# Proceedings of the Fourth Workshop on Explicit Substitutions Theory and Applications (WESTAPP '01)

Pierre Lescanne, editor

Utrecht, May 20, 2001

Logic Group
Preprint Series
No. 210
May 2001

ZENO Institute of Philosophy

ISBN 90-393-2764-5
ISSN 0929-0710
Prof.dr. A. Visser, Editor

# Contents

## Preface

This report contains the proceedings of the fourth *Workshop on Explicit Substitutions Theory and Applications.* Prior events took place at Tsukuba (1998), Trento (1999) and Norwich (2000).

The aim of this workshop is to bring together researchers working on both theoretical and applied aspects of explicit substitutions, to present recent work (possibly still in progress), and to discuss new ideas and emerging trends.

This year two well-known contributors to lambda-calculus explicit substitutions, Corrado Böhm and Paul-André Melliès, were invited to give their view on the development of the field. In addition, four papers presented new developments.

The program committee consisted of

Roel Bloo (TUE, Eindhoven, The Nerherlands)
Therese Hardin (U. P et M. Curie, Paris, France)
Fairouz Kamareddine (HWU, Edinburgh, U.K.)
Pierre Lescanne (ENS, Lyon, France, *chair*)
César Muñoz (Nasa, Virginia, USA)
Frank Pfenning (CMU, Pennsylvania,USA)

The local arrangements chair was Vincent van Oostrom. Sponsors of the events included Centrum voor Wiskunde en Informatica, Instituut voor Programmatuurkunde en Algoritmiek, Department of Philosophy at Universiteit Utrecht, International Federation for Information Processing, Leiden-Utrecht Research Institute, University of Amsterdam Informatics Institute, and University of Tsukuba.

Pierre Lescanne
May 2001

# Separability, Self-Recognition and Unification

— Invited paper —

Corrado Böhm

# A Generic Object-Calculus Based on Addressed Term Rewriting Systems

Dan Dougherty
Wesleyan University
Middletown, CT 06459, USA

Frédéric Lang
INRIA Rhone-Alpes
Montbonnot, 38334 St Ismier, France

Pierre Lescanne
École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon 7, France

Luigi Liquori
École des Mines-INPL-LORIA
Parc de Saurupt, 54042 Nancy, France

Kristoffer Rose
IBM T. J. Watson Research Center
30 Saw Mill River Road, Hawthorne, NY 10532, USA

May 10, 2001

**Abstract**

We describe the foundations of $\lambda\mathcal{O}bj^{+a}$, a framework, or generic calculus, for modeling *object-calculi*. This framework is essentially a detailed formal operational semantics of object based languages, in the style of the Lambda Calculus of Objects. As a formalism for specification $\lambda\mathcal{O}bj^{+a}$ is arranged in *modules*, permitting a natural classification of many object-based calculi according to their features. In particular there are modules for calculi of non-mutable objects (*i.e.*, *functional object-calculi*) and for calculi of mutable objects (*i.e.*, *imperative object-calculi*). As a computational formalism $\lambda\mathcal{O}bj^{+a}$ is based on rewriting rules. Classical first-order term rewriting systems are not appropriate since we want to reflect aspects of implementation practice such as sharing, cycles in data structures and mutation. Therefore we define the notion of *addressed terms*, and develop the corresponding notion of *addressed term rewriting*.

## 1 Introduction

Recent years have seen a great deal of research aimed at providing a rigorous foundation for object-oriented programming languages. In many cases this work has taken the form of "object-calculi" [FHM94, AC96].

Such calculi can be understood in two ways. On one hand, the formal system is a description of the semantics of the language, and can be used as a framework for classifying language design choices, to provide a setting for investigating type systems, or to support a denotational semantics.

Alternatively, we may treat an object-calculus as an intermediate language into which user code (in a high-level object-oriented language) may be translated, and from which an implementation (in machine language) may be derived. Then the main problem is to describe how to correctly and efficiently execute terms in the object-calculus from which support for implementing several high-level object-calculi can be achieved.

In this paper we present the calculus $\lambda\mathcal{O}bj^{+a}$ in which one can give a formal specification of the operational semantics for a variety of object-based programming languages. In fact, $\lambda\mathcal{O}bj^{+a}$ (introduced in [LLL99]) is a *generic framework*, leading to an easy *classification* of object-based languages and their semantics, making a clear distinction between functional and imperative languages, *i.e.*, languages with non-mutable (persistent) objects and languages with mutable (ephemeral) objects.

The calculus $\lambda\mathcal{O}bj^{+a}$ is based on $\lambda$-calculus. Despite this fact, we stress that we do *not* restrict our attention to so-called "functional" object-oriented calculi. A key feature of our approach is the representation

of programs as *addressed terms* [LDLR99] which support reasoning about mutation. Since $\lambda\mathcal{O}bj^{+a}$ contains the $\lambda$-calculus explicitly, we therefore have a modular and uniform treatment of both functional and imperative programming.

Many treatments of functional operational semantics exist in the literature [Lan64, Aug84, Kah87, MTH90]. To go further and accommodate imperative operations one can use the traditional *stack and store* approach [Plo81, Tof90, FH92, WF94, AC96, BF98]. The greater complexity of the presentation of the latter works might lead one to conclude—wrongly—that implementing functional languages is easy in comparison with imperative languages. Such a false impression may be due in part to the fact that typical operational semantics formalisms are based on algebras: this makes them good at abstracting away the complexity of the algebraic structures used in functional languages but ill-suited to express the non-algebraic structure of imperative data structures. The novelty of $\lambda\mathcal{O}bj^{+a}$ is that it provides an *homogeneous* approach to both functional and imperative aspects of programming languages, in the sense the two semantics are treated in the same way using addressed terms, with only a minimal sacrifice in the permitted algebraic structures. Indeed, the addressed terms used were originally introduced to describe sharing behavior for functional programming languages [Ros96, BRL96].

From another point of view the use of addressed terms suggests a bridge between the operational and denotational semantics for a language. This is not a direction we pursue in detail in this paper but the main idea is as follows. A traditional denotational semantics for imperative languages involves the *store*, *i.e.*, a function from locations to values, as one of the key domains. The store typically has no explicit representation in the programming language and it has no structure beyond being a function space.

Now, if we identify locations with addresses in an addressed-term representation of an execution state, we may see the store as intimately bound up with the program expression. Indeed, an execution state expression embodies a function from addresses to sub-expressions whose values in turn comprise the store. Note that the semantics now need not refer to the entire store but only that finite part concerning addresses explicit in the execution state. Thus the store inherits a structure, induced by the execution state's tree-structure and there is now a notion of one store-location occurrence being within the scope of another.

All of this leads to a new and rather subtle relationship between operational and denotational semantics.

Our main concern is thus to find the right level of abstraction, more general and robust than the machine level, yet more concrete and operational than a purely mathematical treatment *à la* $\lambda$-calculus.

Specifically, the calculus $\lambda\mathcal{O}bj^{+a}$ enjoys the following properties:

- It is *faithful to implementation* in the sense that each transition in the system corresponds to a constant-cost operation in the execution of code on a machine. This permits reasoning about resource usage and the actual cost of certain implementation choices;
- It is *a formal system* which can support a careful analysis of some fundamental properties of object-oriented languages, such as type-safety and observational equivalence.

With regard to the first point, $\lambda\mathcal{O}bj^{+a}$ gives an explicit account of substitution, sharing, and redirection. The inclusion of explicit indirection nodes is a crucial innovation here. Indirection nodes allow us to give a more realistic treatment of the so-called collapsing rules of term graph rewriting (rules that rewrite a term to one of its proper sub-terms): more detailed discussion will be found in Sections 3.5 and 4.2.

The framework $\lambda\mathcal{O}bj^{+a}$ is much more than a simple object-calculus. It is defined in terms of a set of four *modules* (L, C, F, and I), each of which captures a particular aspect of object-calculi. Indeed, the modules are sets of rules which describe, in "small steps", the transformations of the objects, whereas the strategies (such as call-by value, call-by-name, etc) describe how these rules are invoked giving the general evolution of the whole program. Usually in the description of an operational semantics, strategies and small steps are tightly coupled. In our approach they are strongly independent. As a consequence, we get the genericity of $\lambda\mathcal{O}bj^{+a}$, in the sense that many semantics can be instantiated in our framework to conform to specific wishes. A specific calculus is therefore a combination of *modules plus a suitable strategy*. Thus we choose to not code strategies into the framework itself and we postpone discussion of specific strategies for future work.

A useful way to understand the current project is by analogy with graph-reduction as an implementation-calculus for functional programming. As such we may see $\lambda\mathcal{O}bj^{+a}$ as part of a fundamental correspondence:

| Funct. Programm. | Graph rewriting and explicit substitutions |
|---|---|
| O.O. Programm. | Addressed term rewriting and explicit substitutions ($\lambda\mathcal{O}bj^{+a}$) |

In the remainder of this introduction we make the analogy above more precise, describing the technical and historical context of $\lambda\mathcal{O}bj^{+a}$, before giving the outline of the rest of the paper.

## 1.1 Addressed Calculi and Explicit Substitutions for $\lambda$-calculi

It is well-understood that the $\lambda$-calculus [Chu41, Bar84] is of fundamental importance in understanding the semantics of both imperative and functional programming languages [Lan66, Sto77]. In this work, we are mainly interested in the role of $\lambda$-calculus in implementations, specifically in two areas: explicit substitutions and sharing.

**Explicit Substitution Calculi.** These were invented in order to give a finer description of the meta-operation of *substitution*, a fundamental notion in any programming language (see for instance [ACCL91, Les94, BR95]). Roughly speaking, an explicit substitution calculus fully includes the substitution operation as part of the syntax, adding suitable rewriting rules to deal with it. These calculi give a good model of the concept of "closures" that represent partially computed function applications. If combined with updating, closures can represent objects by considering the state of the object as what has been computed "so far" [ASS85].

**The semantics of sharing.** Efficient implementations of lazy functional languages (or of computer algebras) require some sharing mechanism to avoid multiple computations of a single argument. Term graphs [Wad71, Tur79, BVEG+87, Plu99] have been studied as a representation of program-expressions intermediate between abstract syntax trees and concrete representations in memory, and term-graph rewriting provides a formal operational semantics of functional programming sensitive to sharing.

However, compared with thinking with finite terms, representing and thinking with graphs can be awkward. Indeed, one is faced with non well-founded relations which prevent proofs by induction. (Observe that graphs differ from trees in that the latter naturally support definition and proof by structural induction).

In this paper we will annotate terms (trees) with *global addresses* [FF89, Ros96]. Levy [Lév80] and Maranget [Mar92] propose using *local addresses*, but from the point of view of the operational semantics, global addresses describe better what is going in a computer or an abstract machine. With explicit global addresses we can keep track of the sharing that could be used in the implementation of a particular $\lambda$-calculus of explicit substitution. Sub-terms which share a common address represent the same sub-graphs, as suggested in Figure 1 (left), where $a$ and $b$ denote addresses. A specific notion of rewriting is introduced in order to rewrite simultaneously all sub-terms sharing a same address, mimicking what would happen in an actual implementation. These ideas were also presented in [BRL96] in the context of a simple $\lambda$-calculus with explicit substitution. We enrich the sharing with a special *back-pointer* to handle *cyclic*

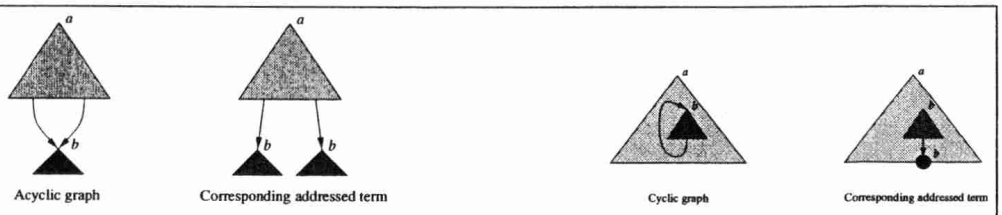



Figure 1: Representation of sharing and cycles through addresses (schematic)

*graphs* [Ros96]. Cycles are used in the functional language setting to represent infinite data-structures and (in some implementations) to represent recursive code; they are also interesting in the context of imperative object-oriented languages where *loops in the store* may be created by imperative updates through the use of `self` (or `this`), see Figure 6. The idea of the representation of cycles via addressed terms is rather natural:

a cyclic path in a finite graph is fully determined by a prefix path ended by a "jump" to some node of the prefix path (represented with a back-pointer), as suggested in Figure 1 (right).

In [LDLR99], addressed terms are studied in the context of *addressed term rewriting*, as an extension of classical first-order term rewriting. The notion of computation on terms is expanded to encompass computations performing mutation, still through rewriting rules.

It is natural to apply these techniques in the setting of object-oriented languages to the notion of destructive updates, or update of the value of a field (as for instance the increment of a counter encapsulated in an object).

### 1.2 Object-based Languages

Recent years have seen the development of *object-based* languages (see [AC96] Chapter 4). Object-based languages can be either viewed as a novel object-oriented style of programming (such as in Self [US87], Obliq [Car95], Kevo [Tai92], Cecil [Cha93], and O-{1,2,3} [AC96]). In object-based languages there is no notion of class: the inheritance takes place at the object level. Objects are built "from scratch" or by inheriting the methods and fields from other objects (sometimes called *prototypes*).

