}{E \vdash (s \circ t) \circ u \quad subst_p}$$

8.4 Substitution reduction

(RedId) $$\frac{\vdash E \quad env}{E \vdash id \rightsquigarrow id \quad subst_{|E|}}$$

(RedShift) $$\frac{E \vdash A :: \mathrm{Ty}}{A, E \vdash \uparrow \rightsquigarrow \uparrow \quad subst_{|E|}}$$

(RedShift2) $$\frac{\vdash E \quad env}{\mathrm{Ty}, E \vdash \uparrow \rightsquigarrow \uparrow \quad subst_{|E|}}$$

(RedCons) $$\frac{E \vdash A[s] :: \mathrm{Ty} \qquad E \vdash a : B \qquad E \vdash B \leftrightarrow A[s] :: \mathrm{Ty} \qquad E \vdash s \ \ \mathit{subst}_p}{E \vdash a : A \cdot s \rightsquigarrow a : A \cdot s \ \ \mathit{subst}_{p+1}}$$

(RedCons2) $$\frac{E \vdash A :: \mathrm{Ty} \qquad E \vdash s \ \ \mathit{subst}_p}{E \vdash A{::}\mathrm{Ty} \cdot s \rightsquigarrow A{::}\mathrm{Ty} \cdot s \ \ \mathit{subst}_{p+1}}$$

(RedCompId) $$\frac{E \vdash s \rightsquigarrow s' \ \ \mathit{subst}_p}{E \vdash id \circ s \rightsquigarrow s' \ \ \mathit{subst}_p}$$

(RedCompShiftId) $$\frac{E \vdash s \rightsquigarrow id \ \ \mathit{subst}_{p+1}}{E \vdash \uparrow \circ s \rightsquigarrow \uparrow \ \ \mathit{subst}_p}$$

(RedCompShiftShiftN) $$\frac{E \vdash s \rightsquigarrow \uparrow^n \ \ \mathit{subst}_{p+1}}{E \vdash \uparrow \circ s \rightsquigarrow \uparrow^{n+1} \ \ \mathit{subst}_p}$$

(RedCompShiftCons) $$\frac{E \vdash s \rightsquigarrow a : A \cdot s' \ \ \mathit{subst}_{p+1} \qquad E \vdash s' \rightsquigarrow s'' \ \ \mathit{subst}_p}{E \vdash \uparrow \circ s \rightsquigarrow s'' \ \ \mathit{subst}_p}$$

(RedCompShiftCons2) $$\frac{E \vdash s \rightsquigarrow A{::}\mathrm{Ty} \cdot s' \ \ \mathit{subst}_{p+1} \qquad E \vdash s' \rightsquigarrow s'' \ \ \mathit{subst}_p}{E \vdash \uparrow \circ s \rightsquigarrow s'' \ \ \mathit{subst}_p}$$

(RedCompCons) $$\frac{E \vdash a[t] : A \cdot (s \circ t) \ \ \mathit{subst}_p}{E \vdash (a : A \cdot s) \circ t \rightsquigarrow a[t] : A \cdot (s \circ t) \ \ \mathit{subst}_p}$$

(RedCompCons2) $$\frac{E \vdash A[t]{::}\mathrm{Ty} \cdot (s \circ t) \ \ \mathit{subst}_p}{E \vdash (A{::}\mathrm{Ty} \cdot s) \circ t \rightsquigarrow A[t]{::}\mathrm{Ty} \cdot (s \circ t) \ \ \mathit{subst}_p}$$

(RedCompComp) $$\frac{E \vdash s \circ (t \circ u) \rightsquigarrow v \ \ \mathit{subst}_p}{E \vdash (s \circ t) \circ u \rightsquigarrow v \ \ \mathit{subst}_p}$$

8.5 Type reductions

(RedTyVar) $$\frac{\vdash E \quad \mathit{env}}{\mathrm{Ty}, E \vdash 1 \leadsto 1 :: \mathrm{Ty}}$$

(RedTyPi) $$\frac{E \vdash A :: \mathrm{Ty} \qquad A, E \vdash B :: \mathrm{Ty}}{E \vdash A \to B \leadsto A \to B :: \mathrm{Ty}}$$

(RedTyPi2) $$\frac{\mathrm{Ty}, E \vdash B :: \mathrm{Ty}}{E \vdash \forall B \leadsto \forall B :: \mathrm{Ty}}$$

(RedTyClosVarId) $$\frac{\mathrm{Ty}, E \vdash s \leadsto \mathit{id} \quad \mathit{subst}_p}{\mathrm{Ty}, E \vdash 1[s] \leadsto 1 :: \mathrm{Ty}}$$

(RedTyClosVarShiftN) $$\frac{E \vdash s \leadsto \uparrow^n \quad \mathit{subst}_p \qquad E \vdash 1[\uparrow^n] :: \mathrm{Ty}}{E \vdash 1[s] \leadsto 1[\uparrow^n] :: \mathrm{Ty}}$$

(RedTyClosVarCons) $$\frac{\begin{array}{c} E \vdash s \leadsto A{::}\mathrm{Ty} \cdot s' \quad \mathit{subst}_p \\ E \vdash A[s'] \leadsto B :: \mathrm{Ty} \end{array}}{E \vdash 1[s] \leadsto B :: \mathrm{Ty}}$$