Two classical problems we find in the literature regarding the implementation of imperative and flexible object-calculi are:

- The capacity to handle *loops in the store* [AC96];
- The capacity to dynamically extend objects.

The latter feature is usually forbidden when one wants to have a sound type system (the imperative object-calculus of [BF98] uses a "functional" extension).

Among the proposals firmly setting the theoretical foundation of object-based languages, the *Lambda Calculus of Objects* ($\lambda\mathcal{O}bj$) of Fisher, Honsell, and Mitchell [FHM94] has formed one of the two major schools of calculi for modeling object-oriented programming (the other is the *Object Calculus* of Abadi and Cardelli [AC96]).

$\lambda\mathcal{O}bj$ is an untyped $\lambda$-calculus enriched with object primitives. Objects are *untyped* and a new object can be created by modifying and/or extending an existing prototype object. The result is a new object which inherits all the methods and fields of the prototype. The consistency of dynamic object-extension with a sound type-system was one of the main goals of $\lambda\mathcal{O}bj$. This calculus is computationally complete, since the $\lambda$-calculus is built in the calculus itself.

The calculus $\lambda\mathcal{O}bj^{+}$ [GHL98] is an extension of $\lambda\mathcal{O}bj$ with a type system and a type soundness result ensuring that a typed program "cannot go wrong". In particular, $\lambda\mathcal{O}bj^{+}$ allows typed objects to extend themselves upon the reception of a message.

### 1.3 Roadmap

This brings us to the $\lambda\mathcal{O}bj^{+a}$ framework, the subject of this paper. $\lambda\mathcal{O}bj^{+a}$ is based on $\lambda\mathcal{O}bj^{+}$, and uses addressed terms and explicit substitution. It is faithful to mainstream object-based programming languages in the sense that an object-calculus represents a core calculus to analyze these languages, and it extends traditional graph-reduction techniques by expressing the basic computational steps of object-oriented programming, including message-passing, method update, and especially object mutation.

In Section 2, we discuss by an example the main concepts that a generic calculus of objects has to take into account. In Section 3, we say how addressed term rewriting systems give solutions to the basic questions of object oriented languages, namely sharing, cycles and mutations. Section 4 presents the four modules of rewriting rules that form the core of $\lambda\mathcal{O}bj^{+a}$. Section 5 concludes and describes related and further works.

## 2 A Simple Example Exploiting Object Inheritance

The examples in this section embody certain choices about language design and implementation (such as deep *vs.* shallow copying, management of run-time storage, and so forth). It is important to stress that

these choices are not bound up with the particular formal calculus $\lambda\mathcal{O}bj^{+a}$ which is the subject of this paper. Indeed, our main point is that $\lambda\mathcal{O}bj^{+a}$ provides a foundation for a wide variety of language styles and language implementations. We hope that the examples are suggestive enough that it will be intuitively clear how to accommodate other design choices. The main body of the paper justifies that intuition.

Those schematic examples will be also useful to understand how objects are represented and how inheritance can be implemented in $\lambda\mathcal{O}bj^{+a}$. For the sake of simplicity, we will not raise issues like privacy or encapsulation (so that we consider methods and fields to belong to the same abstraction level).

Reflecting implementation practice, in $\lambda\mathcal{O}bj^{+a}$ we distinguish two distinct aspects of an object:

- **The object structure:** the actual list of methods/fields;
- **The object identity:** a pointer to the object structure.

We shall use the word "pointer" where others use "handle" or "reference". Objects can be bound to identifiers as "nicknames" (*e.g.*, `pixel`), but the only proper name of an object is its object identity: an object may have several nicknames (aliases) but only one identity.

Consider the following (untyped) definition of a "pixel" prototype with three fields and one method. With a slight abuse of notation, we use `:=` for both assignment of an expression to a variable, or the extension of an object with a new field or method and for overriding an existing field or method inside an object with a new value or body, respectively.

```
pixel = object {
  x := 0;
  y := 0;
  onoff := true;
  set := (a,b,c){ x := a; y := b; onoff := c; };
}
```

After instantiation, the object `pixel` is located at an address, say $a$, and its object structure starts at address $b$. See Figure 2. In what follows, we will derive three other objects from `pixel` and discuss the variations

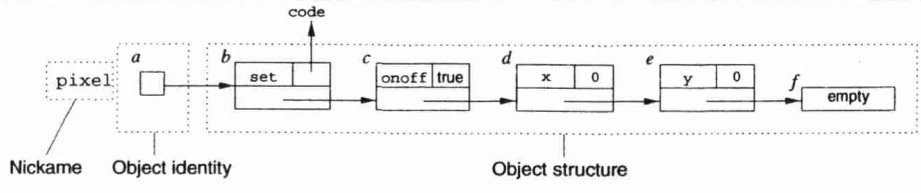



Figure 2: The Object Pixel

of how this may be done below.

## 2.1 Cloning

The first two derived objects, nick-named `p` and `q`, are *clones* of `pixel`:

```
p := clone(pixel);
q := clone(p);
```

Object `p` shares the same object-structure as `pixel` but of course is has its own object-identity. Object `q` shares also the same object-structure of `pixel`, even if it is a clone of `p`. The effect is pictured in Figure 3. The semantics of the `clone` operator we illustrate here differs somewhat from that found in certain existing object-oriented programming languages like SmallTalk and Java. For example in Java there will be sharing between an object and its clone if an instance field of the original is itself a reference to an object. In Java a true "deep" clone of an object is in general not available via the built-in clone method, but may be provided by the programmer overriding the default method.

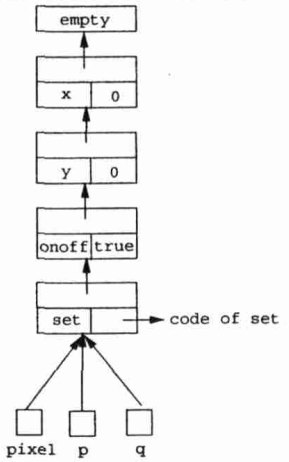



Figure 3: The Clones p and q

The clone operator (used by $\lambda\mathcal{O}bj^{+a}$) always produces a "shallow" copy of the prototype. As will be shown in the next subsection, however, one should not view this as simple aliasing.

In the rest of this section, we discuss the differences between functional and imperative models of object-calculi.

## 2.2 Illustrating an Imperative Calculus

Imperative object-calculi have been shown to be fundamental in describing implementations of class-based languages. They are also essential as foundations of object-based programming languages like Obliq and Self. The main goal when one tries to define the semantics of an imperative object-based language is to say how an object can be modified while maintaining its object-identity. Particular attention must be paid to this when dealing with object extension. The semantics of the imperative update operation is subtle because of side-effects.

Here we show what we want to model in our framework when we *override* the set method of the clone q of pixel, and we extend a clone r of (the modified) q with a new method switch.

```
q.set := (a,b,c){ x := x*a; y := y*b; onoff := c;}; (1)
r := clone(q);                                      (2)
r.switch := (){ onoff := not(onoff);};              (3)
```

Note that we have used a Java-like imperative syntax here to save parentheses.

Figure 4 shows the state of the memory after the execution of the instructions (1,2). Note that after (1) the object q refers to a new object-structure, obtained by chaining the new body for set with the old object-structure. As such when the overridden set method is invoked, thanks to dynamic binding, the newer body will be executed since it will hide the older one.

Observe that the override of the set method does not produce any side-effect on p and pixel; in fact, the code for set used by pixel and p will be just as before. Therefore, (1) only changes the object-structure of q without changing its object-identity. This is the sense in which our clone operator really does implement shallow copying rather than aliasing, even though there is no duplication of object-structure at the time that clone is evaluated.

This implementations model, although space consuming, performs side effects in a very restricted and controlled way. Figure 5, finally, shows the final state of memory after the execution of the instruction (3). Observe that in this case the update operation, denoted by ":=", extend the object r with the onoff method. Again, the addition of the switch method change only the object-structure of r.

In general, changing the nature of an object dynamically by adding a method or a field can be implemented by moving the object identity toward the new method/field (represented by a piece of code or

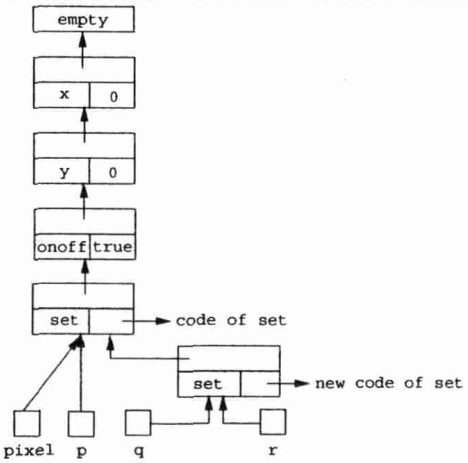



Figure 4: The Memory Structure After (1,2)

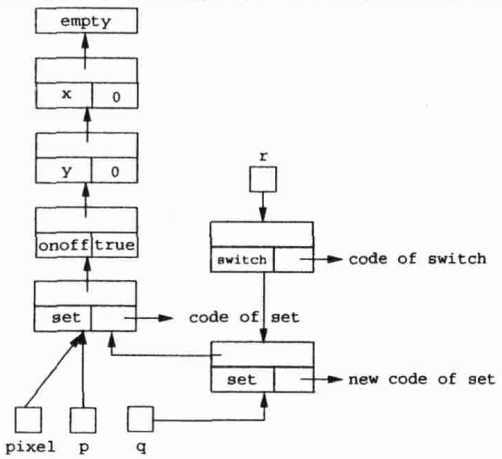



Figure 5: The Memory Structure after (3)

a memory location) and to *chain it* to the original structure. This mechanism is used systematically also for method/field overriding but in practice (for optimization purposes) can be relaxed for field overriding, where a more efficient *field look up and replacement* technique can be adopted. See for example the case of the Imperative Object Calculus ([AC96], Chapter 10), or observe that Java uses *static field lookup* to make the position of each field constant in the object.

This implementation model, however does not avoid the unfortunate *loop in the store*: an example of loop will be given Section 3.

## 2.3 Illustrating a Functional Calculus

Object-calculi can play a role as well in a purely functional setting, where there is no notion of *mutable state*. As said before, an "update" operation, denoted by ":=", can either override or extend an object with some fields or methods. In a functional setting, the update *always* produces another object with its proper object-identity since this ensures that all references to an object have the same meaning whether their evaluation is delayed or not. This property is also known as *referential transparency*. Thus, the result of an update must be a fresh object in the sense that it has a proper (new) object-identity.

Probably, the purists of functional languages would not like the "imperative" notation of the *pixel,p,q,r*

example; note that the definition of `pixel,p,q,r`, for example, could have been written like the following in a more traditional functional object-calculus where `let x = A in B` is syntactic sugar for the functional application `(λx.B) A` and the `clone(_)` function is essentially the identity (since we have no aliasing and application always produces new objects):

```
let p = clone(pixel) in p in
  let q = clone(p).set := (a,b,c)
                {((self.x := self.x*a).y := self.y*b).onoff := c } in
    let r = (q.switch := (){ self.onoff := self.not(onoff);}) in r
```

which obviously reduces to:

```
(clone(pixel).set := (a,b,c) {...}).switch := (){...}
```

Worth noticing is that the above code would be implemented, in a purely functional calculus, in the same way as Figure 5. This is sound since the way we treat imperative features of object-calculi is quite restricted and controlled, hence similar to the functional treatment of objects.

# 3 Addressed Term Rewriting Systems

The paradigm of *term rewriting* [DJ90, Klo90, BN98] is a very convenient and powerful tool to describe the operational semantics of simple calculi. In particular, term rewriting provides a computational interpretation of first-order equational reasoning.

In addition, term rewriting systems are sufficiently flexible to model the operational semantics of functional programs, although at a high level, ignoring certain aspects of memory management, reduction strategy, and parameter-passing. They are widely used to formalize, prototype, and verify software.

However, as suggested in the introduction, classical term rewriting cannot easily express issues of sharing (including cyclic data), mutation, and reduction strategies. Calculi which give an account of memory management usually introduce some *ad-hoc* data-structure to model the memory, called *heap*, or *store*, together with access and update operations. However, the use of these structures necessitates restricting the calculus to a particular strategy. The aim of addressed term rewriting systems, as the computational foundation of $\lambda\mathcal{O}bj^{+a}$, is to abstract out the notion of store so that we recover the flexibility of term rewriting, and permit description of computations in a store, by the way of *addressed rewriting rules* and the subsequent notion of rewriting.

In this section we introduce *addressed term rewriting systems* (ATRS's) informally—in the context of object-oriented programming—by examining these issues in turn and the ways in which they are reflected in features of ATRS's.

## 3.1 Sharing

Sharing has been extensively studied in the context of obtaining implementations of lazy functional programming languages [PJ87, PvE93], and the initial studies of sharing in the notations of *term graph rewriting systems* [BVEG+87, Plu99], were indeed motivated by this application.

**Sharing of computation.** Consider the function square defined by

$$\mathsf{square}(x) = \mathsf{times}(x, x)$$

It is clear that an implementation of this function should not duplicate its input $x$ in the expression $\mathsf{times}(x, x)$, but optimize this by only copying a *pointer* to the input. This not only saves memory but also makes it possible to *share future computations* on $x$, in particular when $x$ is not already required to be a value, as in *e.g.*, lazy programming languages. Classical term representations do not permit us to express this sharing of the actual structure of $x$. However, the memory structures used for the computation of a program can be represented using addressed terms. For instance, the "program" square(square(2)) can be

first instantiated in memory, provided locations $a, b, c$ to each of its constructors, as the addressed term (or memory structure) $\mathsf{square}^a(\mathsf{square}^b(2^c))$. It can then be reduced as follows:

$$\begin{aligned}\mathsf{square}^a(\mathsf{square}^b(2^c)) &\to \mathsf{times}^a(\mathsf{square}^b(2^c), \mathsf{square}^b(2^c)) \\ &\to \mathsf{times}^a(\mathsf{times}^b(2^c, 2^c), \mathsf{times}^b(2^c, 2^c)) \\ &\to \mathsf{times}^a(4^b, 4^b) \\ &\to 16^a,\end{aligned}$$

where "$\to$" designates one step of shared computation (we are assuming that definitions to compute the function $\mathsf{times}(x, y)$ to the value $x \times y$ exist for each $x$ and $y$). The key point of a shared computation is that *all* terms which share a common address are reduced *simultaneously*. This corresponds to a *single computation step* on a small component of the memory.

**Sharing of memory structures.** In object-oriented programming, the aim of sharing is not only to share *computations* as in the former example, but also to share *structures*. Indeed, objects are typically structures which receive multiple pointers. Moreover, the delegation-based model of inheritance insists that object structures are shared between objects with different identities. Representing object structures with the constructors $\langle\,\rangle$ (the empty object), and $\langle \_ \leftarrow \_ \rangle$ (the functional *cons* of an object with a method/field), and object identities by the bracketing symbol $[\![\_]\!]$, the object `pixel` (of Figure 2) and the object `q` (of Figure 3) will be represented by the following addressed terms:

$$\begin{aligned}\mathtt{pixel} &\equiv [\![\, \langle\langle\langle\langle\rangle^f \leftarrow \mathtt{y} = 0\rangle^e \leftarrow \mathtt{x} = 0\rangle^d \leftarrow \mathtt{onoff} = \mathtt{true}\rangle^c \leftarrow \mathtt{set} = \cdots\rangle^b ]\!]^a \\ \mathtt{q} &\equiv [\![\langle\langle\langle\langle\langle\rangle^f \leftarrow \mathtt{y} = 0\rangle^e \leftarrow \mathtt{x} = 0\rangle^d \leftarrow \mathtt{onoff} = \mathtt{true}\rangle^c \leftarrow \mathtt{set} = \cdots\rangle^b \leftarrow \mathtt{set} = \cdots\rangle^g ]\!]^h\end{aligned}$$

The use of the same addresses $b, c, d, e, f$ in `q` as in `pixel` denotes the sharing between both object structures while $g, h$, are unshared locations.

## 3.2 Cycles

Cycles are essential in functional programming when one wants to deal with infinite data-structures in an efficient way, as is the case in lazy functional programming languages. Cycles are also used, in some implementations, to save space in the code of recursive functions.

In the context of object programming languages, cycles can be also used to express *loops* introduced in the memory (the *store*) by the imperative operators. Let us look on an example how ATRS's deal with cycles.

**Example 1 (Loop in the store, [AC96]).** Consider an object `o` which contains one single method, namely `m`. The method `m` overrides itself. In $\lambda\mathcal{O}bj^{+a}$, we represent methods with $\lambda$-abstractions ($\lambda\mathtt{s}.\mathit{body}$) whose first parameter `s` denotes the self variable. The object `o` is then:

$$[\![\langle\langle\rangle^b \leftarrow \mathtt{m} = \underbrace{(\lambda\mathtt{s}.\langle \mathtt{s} \leftarrow\!: \mathtt{m} = \lambda\mathtt{s}'.\mathtt{s}\rangle)}_{N}[\mathrm{id}]^c\rangle^d ]\!]^a$$

where $\langle \_ \leftarrow\!: \_ \rangle$ denote the imperative *cons* of an object with a method/field. For completeness, the precise sense of [id] (the local environment of the method `m`) may be ignored for the moment, it will be explained later. When `m` is invoked on the object `o`, it "self-inflicts" an override of `m` with a new body in which `s` is now bound to `o` itself. The result of this operation could be expressed as the *infinite term* defined by the fixed point equation:

$$\mathtt{o} = [\![\langle\langle\rangle^b \leftarrow \mathtt{m} = N[\mathrm{id}]^c\rangle^d \leftarrow \mathtt{m} = (\lambda\mathtt{s}'.\mathtt{s})[\mathtt{o}/\mathtt{s}; \mathrm{id}]^e\rangle^f ]\!]^a$$

Here, $[\mathtt{o}/\mathtt{s}; \mathrm{id}]$ says that in the $\lambda$-abstraction $(\lambda\mathtt{s}'.\mathtt{s})$, the free variable `s` is bound to `o`.

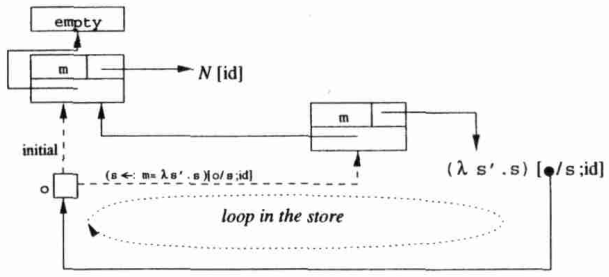



Figure 6: A Loop in the Store

This is not the ATRS approach: rather than having to deal with an infinite term (or a fixed point equation), ATRS's use a *back-pointer* (noted by •) labeled with the same address as o. The following term is the addressed term representing o, in which $\bullet^a$ denotes a back-pointer to the addressed term at location $a$, namely o itself:

$$[\![\langle\langle\rangle^b \leftarrow \mathtt{m} = N[\mathrm{id}]^c\rangle^d \leftarrow \mathtt{m} = (\lambda \mathtt{s}'.\mathtt{s})[\bullet^a/\mathtt{s};\mathrm{id}]^e\rangle^f]\!]^a$$

Figure 6 gives a graphical illustration of this example.

## 3.3 Mutation

Almost all object-oriented programming languages are not purely functional, but rather have some operations that may alter the state of objects without changing object's identity. An example of such mutation was given in Section 2.2 (Figures 4 and 5), where we see that the object denoted by p has had its structure altered by the addition of some methods, without changing its object identity. Note that the object p may be shared, and that all the expressions containing pointers to p undergo the mutation that happened at address $a$.

The key point of mutation in the setting of ATRS is the possibility to modify *in-place* the contents of any node at a given location, with respect to some precise rules. More details on computations performing mutation will be given in Section 4.

## 3.4 Syntax of Addressed Terms

An *addressed preterm* is a tree where each node receives a label (the operator symbol *e.g.*, times) and an address (*e.g.*, $a$). Of course we will not want to treat an arbitrary preterm as an addressed term, because an unconstrained preterm may denote a non-coherent memory structure. Roughly speaking, since each node in a memory has a *unique symbol* and a *unique list* of successor locations, it must be the case that all "sub-terms" at a given address in a term denote a *unique memory sub-structure.*

For instance, the instantiated "program" $\mathsf{times}^a(\mathsf{square}^b(2^c), \mathsf{square}^b(2^d))$ (quite similar to the one presented in Subsection 3.1) is not admissible, because it designates that the sons of the node at address $b$ are both at addresses $c$ and $d$. In contrast, $\mathsf{times}^a(\mathsf{square}^b(2^c), \mathsf{square}^b(2^c))$ is admissible.

Above we used the word *sub-term*, but in the presence of back-pointers, this notion does not really make sense. Indeed let $t$ be an addressed term at address $a$ containing a back-pointer $\bullet^a$; if one takes a sub-term of $t$ in the usual sense, say at address $b$, one obtains a term with a *dangling* $\bullet^a$. Therefore when defining the term at address $b$, which we write $t@b$, one has to expand $\bullet^a$ sufficiently to avoid dangling pointers. Because of this surgery, we do not call $t@b$ a "sub-term" but an *in-term.* Note that the relation "to be an in-term of" is not well-founded.

## 3.5 Rewriting

Addressed terms themselves formalize the data structures representing code and data in a machine; we now formalize the operational semantics of the machine as rewriting of addressed terms.

A rewriting rule, denoted by $l \to r$, is a pair of addressed terms with variables. As for ordinary terms, such a rule induces a reduction relation on the set of addressed terms. This relation is defined with the help of a notion of a term *matching* another term. Roughly speaking, a term $t$ matches $l$ when the variables of $l$ can be substituted by addressed terms, and its addresses by other addresses, resulting in $t$. The intuition that substitution on addressed terms is almost the same as classical term substitution is sufficient to understand the following idea.

In an addressed rewriting rule $l \to r$, $l$ and $r$ must have a common address, say $a$, at their respective roots. The idea is that the rule describes how the node at this address has to be modified for computation. Other addresses reachable from $a$ may be modified as well, and new nodes introduced by $r$.

Intuitively, given an addressed term $t$, the rewriting takes the following four steps:

1. *Find a redex in $t$, i.e.*, an in-term *matching* the left-hand side of a rule.
2. *Create fresh addresses, i.e.*, addresses not used in the current addressed term $t$, which will correspond to the new locations occurring in the right-hand side, but not in the left-hand side.
3. *Substitute the variables and addresses* of the right-hand side of the rule by their new values, as assigned by the matching of the left-hand side or created as fresh addresses. Let us call this new addressed term $u$.
4. For all $a$ that occur both in $t$ and $u$, replace in $t$ the in-terms at address $a$ by the in-term at address $a$ in $u$.

An important constraint of ATRS's is that both members of a rule must have the same address. Hence, one rejects rules like $F^a(X) \to X$, called *collapsing* or *projection* rules. We can recover the effect of such rule by adding to the signature a unary function symbol, intuitively seen as an *indirection node* and written $\lceil\ \rceil$. This constraint is realistic as, in fact, it turns to be the technique used by all actual implementations to avoid searching the memory for pointers to redirect. With that, the above rule can be expressed as $F^a(X) \to \lceil X \rceil^a$. The use of explicit indirection nodes is motivated by our wish to explicit the constraints that every rewriting step must be as close as possible to what happens in a real implementation, so that the complexity of the rewriting and the complexity of the execution on a real machine are closely correlated. It is a simple and efficient way to avoid an unbounded, global, redirection.

The last operation in the above definition of rewriting corresponds to the simulation of updates *in-place* in a memory: all over the rewritten term, address contents are modified to give an account of the sharing and mutation. This operation is the key point of the following property: any reduction starting from an addressed term results in an addressed term, *i.e.*, the coherence of the underlying memory structures is preserved by the application of any rule. Formally we have the following theorem.

**Theorem 1.** *Let $R$ be an addressed term rewriting system and $t$ be an addressed term. If $t \to u$ in $R$ then $u$ is also an addressed term.*

A formal definition of ATRS's, and the proof of the theorem, are given in the Appendix of the full version of the paper (see web pages of the authors). Section 4 gives an intuition of the way rules are defined and used, in the particular setting of $\lambda\mathcal{O}bj^{+a}$. See especially Section 4.3 in which some typical computations in $\lambda\mathcal{O}bj^{+a}$ are modeled by (addressed) rewriting.

# 4 Modules and Rules of $\lambda\mathcal{O}bj^{+a}$

The purpose of this section is to describe the rules of the framework $\lambda\mathcal{O}bj^{+a}$. The framework is described by a set of rules arranged in *modules*. The four modules are called respectively L, C, F, and I.

**L** is the *functional* module, and is essentially the calculus $\lambda\sigma^a_w$ of [BRL96]. This module alone defines the core of a purely functional programming language based on $\lambda$-calculus and weak reduction.

**C** is the *common object* module, and contains all the rules common to all instances of object calculi defined from $\lambda\mathcal{O}bj^{+a}$. It contains rules for instantiation of objects and invocation of methods.

**F** is the module of *functional update*, containing the rules needed to implement object update that also changes object identity.

**I** is the module of *imperative update*, containing the rules needed to implement object update that does not change object identity. It also provides a dynamic semantics of the `clone` operator.

The set of rules L + C + F is the instance of $\lambda\mathcal{O}bj^{+a}$ for non-mutable object calculi while L + C + I is for mutable object calculi.[1] We summarize this correspondence and the relationship between $\lambda\mathcal{O}bj^{+a}$ and classical operational semantics in the following table:

| | lambda | lambda + object |
|---|---|---|
| functional | graph-rewriting | $\lambda\mathcal{O}bj^{+a}$ (L + C + F) |
| imperative | stack & store | $\lambda\mathcal{O}bj^{+a}$ (L + C + I) |

We first introduce the syntax of $\lambda\mathcal{O}bj^{+a}$, then explain the rules.