(RedTyClosPi) $$\frac{\begin{array}{c} E \vdash A[s] :: \mathrm{Ty} \\ A[s], E \vdash B[1 : A \cdot (s \circ \uparrow)] :: \mathrm{Ty} \end{array}}{E \vdash (A \to B)[s] \leadsto A[s] \to B[1 : A \cdot (s \circ \uparrow)] :: \mathrm{Ty}}$$

(RedTyClosPi2) $$\frac{\mathrm{Ty}, E \vdash B[1{::}\mathrm{Ty} \cdot (s \circ \uparrow)] :: \mathrm{Ty}}{E \vdash (\forall B)[s] \leadsto \forall(B[1{::}\mathrm{Ty} \cdot (s \circ \uparrow)]) :: \mathrm{Ty}}$$

(RedTyClosClos) $$\frac{E \vdash A[s \circ t] \leadsto B :: \mathrm{Ty}}{E \vdash A[s][t] \leadsto B :: \mathrm{Ty}}$$

8.6 Type equivalence

(EqTyVar) $$\frac{E \vdash A \leadsto 1 :: \mathrm{Ty} \qquad E \vdash A' \leadsto 1 :: \mathrm{Ty}}{E \vdash A \leftrightarrow A' :: \mathrm{Ty}}$$

(EqTyPi)
$$\frac{\begin{array}{c} E \vdash A \leadsto B \to C :: \mathrm{Ty} \\ E \vdash A' \leadsto B' \to C' :: \mathrm{Ty} \\ E \vdash B \leftrightarrow B' :: \mathrm{Ty} \\ B, E \vdash C \leftrightarrow C' :: \mathrm{Ty} \end{array}}{E \vdash A \leftrightarrow A' :: \mathrm{Ty}}$$

(EqTyPi2)
$$\frac{\begin{array}{c} E \vdash A \leadsto \forall B :: \mathrm{Ty} \qquad E \vdash A' \leadsto \forall B' :: \mathrm{Ty} \\ \mathrm{Ty}, E \vdash B \leftrightarrow B' :: \mathrm{Ty} \end{array}}{E \vdash A \leftrightarrow A' :: \mathrm{Ty}}$$

(EqTyClos)
$$\frac{E \vdash A \leadsto 1[\uparrow^n] :: \mathrm{Ty} \qquad E \vdash A' \leadsto 1[\uparrow^n] :: \mathrm{Ty}}{E \vdash A \leftrightarrow A' :: \mathrm{Ty}}$$

8.7 Inference for environments

(Nil) $\vdash \mathit{nil} \quad \mathit{env}$

(Ext)
$$\frac{\vdash E \quad \mathit{env} \qquad E \vdash A :: \mathrm{Ty}}{\vdash A, E \quad \mathit{env}}$$

(Ext2)
$$\frac{\vdash E \quad \mathit{env}}{\vdash \mathrm{Ty}, E \quad \mathit{env}}$$

References

[1] H.P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, North Holland, 1985.

[2] N. De Bruijn, Lambda-calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, Indag. Mat. 34, pp. 381–392, 1972.

[3] L. Cardelli, Typeful Programming, SRC Report No. 45, Digital Equipment Corporation, 1989.

[4] H.P. Curry and R. Feys, *Combinatory Logic*, Vol. 1, North Holland, 1958.

[5] P.-L. Curien, The $\lambda\rho$-calculi: an Abstract Framework for Closures, unpublished (preliminary version printed as LIENS report, 1988).

[6] P.-L. Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming*, Pitman, 1986.

[7] J. Field, On Laziness and Optimality in Lambda Interpreters: Tools for Specification and Analysis, in the Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, pp. 1–15, San Francisco, January 1990.

[8] T. Hardin, Confluence Results for the Pure Strong Categorical Combinatory Logic CCL: λ-calculi as Subsystems of CCL, Theoretical Computer Science 65, pp. 291–342, 1989.

[9] T. Hardin, A. Laville, Proof of Termination of the Rewriting System SUBST on CCL, Theoretical Computer Science 46, pp. 305–312, 1986.

[10] T. Hardin, J.-J. Lévy, A Confluent Calculus of Substitutions, France-Japan Artificial Intelligence and Computer Science Symposium, Izu, December 1989.

[11] G. Huet, D.C. Oppen, Equations and Rewrite Rules: A Survey, in *Formal Languages Theory: Perspectives and Open Problems* (R. Book, editor), pp. 349–393, Academic Press, 1980.

[12] J.W. Klop, *Combinatory Reduction Systems*, Mathematical Center Tracts 129, Amsterdam, 1980.

[13] J.-L. Krivine, unpublished.

[14] P. Martin-Löf, *Intuitionistic Type Theory*, notes by G. Sambin of a series of lectures given in Padova in 1980, Bibliopolis, 1984.

[15] C.P. Wadsworth, *Semantics and Pragmatics of the Lambda Calculus*, Dissertation, Oxford University, 1971.

SRC Research Reports

SRC Reports

Martín Abadi, Luca Cardelli, Pierre-Louis Curien,
Jean-Jaques Lévy

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301