## 4.1 Syntax of $\lambda\mathcal{O}bj^{+a}$

The syntax of $\lambda\mathcal{O}bj^{+a}$ is summarized in Figure 7. The first category of expressions is the *code* of programs. Code contains all the constructs of the calculus $\lambda\mathcal{O}bj^{+}$ [GHL98], plus an imperative update and a clone operator. Terms that define the code have no addresses, because code contains no environment and is not subject to any change during the computation (remember that addresses are meant to tell the computing engine which parts of the computation structure can change simultaneously). The second and third categories define dynamic entities, or inner structures: the *evaluation contexts*, and the *internal structure of objects* (or simply *object structures*). Terms in these two categories have explicit addresses. The last category defines *substitutions* also called *environments, i.e.,* lists of terms bound to variables, which are to be distributed and augmented over the code.

$$
\begin{array}{rll}
M,N ::= & \lambda x.M \mid MN \mid x \mid c \mid \langle\rangle \mid \langle M \leftarrow m = N\rangle \mid & \text{(Code)} \\
& M \Leftarrow m \mid \langle M \leftarrow\!: \; m = N\rangle \mid \mathsf{clone}(x) & \\
U,V ::= & M[s]^a \mid (UV)^a \mid & \text{(Evaluation Contexts)} \\
& (U \Leftarrow m)^a \mid \langle U \leftarrow m = V\rangle^a \mid & \\
& \langle U \leftarrow\!: \; m = V\rangle^a \mid \llbracket O \rrbracket^a \mid Sel^a(O,m,U) \mid & \\
& \lceil U \rceil^a \mid \bullet^a & \\
O ::= & \langle\rangle^a \mid \langle O \leftarrow m = V\rangle^a \mid \bullet^a & \text{(Object structures)} \\
s ::= & U/x; s \mid \mathrm{id} & \text{(Substitutions)}
\end{array}
$$

Figure 7: The Syntax of $\lambda\mathcal{O}bj^{+a}$

**The code category.** Code terms, written $M$ and $N$, provide the following constructs:

- Pure $\lambda$-terms, constructed from abstractions, applications, variables, and constants. This allows the definition of higher-order functions.
- Objects, constructed from the empty object $\langle\rangle$ and update operators: the functional $\langle \_ \leftarrow \_ \rangle$ and the imperative $\langle \_ \leftarrow\!: \_ \rangle$. An informal semantics of the update operators has been given in Section 2. As in [GHL98], these operators can be understood as extension as well as override operators, since an override is handled as a particular case of extension.

[1] Okasaki [Oka98] calls mutable objects *persistent* and non-mutable objects *ephemeral*.

- Method invocation ($\_ \Leftarrow \_$).
- Cloning. clone($x$) is an operator which creates a new object identity for the object pointed to by $x$ but which still shares the same object structure as the object $x$ itself (it is "shallow" as discussed in Section 2).

**Evaluation contexts.** These terms, written $U$ and $V$, model *states of abstract machines*. Evaluation contexts contain an abstraction of the temporary structure needed to compute the result of an operation. They are given addresses as they denote dynamically instantiated data structures; they always denote a term closed under the distribution of an environment. There are the following evaluation contexts:

- *Closures*, of the form $M[s]^a$, are pairs of a code and an environment. Roughly speaking, $s$ is a list of evaluation contexts that must replace the free variables in the code $M$.
- *Objects*, of the form $[\![O]\!]^a$, represent evaluated objects whose internal object structure is $O$ and whose object identity is $a$. In other words, the address $a$ plays the role of an *entry point* or *handle* to the object structure $O$, as illustrated by Figure 2.
- The terms $(UV)^a$, $(U \Leftarrow m)^a$, $\langle U \leftarrow m = V\rangle^a$, and $\langle U \leftarrow\!: m = V\rangle^a$, are the evaluation contexts associated with the corresponding code constructors. Direct sub-terms of these evaluation contexts are themselves evaluation contexts instead of code.
- The term $Sel^a(O, m, U)$ is the evaluation context associated to a method-lookup, *i.e.*, the scanning of the object structure $O$ to find the method $m$, and apply it to the object $U$. It is an auxiliary operator invoked when one sends a message to an object.
- The term $\lceil U \rceil^a$ denotes an indirection from the address $a$ to the root of the addressed term $U$.
- The term $\bullet^a$ is a *back-pointer* intended to denote cycles as explained in Sections 1.1 and 3.

**Internal Objects.** The crucial choice of $\lambda\mathcal{O}bj^{+a}$ is the use of *internal objects*, written $O$, to model object structures in memory. They are persistent structures which may only be accessed through the address of an object, denoted by $a$ in $[\![O]\!]^a$, and are never destroyed nor modified (but eventually removed by a garbage collector in implementations, of course). Since our calculus is inherently delegation-based, objects are implemented as linked lists (of fields/methods), but a more efficient array structure can be envisaged. Again, the potential presence of cycles means that object structures can contain occurrences of back-pointers $\bullet^a$.

The evaluation of a program, *i.e.*, a code term $M$, always starts in an empty environment, *i.e.*, as a closure $M[\mathrm{id}]^a$.

## 4.2 Architecture of $\lambda\mathcal{O}bj^{+a}$

The rules of $\lambda\mathcal{O}bj^{+a}$ as a computational-engine are defined in Figure 8. We will explain the rules, module by module.

**The Module L.** The module L is very similar to the calculus $\lambda\sigma_w^a$ of [BRL96], a calculus of explicit substitution enriched with addresses, to which we have added explicit indirections. Module L hence defines the core of a very simple functional programming language.

Rule (App) tells how environments have to be distributed over applications: it creates two new evaluation contexts (closures) located at new fresh addresses $b$ and $c$; each of these closures is reachable from address $a$, updated so as to contain an evaluation context of application. Note that the two occurrences of $s$ in the right-hand side of the rule contain the same addressed sub-terms. This means that these sub-terms are shared.

The Module L

$$(MN)[s]^a \to (M[s]^b\, N[s]^c)^a \qquad \text{(App)}$$
$$\big((\lambda x.M)[s]^b\, U\big)^a \to M[U/x;s]^a \qquad \text{(Bw)}$$
$$x[U/y;s]^a \to x[s]^a \quad x \not\equiv y \qquad \text{(RVar)}$$
$$x[U/x;s]^a \to \lceil U \rceil^a \qquad \text{(FVar)}$$
$$(\lceil U \rceil^b\, V)^a \to (U\, V)^a \qquad \text{(AppRed)}$$
$$\lceil (\lambda x.M)[s]^b \rceil^a \to (\lambda x.M)[s]^a \qquad \text{(LCop)}$$

The Module C

$$\langle\rangle[s]^a \to \llbracket \langle\rangle^b \rrbracket^a \qquad \text{(NO)}$$
$$(M \Leftarrow m)[s]^a \to (M[s]^b \Leftarrow m)^a \qquad \text{(SP)}$$
$$(\llbracket O \rrbracket^b \Leftarrow m)^a \to Sel^a(O, m, \llbracket O \rrbracket^b) \qquad \text{(SA)}$$
$$(\lceil U \rceil^b \Leftarrow m)^a \to (U \Leftarrow m)^a \qquad \text{(SRed)}$$
$$Sel^a(\langle O \leftarrow m = U\rangle^b, m, V) \to (U\, V)^a \qquad \text{(SU)}$$
$$Sel^a(\langle O \leftarrow n = U\rangle^b, m, V) \to Sel^a(O, m, V) \quad m \not\equiv n \qquad \text{(NE)}$$

The Module F

$$\langle M \leftarrow m = N\rangle[s]^a \to \langle M[s]^b \leftarrow m = N[s]^c\rangle^a \qquad \text{(FP)}$$
$$\langle \llbracket O \rrbracket^b \leftarrow m = V\rangle^a \to \llbracket \langle O \leftarrow m = V\rangle^c \rrbracket^a \qquad \text{(FC)}$$
$$\langle \lceil U \rceil^b \leftarrow m = V\rangle^a \to \langle U \leftarrow m = V\rangle^a \qquad \text{(FRed)}$$

The Module I

$$\langle M \leftarrow\!: m = N\rangle[s]^a \to \langle M[s]^b \leftarrow\!: m = N[s]^c\rangle^a \qquad \text{(IP)}$$
$$\langle \llbracket O \rrbracket^b \leftarrow\!: m = V\rangle^a \to \lceil \llbracket \langle O \leftarrow m = V\rangle^c \rrbracket^b \rceil^a \qquad \text{(IC)}$$
$$\langle \lceil U \rceil^b \leftarrow\!: m = V\rangle^a \to \langle U \leftarrow\!: m = V\rangle^a \qquad \text{(IRed)}$$
$$\mathsf{clone}(x)[U/y;s]^a \to \mathsf{clone}(x)[s]^a \quad x \not\equiv y \qquad \text{(SRVar)}$$
$$\mathsf{clone}(x)[\llbracket O \rrbracket^b/x;s]^a \to \llbracket O \rrbracket^a \qquad \text{(SFVar)}$$
$$\mathsf{clone}(x)[\lceil U \rceil^b/x;s]^a \to \mathsf{clone}(x)[U/x;s]^a \qquad \text{(CRed)}$$

All addresses occurring in right-hand sides but not in left-hand sides are *fresh*.

Figure 8: Rules of $\lambda\mathcal{O}bj^{+a}$

Once a substitution reaches an abstraction, a redex can be contracted by applying rule (Bw).[2] This extends the substitution by adding a pair, binding the parameter of the abstraction to the argument of the application.

Once a variable is reached by a substitution, a lookup has to be performed in the substitution to find the evaluation context to be substituted, *i.e.*, the one bound to the variable. This is described by rules (FVar), and (RVar). Note that, since modifications *must* be performed in place, and since $U$ has its own address, the only simple way to get access to $U$ from $a$ is to set an indirection (denoted by a pair of $\lceil\ \rceil$-brackets) from $a$ to the root of $U$.

The last two rules (AppRed), and (LCop) say how to get rid of indirections that could "block" the identification of redexes. Intuitively, we are here treating the situation in which address $a$ "really" has a redex, but one of its components is available only by following a redirection. In this module, an indirection blocks a reduction if the indirected node is an abstraction, and the indirection node is the left argument of an application. We have two alternative ways to get rid of such indirections, modeling choices that may be made in an implementation;

1. Redirect from the address $a$ to the root of $U$ as in rule (AppRed);
2. Copy the indirected abstraction node lying at address $b$, at the address of the indirection node $a$, as in rule (LCop). Note that the copy is only a copy of the node at $b$, not of a whole graph, since addresses in $s$ and (implicit addresses) in $\lambda x.M$ do not change. Note as well that this copy may not cause a loss in the sharing of computation since an abstraction is already a value and can not be reduced further.

Finally observe that in contrast to [BRL96], no rule is given which allows us to copy shared structures for applications and other closures. There are two reasons to do this: the first is that it could induce a *loss in the sharing* of computations since applications and closures are not values; the second (stronger) is that such closures can reduce to objects, and, as we will see later, a copy would have the same effect as a *clone of object*. We certainly do not want to have uncontrolled cloning of objects, particularly in the presence of imperative update. In fact, the way an implementer is going to handle redirections is an essential component of the design of an object oriented language. One main purpose of our approach is to make this pointer manipulation explicit in a rewriting framework.

**The Common Object Module C.** This module handles object instantiation and message sending. *Object instantiation* is defined by rule (NO) where an empty object is given an object identity. More sophisticated objects may then be obtained by functional or imperative updates, defined in modules F and I. *Message sending* is formalized by the five remaining rules, namely rule (SP), which propagates the environment into the receiver of the message, rule (SA), which performs the self-application, rules (SU) and (NE), which perform the method-lookup, and at last rule (SRed) which redirects a blocking indirection node. Note that there is no *copy* alternative to rule (SRed), since we still do not want to lose control of the cloning of objects.

**The Functional Object Module F.** Module F gives the operational semantics of a calculus of non mutable objects. It contains only three rules. Rule (FP) propagates substitutions over functional update operators, installing the evaluation context needed to proceed, while rule (FC) describes the actual update of an object of identity $b$. The update is not made in place at address $b$, hence no side effect is performed, but the result is a new object, with a new object identity $a$ which used to be the address of the evaluation context that has led to this new object. This is why we call this operator *functional* or *non mutating*. The last rule (FRed) is the way to get rid of blocking indirection nodes in the case of functional update.

**The Imperative Object Module I.** Module I contains rules for the mutation of objects (imperative update) and cloning primitive. Imperative update is formalized in a way close to the functional update. Rule (IC) differ from rule (FC) in address management, as illustrated in Section 2. Indeed look at address $b$ in rule (IC). In the left-hand side, $b$ is the identity of an object $[\![O]\!]$, when in the right-hand side it is the identity of the whole object modified by the rule. Since $b$ may be shared from anywhere in the context

[2]The name (Bw) comes from "Beta weak", the name this rule is assigned in functional languages. A more appropriate pronunciation would be "bind weakly".

of evaluation, this modification is observable *non locally* as a *side effect* or *mutation*. Moreover, since the result of this transformation has to be accessible from address $a$, an indirection node is set from $a$ to $b$. As described in Section 3, rule (IC) may create cycles because it is possible that the address $b$ is a sub-address of $V$. Module I has also a rule that redirects blocking indirection nodes in the case of imperative extension, namely rule (IRed).

The term clone($x$) is a primitive for cloning, that performs a lookup in the environment as variable access, but always creates a copy of the found object. As we said before, by copy, we mean a *shallow* copy that creates a new object identity for an existing object even though $x$ and clone($x$) share the same object structure. Rule (CRed) gets rid of a blocking indirection by local redirection.

## 4.3 Examples in $\lambda\mathcal{O}bj^{+a}$

We first give an example showing a functional object which extends itself [GHL98].

**Example 2.** Let

$$\texttt{self\_ext} \triangleq \langle\,\langle\rangle \leftarrow \texttt{add\_n} = \underbrace{\lambda\texttt{self}.\langle\texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle}_{N}\,\rangle$$

The reduction of $M \triangleq (\texttt{self\_ext} \Leftarrow \texttt{add\_n})$ in $\lambda\mathcal{O}bj^{+a}$ is as follows:

$$\begin{aligned}
M[\mathrm{id}]^a &\to^* (\langle\langle\rangle[\mathrm{id}]^d \leftarrow \texttt{add_n} = N[\mathrm{id}]^c\rangle^b \Leftarrow \texttt{add_n})^a && (1)\\
&\to (\langle\llbracket\langle\rangle^e\rrbracket^d \leftarrow \texttt{add_n} = N[\mathrm{id}]^c\rangle^b \Leftarrow \texttt{add_n})^a && (2)\\
&\to (\llbracket\underbrace{\langle\langle\rangle^e \leftarrow \texttt{add_n} = N[\mathrm{id}]^c\rangle^f}_{O}\rrbracket^b \Leftarrow \texttt{add_n})^a && (3)\\
&\to Sel^a(O, \texttt{add_n}, \llbracket O\rrbracket^b) && (4)\\
&\to ((\lambda\texttt{self}.\langle\texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle)[\mathrm{id}]^c\ \llbracket O\rrbracket^b)^a && (5)\\
&\to \langle\texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle[\llbracket O\rrbracket^b/\texttt{self};\mathrm{id}]^a && (6)\\
&\to \langle\texttt{self}[\llbracket O\rrbracket^b/\texttt{self};\mathrm{id}]^h \leftarrow \texttt{n} = (\lambda\texttt{s}.1)[\llbracket O\rrbracket^b/\texttt{self};\mathrm{id}]^g\rangle^a && (7)\\
&\to \langle\lceil\llbracket O\rrbracket^b\rceil^h \leftarrow \texttt{n} = (\lambda\texttt{s}.1)[\llbracket O\rrbracket^b/\texttt{self};\mathrm{id}]^g\rangle^a && (8)\\
&\to \langle\llbracket O\rrbracket^b \leftarrow \texttt{n} = (\lambda\texttt{s}.1)[\llbracket O\rrbracket^b/\texttt{self};\mathrm{id}]^g\rangle^a && (9)\\
&\to \llbracket\langle O \leftarrow \texttt{n} = (\lambda\texttt{s}.1)[\llbracket O\rrbracket^b/\texttt{self};\mathrm{id}]^g\rangle^h\rrbracket^a && (10)
\end{aligned}$$

In (1), two steps are performed to distribute the environment inside the extension, using rules (SP), and (FP). In (2), the empty object is given an object-structure and an object identity (NO). In (3), this new object is functionally extended (FC), hence it shares the structure of the former object but has a new object-identity. In (4), and (5), two steps (SA) (SU) perform the look up of method add_n. In (6) we apply (Bw). In (7), the environment is distributed inside the functional extension (FP). In (8), self is replaced by the object it refers (FVar), setting an indirection from $h$ to $b$. In (9) the indirection is eliminated (FRed). Step (10) is another functional extension (FC). There is no redex in the last term of the reduction.

Some sharing of structures appears in the example above, since *e.g.* $\llbracket O\rrbracket^b$ has several occurrences in some terms of the derivation. However, this example does not show any sharing of computation. The following is a very simple example of a simple "rewriting step" which gives an account of sharing of computation.

**Example 3.** The addressed term

$$\mathrm{x}[\langle\rangle[\mathrm{id}]^a/\mathrm{x};\langle\rangle[\mathrm{id}]^a/\mathrm{y};\mathrm{id}]^b$$

rewrites in one step to

$$\mathrm{x}[\llbracket\langle\rangle^c\rrbracket^a/\mathrm{x};\llbracket\langle\rangle^c\rrbracket^a/\mathrm{y};\mathrm{id}]^b$$

by rule (NO). Note how the instance of the new object of identity $a$ is shared by both variables x and y, which are in fact aliases for this object.

The next example shows a rewriting step performing mutation, then the introduction of a cycle in an acyclic addressed term by an imperative update.

**Example 4.** The term

$$\mathtt{x}[(\llbracket\langle\rangle^c\rrbracket^a \leftarrow\colon \mathtt{m} = (\lambda\mathtt{s}.\mathtt{s})[\mathrm{id}]^d)^e/\mathtt{x}; \llbracket\langle\rangle^c\rrbracket^a/\mathtt{y}; \mathrm{id}]^b$$

reduces in one step by rule (IC) to $\mathtt{x}[\lceil O\rceil^e/\mathtt{x}; O/\mathtt{y}; \mathrm{id}]^b$, where

$$O \equiv \llbracket\langle\langle\rangle^c \leftarrow \mathtt{m} = (\lambda\mathtt{s}.\mathtt{s})[\mathrm{id}]^d\rangle^f\rrbracket^a$$

Note how the object referred by y (in fact y becomes an alias of x) undergoes the extension with the new method m, while there was no local operation intended to perform this extension. This is why such a rewriting is called a mutation: from the point of view of y, the referred object has changed—not only syntactically, but also observationally—due to the evaluation of an evaluation context somewhere else in the addressed term.

**Example 5.** The term

$$\langle\llbracket\langle\rangle^a\rrbracket^b \leftarrow\colon \mathtt{m} = (\lambda\mathtt{self}.\mathtt{x})[\llbracket\langle\rangle^a\rrbracket^b/\mathtt{x}; \mathrm{id}]^c\rangle^d$$

reduces by (IC) to

$$\lceil\langle\llbracket\langle\rangle^a \leftarrow \mathtt{m} = (\lambda\mathtt{self}.\mathtt{x})[\bullet^b/\mathtt{x}; \mathrm{id}]^c\rangle^e\rrbracket^b\rceil^d$$

This is another example of a loop in the store, easily visualizable by the occurrence of $\bullet$, as the object of identity $b$ contains now a method that references itself. Note how address $d$ redirects to address $b$ where the result of the evaluation context previously assigned address $d$ is stored.

# 5 Conclusions

We have presented a framework to describe many object-based calculi. To our knowledge, this framework has no equivalent in the literature; it has the following features:

- It is computationally complete since the $\lambda$-calculus is explicitly built-in to the language of expressions;
- It gives an account of the delegation-based techniques of inheritance;
- It is compatible with dynamic object extension and self-extension in the style of [GHL98];
- It is generic, due to the partition of rules in independent modules, which can be combined to model (for example) functional *vs.* imperative implementations;
- It supports the analysis of implementations at the level of resource usage, as it models sharing of computations and sharing of storage, and each computation-step in the calculus corresponds to a constant-cost computation in practice;
- It is founded on a novel and mathematically precise theory, *i.e.*, addressed term rewriting systems.

Furthermore, $\lambda\mathcal{O}bj^{+a}$ is generic in the sense that many strategies may be implemented. We have not given any definition of particular strategies since we do not want to privilege one strategy over another. However, we believe it is interesting to investigate what a strategy is, and how it may be defined. The approach for functional languages studied in [BRL96] should be generalizable to $\lambda\mathcal{O}bj^{+a}$: from a very general point of view, a strategy is a binary relation between addressed terms and addresses. The addresses, in relation with a given term, determines which redexes of the term has to be reduced next (note that in a given term at a given address, at most one rule applies). This is a restriction *w.r.t.* the calculus in which not all the redexes may be reduced. If this relation is a one-to-many relation, the strategy is *non deterministic.* If this relation is a function, then the strategy is *deterministic and sequential.* If this function is computable, then the strategy is *computable.* Implementors and designers of languages are usually interested in some subclass of the computable strategies, that follows some locality principle—namely that a lot of reductions happen in a small connected part of the whole structure before "jumping" to another distant part. The definition of such

strategies—which includes the usual call-by-value, call-by-name, call-by-reference, *etc.*—can be expressed using a very simple set of inference rules (those rule will be collected in another module of $\lambda\mathcal{O}bj^{+a}$ not presented in this paper). These rules can be combined, as basic building blocks, provided possible conditions on their application, to define a lot of strategies.

Finally, we plan to extend $\lambda\mathcal{O}bj^{+a}$ to handle the embedding-based technique of inheritance, following [LLL99], to include a type system, compliant with imperative feature and allowing to type objects extending themselves, following [LLL98, GHL98], and to build a prototype of $\lambda\mathcal{O}bj^{+a}$, from which it should be easy to embed specific calculi and to make experiments on the design of realistic object oriented languages.

# References


[AC96] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Massachusetts, 1985.

[Aug84] L. Augustson. A compiler for lazy ML. In *Symposium on Lisp and Functional Programming*, pages 218–227. The ACM Press, 1984.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[BF98] V. Bono and K. Fisher. An Imperative, First-Order Calculus with Object Extension. In *European Conference for Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 462–497. Springer-Verlag, 1998.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[BR95] R. Bloo and K. H. Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *Computer Science in the Netherlands*, pages 62–72, 1995.

[BRL96] Z.-E.-A. Benaissa, K.H. Rose, and P. Lescanne. Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution. In *Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, pages 393–407. Springer-Verlag, 1996.

[BVEG+87] H. P. Barendregt, M. C. J. D. Van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term Graph Rewriting. In *Parallel Architectures and Languages Europe*, number 259 in Lecture Notes in Computer Science, pages 141–158. Springer-Verlag, 1987.

[Car95] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[Cha93] C. Chambers. The Cecil language specification, and rationale. Technical Report 93-03-05, Department of Computer Science and Engineering, University of Washington, USA, 1993.

[Chu41] A. Church. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, N. J., 1941.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers, 1990.

[FF89] M. Felleisen and D. P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69:243–287, 1989.

[FH92] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102, 1992.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[GHL98] P. Di Gianantonio, F. Honsell, and L. Liquori. A Lambda Calculus of Objects with Self-inflicted Extension. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 166–178. The ACM Press, 1998.

[Kah87] G. Kahn. Natural semantics. Technical Report 601, Institut National de Recherche en Informatique et en Automatique, Sophia Antipolis, France, 1987.

[Klo90] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter 6. Oxford University Press, 1990.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6, 1964.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.

[LDLR99] F. Lang, D. Dougherty, P. Lescanne, and K. Rose. Addressed term rewriting systems. Technical Report RR 1999-30, Laboratoire de l'informatique du parallèlisme, ENS de Lyon, France, 1999. Available online at `ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1999/RR1999-30.ps.Z`.

[Les94] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In *Principles of Programming Languages*, pages 60–69, 1994.

[Lév80] J.-J. Lévy. Optimal reductions in the lambda-calculus. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.

[LLL98] F. Lang, P. Lescanne, and L. Liquori. A framework for defining object-calculi. Technical Report RR1998-51, Laboratoire de l'informatique du parallèlisme, ENS de Lyon, France, 1998.

[LLL99] F. Lang, P. Lescanne, and L. Liquori. A framework for defining object-calculi (extended abstract). In J.M. Wing, J. Woodcook, and J. Davies, editors, *World Congress on Formal Methods in the Design of Computing Systems*, number 1709 in Lecture Notes in Computer Science, pages 963–982. Springer-Verlag, 1999.

[Mar92] L. Maranget. Optimal Derivations in Weak Lambda Calculi and in Orthogonal Rewriting Systems. In *Principles of Programming Languages*, pages 255–268, 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[Oka98] Ch. Okasaki. *Purely Functional Data Structures.* Cambridge U. Press, 1998.

[PJ87] S. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Plo81] Gordon Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[Plu99] D. Plump. Term graph rewriting. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific, 1999. To appear.

[PvE93] M. J. Plasmeijer and M. C. D. J. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* International Computer Science Series. Addison-Wesley, 1993.

[Ros96] K. H. Rose. *Operational Reduction Models for Functional Programming Languages.* PhD thesis, DIKU, København, Denmark, 1996. DIKU report 96/1.

[Sto77] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Tai92] A. Tailvalsaari. Kevo, a prototype-based object-oriented language based on concatenation and modules operations. Technical Report LACIR 92-02, University of Victoria, Canada, 1992.

[Tof90] M. Tofte. Type inference for polymorphic references. *Information and Computation,* 89(1):1–34, 1990.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience,* 9:31–49, 1979.

[US87] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Object-Oriented Programming, Systems, Languages, and Applications,* pages 227–241. The ACM Press, 1987.

[Wad71] C. P. Wadsworth. *Semantics and pragmatics of the lambda calculus.* PhD thesis, Oxford, 1971.

[WF94] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation,* 115(1), 1994.

# Strong normalisation of Herbelin's explicit substitution calculus with substitution propagation

Roy Dyckhoff[1]* and Christian Urban[2]

[1] University of St Andrews rd@dcs.st-and.ac.uk
[2] University of Cambridge cu200@dpmms.cam.ac.uk

**Abstract.** Herbelin presented (at CSL'94) a simple sequent calculus for minimal implicational logic, extensible to full first-order intuitionistic logic, with a complete system of cut-reduction rules which is both confluent and strongly normalising. Some of the cut rules may be regarded as rules to construct explicit substitutions. He observed that the addition of a cut permutation rule, for propagation of such substitutions, breaks the proof of strong normalisation; the implicit conjecture is that the rule may be added without breaking strong normalisation. We prove this conjecture, thus showing how to model beta-reduction in his calculus (extended with rules to allow cut permutations).

## 1 Introduction

Herbelin gave in [8] a calculus for minimal implicational logic, using a notation for proof terms that, in contrast to the usual lambda-calculus notation for natural deduction, brings head variables to the surface. It is thus a sequent calculus, with the nice feature that its cut-free terms are in a natural 1-1 correspondence with the normal terms of the simply typed $\lambda$-calculus. Other intuitionistic connectives can be added without difficulty.

The cut rules of the calculus are in part analogous to explicit substitution constructors [11] and certain auxiliary operators (e.g. list concatenation); the only exception is a rule that in some circumstances constructs an auxiliary term and in others constructs a term analogous to a $\beta$-redex. Herbelin showed strong normalisation and confluence of a complete system of rules for eliminating cuts, observed that the addition of a further "cut permutation" rule, needed to allow explicit substitutions to propagate properly, as required for the simulation of $\beta$-reduction of the simply typed $\lambda$-calculus, would break the strong normalisation proof, thus raising the question of whether it also broke the strong normalisation result. In the present paper we answer this question; strong normalisation holds for the calculus with the addition of this rule (and of other cut permutation rules, to retain confluence).

Herbelin's calculus can thus be seen in two ways:

1. In the cut-free case it is a natural basis [5] for automated proof search in logic programming, since it is a sequent calculus but free from the permutation problems of Gentzen's calculus;
2. In the general case, when extended with the cut permutation rules, it can simulate $\beta$-reduction and thus, with its strong proof-theoretic foundations, may be a natural basis for implementation of functional languages.

Our proof uses standard techniques, e.g. from [2]. That paper, in its use of recursive path ordering techniques, shows the use of higher-order rewriting to be unnecessary, by translating to a first-order system. In order not to obscure our argument, we omit the details of this translation; the conscientious reader is invited to fill in the missing details.

* Thanks are due to the second author's family and the Dresden University of Technology for support of various kinds during a visit to Dresden covering the genesis of this paper.

## 2 Technical Background

We use a notation developed in [6], since it distinguishes the different kinds of cut term and in proofs emphasises the role of the *stoup* formula. The syntax of the calculus is as follows: formulae $A$ are as in implicational logic, there are (term-)variables $x, y, \ldots$, there are two kinds $Ms, M$ of proof-term and two kinds of sequent, one with and one without a *stoup* formula, which is written in the first case below the sequent arrow. $Ms$ is used in the stoup sequents; although the notation may suggest a list, not all terms $Ms$ are lists. Notions of "free" and "bound" and variable conventions are as usual, with variable binding not just in $\lambda$-terms but also in $cut_2$ and $cut_4$ terms. *Contexts* $\Gamma$ are finite functions from variables to formulae; $\Gamma, x : A$ indicates the extension, by the assignment of $A$ to $x$, of a context $\Gamma$ in which there is no assignment to $x$.

**Syntax of cut-free terms**

$$Ms ::= [] \mid (M :: Ms)$$
$$M ::= (x; Ms) \mid \lambda x.M$$

**Logical/Typing rules**

$$\frac{\Gamma \Rightarrow M : A \quad \Gamma \underset{B}{\longrightarrow} Ms : C}{\Gamma \underset{A \supset B}{\longrightarrow} (M :: Ms) : C}\, S\supset \qquad \frac{}{\Gamma \underset{A}{\longrightarrow} [] : A}\, Ax$$

$$\frac{\Gamma, x : A \underset{A}{\longrightarrow} Ms : B}{\Gamma, x : A \Rightarrow (x; Ms) : B}\, Sel \qquad \frac{\Gamma, x : A \Rightarrow M : B}{\Gamma \Rightarrow \lambda x.M : A \supset B}\, R\supset$$

**Syntax of cut terms** (type information is omitted in the type-free case)

$$Ms ::= cut_1^A(Ms, Ms) \mid cut_2^A(M, x.Ms)$$
$$M ::= cut_3^A(M, Ms) \mid cut_4^A(M, x.M)$$

**Logical/Typing rules for Cuts**

$$\frac{\Gamma \underset{B}{\longrightarrow} Ms : A \quad \Gamma \underset{A}{\longrightarrow} Ms' : C}{\Gamma \underset{B}{\longrightarrow} cut_1^A(Ms, Ms') : C}\, Cut_1 \qquad \frac{\Gamma \Rightarrow M : A \quad \Gamma, x : A \underset{B}{\longrightarrow} Ms : C}{\Gamma \underset{B}{\longrightarrow} cut_2^A(M, x.Ms) : C}\, Cut_2$$

$$\frac{\Gamma \Rightarrow M : A \quad \Gamma \underset{A}{\longrightarrow} Ms : B}{\Gamma \Rightarrow cut_3^A(M, Ms) : B}\, Cut_3 \qquad \frac{\Gamma \Rightarrow M : A \quad \Gamma, x : A \Rightarrow M' : B}{\Gamma \Rightarrow cut_4^A(M, x.M') : B}\, Cut_4$$

**Rules for cut-reduction**

Let $ES$ denote the system of "explicit substitution" rules $1 \ldots 4$ on terms, and $CC$ the system of "commuting cuts" rule 5 on terms:

1. (a) $cut_1^A([], Ms) \rightsquigarrow Ms$
   (b) $cut_1^A((M :: Ms), Ms') \rightsquigarrow (M :: cut_1^A(Ms, Ms'))$
2. (a) $cut_2^A(M, x.[]) \rightsquigarrow []$
   (b) $cut_2^A(M, x.(M' :: Ms)) \rightsquigarrow (cut_4^A(M, x.M') :: cut_2^A(M, x.Ms))$
3. (a) $cut_3^A((x; Ms), Ms') \rightsquigarrow (x; cut_1^A(Ms, Ms'))$
   (b) $cut_3^A(\lambda y.M, []) \rightsquigarrow \lambda y.M$
4. (a) $cut_4^A(M, x.(y; Ms)) \rightsquigarrow (y; cut_2^A(M, x.Ms)) \quad (y \neq x)$
   (b) $cut_4^A(M, x.(x; Ms)) \rightsquigarrow cut_3^A(M, cut_2^A(M, x.Ms))$
   (c) $cut_4^A(M, x.\lambda y.M') \rightsquigarrow \lambda y.cut_4^A(M, x.M')$
5. (a) $cut_1^A(cut_1^B(Ms, Ms'), Ms'') \rightsquigarrow cut_1^B(Ms, cut_1^A(Ms', Ms''))$
   (b) $cut_2^A(M, x.cut_1^B(Ms, Ms')) \rightsquigarrow cut_1^B(cut_2^A(M, x.Ms), cut_2^A(M, x.Ms'))$
   (c) $cut_3^A(cut_3^B(M, Ms), Ms') \rightsquigarrow cut_3^B(M, cut_1^A(Ms, Ms'))$
   (d) $cut_4^A(M, x.cut_3^B(M', Ms)) \rightsquigarrow cut_3^B(cut_4^A(M, x.M'), cut_2^A(M, x.Ms))$

Strictly speaking, the rules 1 and 3 are not explicit substitution reduction rules but auxiliary rules. The last of all these rules, 5(d), is the one mentioned in the introduction as the proper propagation of an explicit substitution, e.g. inside a "beta-redex". The other $CC$ rules are added to ensure confluence once that rule is added. We allow rule applications inside terms, i.e. we in fact consider the contextual closures of all rules, as usual.

We need to consider one further rule, deliberately omitted from group 3:

**B** $\quad cut_3^{A\supset B}(\lambda x.M, (M' :: Ms)) \rightsquigarrow cut_3^B(cut_4^A(M', x.M), Ms)$

which generates a $cut_4$-instance; to simulate ordinary beta-reduction, this may need to propagate into its body $M$; the latter may be a $cut_3$-term, hence the utility of rule 5(d), and so on.

Herbelin [8] showed that a system of rules, essentially $ES + B$ with different terminology, is complete, confluent and (for typed terms) strongly normalising. Note that the $CC$-rules are NOT required for completeness: without them, a strategy that (more or less) ignores cuts whose arguments are cuts is imposed.

A simpler (but less direct) proof of his SN theorem using the multiset path ordering theorem is given in [6]; the termination comes from the ordering of all the *cut* operators as greater than the non-cut operators, with $cut^A > cut^B$ for $A > B$ and with $cut_4 = cut_2 > cut_3 = cut_1$ for *cut* operators with the same type annotation.

The addition of any of the rules in 5 breaks both of these proofs, because of the switching of the types.

Note that there are no rules allowing permutation of $cut_2$ or $cut_4$ operators with $cut_2$ or $cut_4$ operators.

Routinely, "Subject Reduction" holds for all these reduction rules; thus, all the results (e.g. confluence) that we state or prove for untyped terms hold also for the typed terms.

## 3 Pure Terms

There are two obvious (and in fact equivalent) candidates for the class of terms that we will use to interpret lambda terms: those free of $(ES + CC)$-redexes and those free of all instances (other than $B$-redexes) of *cut*: we choose the latter.

**Definition 1 ($(ES + CC)$-normality; purity).**

1. *A term is* $(ES + CC)$-normal *iff it is free of all* $ES + CC$*-redexes;*
2. *A term is* pure *iff it is free of all instances (other than* $B$*-redexes) of cut.*

By induction, $(ES + CC)$-normal terms $Ms$ are of the form $[]$ or $M :: Ms'$. Pure terms are $(ES + CC)$-normal, since all $(ES + CC)$-redexes are instances of *cut*; but the converse also holds:

**Proposition 2.** *$(ES + CC)$-normal terms $Ms, M$ are pure.*

*Proof.* The $Ms$ case depends just on the $M$ case. We proceed by induction and case analysis; we show that if an $(ES + CC)$-normal term $M$ is an instance of *cut*, then it is a redex; our inductive hypothesis is that all smaller $(ES + CC)$-normal terms are pure. Consider the cases for $M$ an instance of *cut*:

1. $cut_3((x; Ms), Ms')$ is a 3(a)-redex;
2. $cut_3(\lambda x.M', [])$ is a 3(b)-redex;
3. $cut_3(\lambda x.M', (M'' :: Ms))$ is a $B$-redex;
4. $cut_3(cut_3(M', Ms), Ms')$ is a 5(c)-redex;
5. $cut_3(cut_4(M', x.M''), Ms)$ is not allowed, since (by inductive hypothesis) $cut_4(M', x.M'')$ is pure, contrary to its being a (non $B$-redex) *cut*-term;
6. $cut_4(M', x.(z; Ms))$ is a 4(a) or 4(b)-redex;
7. $cut_4(M', x.\lambda y.M'')$ is a 4(c)-redex;
8. $cut_4(M', x.cut_3(M'', Ms))$ is a 5(d)-redex;

9. $cut_4(M', x.cut_4(M'', y.M'''))$ is not allowed, for the same reason as in (5) above. ■

It follows that the $(ES + CC)$-normal/pure terms $M$ are of the form $(y; Ms)$, $\lambda x.M'$ or $cut_3(\lambda y.M', (M'' :: Ms))$.

**Definition 3 (Implicit substitution; concatenation; generalised application).** *For pure terms $Ms, Ms', M, M'$, and for the variable $x$, we define by simultaneous induction on the structure of $Ms$ (resp. $Ms$, resp. $M$, resp. $M'$)*

1. *The* concatenation[1] $Ms@Ms'$ *of* $Ms$ *with* $Ms'$*;*
2. *The* implicit substitution $[M/x]Ms$ *of* $M$ *for* $x$ *in* $Ms$*;*
3. *The* generalised application $\{M\}Ms$ *of* $M$ *to* $Ms$*;*
4. *The* implicit substitution $[M/x]M'$ *of* $M$ *for* $x$ *in* $M'$.

*as follows:*

$$
\begin{array}{rcl}
[]@Ms' & =_{def} & Ms' \\
(M'' :: Ms'')@Ms' & =_{def} & (M'' :: (Ms''@Ms'))
\end{array}
$$

$$
\begin{array}{rcl}
[M/x][] & =_{def} & [] \\
[M/x](M'' :: Ms'') & =_{def} & ([M/x]M'' :: [M/x]Ms'')
\end{array}
$$

$$
\begin{array}{rcl}
\{(y; Ms'')\}Ms & =_{def} & (y; (Ms''@Ms)) \\
\{\lambda y.M''\}[] & =_{def} & \lambda y.M'' \\
\{\lambda y.M''\}(M''' :: Ms'') & =_{def} & cut_3(\lambda y.M'', (M''' :: Ms'')) \\
\{cut_3(\lambda y.M'', (M''' :: Ms''))\}Ms & =_{def} & cut_3(\lambda y.M'', (M''' :: (Ms''@Ms)))
\end{array}
$$

$$
\begin{array}{rcll}
[M/x](y; Ms'') & =_{def} & (y; [M/x]Ms'') & (y \neq x) \\
[M/x](x; Ms'') & =_{def} & \{M\}[M/x]Ms'' & \\
[M/x](\lambda y.M'') & =_{def} & \lambda y.[M/x]M'' & \\
[M/x]cut_3(\lambda y.M'', (M''' :: Ms'')) & =_{def} & cut_3(\lambda y.[M/x]M'', ([M/x]M''' :: [M/x]Ms'')). &
\end{array}
$$

Note that the mutual induction is straightforward; first, concatenation is well-defined (as usual); second, generalised application has a definition depending only on concatenation; finally, the two forms of implicit substitution depend on simpler instances of themselves and of each other and on instances of generalised application.

## 4 Lambda Calculus

Our exposition of the lambda calculus uses approximately the notation of [10]. Lambda terms $N$ and lists $Ns$ of lambda terms are defined by the grammar

$$
\begin{array}{rcl}
N & ::= & (x\ Ns) \mid (\lambda x.N) \mid ((\lambda x.N)N\,Ns) \\
Ns & ::= & [] \mid (N :: Ns)
\end{array}
$$

in which, if we omit the last production for $N$, we get just the normal terms. Note that, for example, a term of the form $((\lambda x.N)N'Ns)$ is NOT the term list $((\lambda x.N) :: (N' :: Ns))$; it is a term. The structure of this inductive definition is intended to make certain parts of the normalisation proof in [10] easy, but no definition of substitution is given in [10]; essentially, the translation to standard

[1] This is just the usual concatenation of lists, included here for completeness.

notation is used, then the standard definition is used, then one translates back. We remedy this minor oversight as follows:

We define implicit substitution $[N/x]Ns$ of $N$ for $x$ in $Ns$ (and similarly for substitution in $N'$) as follows, by induction on the structure of $Ns$ (resp. $N'$); we need an auxiliary definition of a term $\{N\}Ns$, by induction on the structure of $N$ (and a subsidiary induction in one case on $Ns$), to apply the term $N$ to the list $Ns$ of arguments in the usual way (this is not the construction of a list, but of a term):

$$\begin{aligned}
[N/x][] &=_{def} [] \\
[N/x](N' :: Ns) &=_{def} ([N/x]N' :: [N/x]Ns)
\end{aligned}$$

$$\begin{aligned}
\{(y\ Ns')\}Ns &=_{def} (y\ (Ns'@Ns)) \\
\{(\lambda y.N)\}[] &=_{def} (\lambda y.N) \\
\{(\lambda y.N)\}(N' :: Ns) &=_{def} ((\lambda y.N)N'Ns) \\
\{((\lambda y.N)N'Ns)\}Ns' &=_{def} ((\lambda y.N)N'(Ns@Ns'))
\end{aligned}$$

$$\begin{aligned}
[N/x](y\ Ns) &=_{def} (y\ [N/x]Ns) \quad (y \neq x) \\
[N/x](x\ Ns) &=_{def} \{N\}[N/x]Ns \\
[N/x](\lambda y.N') &=_{def} (\lambda y.[N/x]N') \\
[N/x]((\lambda y.N')N''Ns) &=_{def} ((\lambda y.[N/x]N')([N/x]N''[N/x]Ns))
\end{aligned}$$

where @ is for the standard function that concatenates two lists.

Termination of this definition is as in the previous section. The equivalence between this definition and the usual definition of substitution (with the usual notation) is a tedious exercise.

*$\beta$-reduction* is thus the reduction of a term by careful replacement of a subterm (the *$\beta$-redex*) of the form $((\lambda x.N)N'Ns)$ by the *reduct* $\{[N'/x]N\}Ns$ in a single step.

## 5 Interpretation of Lambda Calculus

We now define (really trivial) bijective interpretations $(.)^\bullet$ of $\lambda$-terms $N$ as the pure terms of the form $M$ and $(.)^{\bullet\bullet}$ of term-lists $Ns$ as the pure terms of the form $Ms$:

$$\begin{aligned}
(x\ Ns)^\bullet &=_{def} (x; Ns^{\bullet\bullet}) \\
(\lambda x.N)^\bullet &=_{def} (\lambda x.N^\bullet) \\
((\lambda x.N)N'Ns)^\bullet &=_{def} cut_3(\lambda x.N^\bullet, (N'^\bullet :: Ns^{\bullet\bullet}))
\end{aligned}$$

$$\begin{aligned}
[]^{\bullet\bullet} &=_{def} [] \\
(N :: Ns)^{\bullet\bullet} &=_{def} (N^\bullet :: Ns^{\bullet\bullet})
\end{aligned}$$

**Proposition 4.** *For $\lambda$-terms $N, N'$ and term lists $Ns, Ns'$, the following hold:*

1. $(Ns@Ns')^{\bullet\bullet} = Ns^{\bullet\bullet}@Ns'^{\bullet\bullet}$;
2. $([N/x]Ns)^{\bullet\bullet} = [N^\bullet/x]Ns^{\bullet\bullet}$;
3. $(\{N\}Ns)^\bullet = \{N^\bullet\}Ns^{\bullet\bullet}$;
4. $([N/x]N')^\bullet = [N^\bullet/x]N'^\bullet$.

*Proof.* Routine. ∎

## 6 Strong Normalisation and Confluence of $ES + CC$

**Proposition 5.** *The system $ES + CC$ is strongly normalising (SN).*

*Proof.* A lexicographic path order suffices, completely ignoring all the type information, with the $cut_2 = cut_4$ operators equal, and greater than $cut_1 = cut_3$, and with all *cut* operators greater than the non-*cut* operators. We rely throughout on the exposition of the lexicographic path order given in [1] rather than that in [2]. For an alternative proof using a polynomial interpretation, see Appendix A. ■

**Proposition 6.** *The system $ES + CC$ is confluent.*

*Proof.* By Proposition 5, it suffices to check the local confluence; for details see Appendix B. ■

**Definition 7.** *For a term $Ms$ or $M$, its* purification *is its $(ES + CC)$-normal form, written $\overline{Ms}$ (resp. $\overline{M}$).*

**Lemma 8.** *If $M \leadsto^*_{ES+CC} M'$, then $\overline{M} = \overline{M'}$. (Similarly for $Ms$.)*

*Proof.* Trivial. ■

## 7 Simulation of $\beta$-reduction

**Proposition 9.** *Let $M, M', Ms, Ms'$ be pure terms. Then*

1. $cut_1(Ms, Ms') \leadsto^*_{ES+CC} Ms@Ms'$;
2. $cut_2(M, x.Ms) \leadsto^*_{ES+CC} [M/x]Ms$;
3. $cut_3(M, Ms) \leadsto^*_{ES+CC} \{M\}Ms$;
4. $cut_4(M, x.M') \leadsto^*_{ES+CC} [M/x]M'$.

*Proof.* The first part is trivial; the third part is proved by induction; the remaining two parts are proved by a simultaneous induction on the heights of the LHS terms. For details see Appendix C. ■

The above may be regarded as a weak normalisation result (for $ES + CC$), since we may use it to purify any innermost (non-$B$)-redex, repeating this operation until all such redexes are eliminated.

**Corollary 10.** *For pure terms $M, M'$ and $Ms$,*

$$cut_3(cut_4(M', x.M), Ms) \leadsto^*_{ES+CC} \{[M'/x]M\}Ms.$$ ■

**Theorem 11.** *If $N_1 \leadsto_\beta N_2$ in the $\lambda$-calculus, then $N_1^\bullet$ reduces to $N_2^\bullet$ by a $B$-reduction followed by 0 or more $ES + CC$ reductions.*

*Proof.* Consider first the case where $N_1$ is the $\beta$-redex, and so of the form $((\lambda x.N)N'Ns)$, for terms $N, N'$ and term list $Ns$. Thus, $N_2 = \{[N'/x]N\}Ns$. By Corollary 10, the reduct

$$cut_3(cut_4(N'^\bullet, x.N^\bullet), Ns^{\bullet\bullet})$$

of the $B$-redex $N_1^\bullet = cut_3(\lambda x.N^\bullet, (N'^\bullet :: Ns^{\bullet\bullet}))$ is $(ES + CC)$-reducible to $\{[N'^\bullet/x]N^\bullet\}Ns^{\bullet\bullet}$. By Proposition 4, this is just $N_2^\bullet$. The general case, where the reduction is not at the root position of $N_1$, follows by induction on the structure of $N_1$. ■

In other words, the system $(ES + CC + B)$ of rules acting on the untyped terms *simulates* $\beta$-reduction of the untyped $\lambda$-calculus; similarly for typed terms and the typed $\lambda$-calculus.

## 8 $\beta$-reduction

We may now define a rule $\beta$ on pure terms, typed or untyped, omitting the types in the latter case:

$\beta \qquad cut_3^{A\supset B}(\lambda x.M, M' :: Ms) \rightsquigarrow_\beta \overline{cut_3^B(cut_4^A(M', x.M), Ms)}$

Note that, in the untyped case, the RHS of this is, by Corollary 10, just $\{[M'/x]M\}Ms$. The correspondence between pure untyped terms and the terms of the untyped $\lambda$-calculus routinely extends to $\beta$-reduction.

Thus, a $\beta$-reduction is a single $B$-reduction followed by purification, i.e. zero or more $(ES{+}CC)$-reductions to normal form.

**Proposition 12.** *The system, on pure typed terms, consisting just of the rule $\beta$ is SN.*

*Proof.* Use, for example, the proof, in different notation, in [10]. ∎

If we had a direct proof of Proposition 12, then we would have shown the strong normalisation of the simply-typed $\lambda$-calculus.

**Definition 13.** *For any term $M$, we define $\|M\|$ to be the maximal length of all $\beta$-reduction sequences from $\overline{M}$ if the latter is $\beta$-SN; otherwise we define $\|M\| = \infty$. (Similarly for Ms.)*

**Corollary 14.** *For every typed term $M$, we have $\|M\| < \infty$. (Similarly for Ms.)*

*Proof.* By Proposition 12 and König's Lemma. ∎

**Lemma 15.** *For pure terms $M, Ms, M', M'', Ms'$, with $M \rightsquigarrow_\beta^* M'$ and $Ms \rightsquigarrow_\beta^* Ms'$, we have*

1. $\{M\}Ms \rightsquigarrow_\beta^* \{M'\}Ms$;
2. $\{M\}Ms \rightsquigarrow_\beta^* \{M\}Ms'$;
3. $[M/x]Ms \rightsquigarrow_\beta^* [M'/x]Ms$;
4. $[M/x]Ms \rightsquigarrow_\beta^* [M/x]Ms'$;
5. $[M/x]M'' \rightsquigarrow_\beta^* [M'/x]M''$;
6. $[M''/x]M \rightsquigarrow_\beta^* [M''/x]M'$.

*Proof.* These follow from consideration of the untyped $\lambda$-calculus: for example, substitution into a $\beta$-redex leaves it as a $\beta$-redex. ∎

**Lemma 16.** *For pure terms $M, M', Ms$ with $M \rightsquigarrow_\beta M'$ we have $\{M\}Ms \rightsquigarrow_\beta \{M'\}Ms$.*

*Proof.* By induction on the definition of $\{.\}$.. The essential idea is that any $\beta$-redex in $M$ is still a $\beta$-redex in $\{M\}Ms$, even though $M$ may well not be a subterm of $\{M\}Ms$. ∎

## 9 Adding $B$-reduction

We will show that the addition of the $B$-rule upsets neither confluence nor, provided we stick at least to (e.g.) typed terms, termination. A key ingredient in both of these proofs is the Projection Lemma, i.e. that a root $B$-reduction translates (under purification) to exactly one $\beta$-reduction.

**Lemma 17.** $\overline{cut_3(\lambda x.M, (M' :: Ms))} \rightsquigarrow_\beta \overline{cut_3(cut_4(M', x.M), Ms)}$.

*Proof.* The $LHS = cut_3(\lambda x.\overline{M}, (\overline{M'} :: \overline{Ms}))$, the latter (as a $B$-redex) being pure; this $B$-reduces to $cut_3(cut_4(\overline{M'}, x.\overline{M}), \overline{Ms})$ and thus $\beta$-reduces to $\overline{cut_3(cut_4(\overline{M'}, x.\overline{M}), \overline{Ms})}$. By Lemma 8, this equals the $RHS$. ∎

**Lemma 18 (Projection Lemma).** *If $M \rightsquigarrow_B M'$ at the root position, then $\overline{M} \rightsquigarrow_\beta \overline{M'}$.*

*Proof.* Apart from the names of variables, this is just a restatement of Lemma 17. ∎

**Corollary 19.** *For terms $M, M', Ms$, we have (if the $RHS < \infty$)*

$$\|cut_3(\lambda x.M, (M' :: Ms))\| > \|cut_3(cut_4(M', x.M), Ms)\|.$$ ∎

We also need to know that an arbitrary $B$-reduction translates, after purification, to zero or more $\beta$-reductions.

**Proposition 20.** *The following hold:*

1. *If $Ms \leadsto_B Ms'$, then $\overline{Ms} \leadsto_\beta^* \overline{Ms'}$;*
2. *If $M \leadsto_B M'$, then $\overline{M} \leadsto_\beta^* \overline{M'}$.*

*Proof.* By simultaneous inductions on the size of $Ms$ or $M$, and case analysis:

1. (a) $Ms = []$: trivial;
   (b) $Ms = (M :: Ms'')$: routine use of inductive hypothesis;
   (c) $Ms = cut_1(Ms'', Ms''')$: similar to the first two parts of case 2(c) below.
   (d) $Ms = cut_2(M'', x.Ms''')$: similar to the case 2(d) below.
2. (a) $M = \lambda x.M''$: routine;
   (b) $M = (x; Ms'')$: routine;
   (c) $M = cut_3(M'', Ms'')$: there are three cases:
      i. the $B$-reduction is of $M''$ to $M'''$: by inductive hypothesis, $\overline{M''} \leadsto_\beta^* \overline{M'''}$, whence, by Lemma 15 (1), $\{\overline{M''}\}\overline{Ms''} \leadsto_\beta^* \{\overline{M'''}\}\overline{Ms''}$. Now,

$$\overline{cut_3(M'', Ms'')} = \overline{cut_3(\overline{M''}, \overline{Ms''})} = \{\overline{M''}\}\overline{Ms''}$$

      by Lemma 8 and Proposition 9 (3) respectively; and similarly

$$\overline{cut_3(M''', Ms'')} = \overline{cut_3(\overline{M'''}, \overline{Ms''})} = \{\overline{M'''}\}\overline{Ms''}$$

      whence $\overline{M} \leadsto_\beta^* \overline{M'}$.
      ii. the $B$-reduction is of $Ms''$ to $Ms'''$: similar, using Lemma 15 (2).
      iii. the $B$-reduction is at the root of $M$: we use the Projection Lemma.
   (d) $M = cut_4(M'', x.M''')$: there are two cases, dealt with as in (c), using Lemma 15 (5) and (6). ∎

It is now easy to show that the system $ES + CC + B$ is confluent, using the confluence of $\beta$-reduction in the $\lambda$-calculus.

**Theorem 21.** *The system $ES + CC + B$ is confluent.*

*Proof.* Suppose that $M \leadsto_{ES+CC+B}^* M_1$ and $M \leadsto_{ES+CC+B}^* M_2$. Then, by Lemma 8 and Proposition 20, both $\overline{M} \leadsto_\beta^* \overline{M_1}$ and $\overline{M} \leadsto_\beta^* \overline{M_2}$, whence, by confluence of $\beta$-reduction in the $\lambda$-calculus, for some (pure) term $M^\circ$ we have $\overline{M_1} \leadsto_\beta^* M^\circ$ and $\overline{M_2} \leadsto_\beta^* M^\circ$. But then, both $M_1 \leadsto_{ES+CC+B}^* M^\circ$ and also $M_2 \leadsto_{ES+CC+B}^* M^\circ$. (Similarly for $Ms$.) ∎

## 10 Strong Normalisation

**Definition 22 (Bounded terms).** *A term $M$ (or $Ms$) is* bounded *iff for every subterm $M'$ or $Ms'$ thereof, $\|M'\| < \infty$ (resp. $\|Ms'\| < \infty$).*

**Proposition 23 (Boundedness).**

1. *Every typed term is bounded;*
2. *Every $(ES + CC + B)$-SN term is bounded;*
3. *For pure terms, $\beta$-SN is equivalent to "bounded".*

*Proof.* We consider the three parts in order:

1. Trivial, since every subterm of a typed term is typed and the purification of a typed term is typed, and (by Proposition 12) every pure typed term is $\beta$-SN.
2. Consider a subterm $M$ of an $(ES + CC + B)$-SN term; it also is $(ES + CC + B)$-SN and so is its purification $\overline{M}$. But then any infinite sequence of $\beta$-reductions from $\overline{M}$ can be seen, by Corollary 10, as a sequence of $(ES + CC + B)$-reductions, including infinitely many $B$-reductions. (Similarly for subterms $Ms$.)
3. Pure $\beta$-SN terms are bounded, since, for every subterm $M$ of such a term, $M$ is pure and $\beta$-SN; so $\overline{M}$ $(= M)$ is $\beta$-SN (and similarly for subterms $Ms$). The converse is even more trivial. ■

Our aim now is to prove Theorem 31, i.e. the converse of part (2) of Proposition 23.

**Lemma 24.** *For all terms $M, M', Ms, Ms'$:*

1. *If $M \leadsto_{ES+CC} M'$ then $\| M \| = \| M' \|$;*
2. *If $Ms \leadsto_{ES+CC} Ms'$ then $\| Ms \| = \| Ms' \|$;*
3. *If $M \leadsto_{B} M'$ then $\| M \| \geq \| M' \|$;*
4. *If $Ms \leadsto_{B} Ms'$ then $\| Ms \| \geq \| Ms' \|$.*

*Proof.* The first part is trivial, since $\overline{M} = \overline{M'}$; the second part is similar. The other two parts use Proposition 20. ■

**Definition 25 (Cosy occurrence; cosy embedding).** *An occurrence of a proper subterm in a term is* cosy *iff no $cut_2$ or $cut_4$ constructors intervene on the path to the root, apart from a possible occurrence at the subterm itself. A term is* cosily embedded *in a term iff it has a cosy occurrence in the latter term.*

For example, in a term of the form $cut_1(cut_2(M, x.Ms), cut_2(M, x.Ms'))$, the two $cut_2$ subterms are the only cosily embedded proper subterms; and in a term of the form $cut_2(M, x.Ms)$, no proper subterms are cosily embedded.

**Lemma 26.** *For all terms $M, Ms, Ms'$:*

1. *(a)* $\| (M :: Ms) \| \geq \| M \|$;
   *(b)* $\| (M :: Ms) \| \geq \| Ms \|$;
2. $\| (x; Ms) \| = \| Ms \|$;
3. $\| \lambda x.M \| = \| M \|$;
4. *(a)* $\| cut_1(Ms, Ms') \| \geq \| Ms \|$;
   *(b)* $\| cut_1(Ms, Ms') \| \geq \| Ms' \|$;
5. $\| cut_3(M, Ms) \| \geq \| M \|$;
6. $\| cut_3(M, Ms) \| \geq \| Ms \|$.

*Proof.* 1. Because $\overline{(M :: Ms)} = (\overline{M} :: \overline{Ms})$;
2. Because $\overline{(x; Ms)} = (x; \overline{Ms})$;
3. Because $\overline{\lambda x.M} = \lambda x.\overline{M}$;
4. Because, by Proposition 9 (1), $\overline{cut_1(Ms, Ms')} = \overline{Ms} @ \overline{Ms'}$;
5. Since $\overline{cut_3(M, Ms)} = cut_3(\overline{M}, \overline{Ms})$ and $\| M \| = \| \overline{M} \|$, we may, without loss of generality, assume that the terms $M$ and $Ms$ are pure; we now just appeal to Lemma 16.
6. Without loss of generality, for the same reason as in (5), $M$ and $Ms$ may be assumed to be pure. If $Ms = []$, then the result is trivial. Otherwise, we argue by induction on the size of $M$ and case analysis. Consider the possible forms of $M$:
   (a) $M = (x; Ms'')$, whence by rule 3(a) it suffices to observe that $\| (x; Ms'' @ Ms) \| = \| Ms'' \| + \| Ms \|$;
   (b) $M = \lambda x.M'$; so $cut_3(M, Ms)$ is pure and has $Ms$ as a sub-term, whence $\| cut_3(M, Ms) \| \geq \| Ms \|$;

(c) $M = cut_3(M', Ms'')$; so, by rule 5(c), $\overline{cut_3(M, Ms)} = \overline{cut_3(M', Ms''@Ms)}$ and, by inductive hypothesis, $\| cut_3(M', Ms''@Ms) \| \geq \| Ms''@Ms \| \geq \| Ms \|$. ■

**Corollary 27.** *If a term $M$ is cosily embedded in $M'$, then $\| M \| \leq \| M' \|$, and similarly for other combinations such as $M$ cosily embedded in $Ms$, etc.*

*Proof.* By the lemma, using induction on the number of constructors between the subterm and the term. ■

On the other hand, whenever $\| M \| > 0$, we have

$$\| M \| \not\leq \| cut_2(M, x.[]) \| = 0$$

and

$$\| M \| \not\leq \| cut_4(M, x.\lambda y.(y; [])) \| = 0.$$

**Corollary 28.** *We have the following:*

1. *For each of the $(ES + CC)$-reduction rules $L \rightsquigarrow R$, and for each non-variable subterm $S$ of $R$, and instantiation $\theta$ of the variables in the rule, we have $\| L^\theta \| \geq \| S^\theta \|$;*
2. *For the B-reduction rule $L \rightsquigarrow R$, and for each non-variable subterm $S$ of $R$, and instantiation $\theta$ of the variables in the rule, we have $\| L^\theta \| \geq \| S^\theta \|$, the inequality being strict if either side is finite.*

*Proof.* 1. By Lemma 24 (1 or 2), we have $\| L^\theta \| = \| R^\theta \|$. It now suffices to note that, for each rule $L \rightsquigarrow R$, each non-variable proper subterm of $R$ is cosily embedded.
2. By Lemma 18, assuming $\| L^\theta \| < \infty$, we have $\| L^\theta \| > \| R^\theta \|$; as before, the only non-variable proper sub-term $S$ of $R$ is cosily embedded. ■

This corollary is the crux of the present paper: it gives us information about the (ES+CC+B)-rules that will be exploited when we show that the rules are decreasing w.r.t. a suitably chosen ordering. Note that the above corollary tells us just about reductions at the root position; for reductions at non-root positions, it says nothing.

**Corollary 29.** *Bounded terms are closed under $(ES + CC + B)$-reduction.*

*Proof.* Let $M$ be bounded and let $R$ be a rule in (ES+CC+B) such that $M \rightsquigarrow_R M'$. Consider a subterm $M''$ of $M'$; we must show that $\| M'' \| < \infty$. Comparing the position of $M''$ in $M'$ to that of the reduct, we find three cases:

1. The reduct occurs as a subterm of $M''$; so we can pull $M''$ back to a subterm $M^*$ of $M$, with $M^* \rightsquigarrow M''$. Since $M$ is bounded, $\| M^* \| < \infty$. By Lemma 24, $\| M'' \| < \infty$.
2. The reduct has $M''$ as a proper subterm, and $M''$ is obtained by instantiation of a variable in the rule $R$; thus already $M''$ is a subterm of $M$, which is bounded, so $\| M'' \| < \infty$.
3. The reduct has $M''$ as a proper subterm, and $M''$ is obtained by instantiation of a non-variable subterm of the RHS of the rule $R$; by Corollary 28, $\| M'' \| \leq \| M^* \|$ for some term $M^*$ which is in fact a subterm of $M$, so $\| M'' \| < \infty$. ■

We now consider the bounded cut terms superfixed not, as before, with types but with, for each such term $M$, the natural number $\| M \|$ (and similarly for terms $Ms$). We again order the cut operators by $cut^n > cut^m$ for $n > m$, use the suffices $(4 = 2 > 3 = 1)$ as before for ordering cut operators with the same superfix, and order all cut operators as greater than all non-cut operators. There are now infinitely many operators; but for a given bounded term, with only finitely many cut sub-terms, we can compute an upper bound for all their superfixes; by Corollary 28, this bound suffices for all terms reachable from the term by any of the reduction rules, so we can w.l.o.g. assume that our signature is finite, as required for use of the fact that the lexicographic path ordering generated below is a simplification ordering and thus is well-founded.

From this ordering on this (finite) signature we generate the lexicographic path ordering "$>_{LPO}$" on all terms. As in [2], we can avoid the problem of the LPO techniques not being applicable to higher-order systems by translation into a terminating (but non-confluent) intermediate system where the bound variables are omitted. However in order not to obscure our argument, we will not give this translation, but rather apply the LPO techniques directly to our higher-order system.

**Proposition 30.** *If $M'$ is a bounded term and $M' \leadsto_{EC+CC+B} M''$, then $M' >_{LPO} M''$. (Similarly for Ms.)*

*Proof.* It suffices to consider only reductions at root position, since $>_{LPO}$ is closed under contextual closure. In fact, the previous proof (of Proposition 5) for $(ES+CC)$ now works almost unchanged. We consider the rule $B$ in order to illustrate the method: let

$$M' = cut_3^m(\lambda x.M, (M''' :: Ms)) \leadsto_B cut_3^r(cut_4^s(M''', x.M), Ms) = M''$$

be an instance of rule $B$. By Corollary 19, we have $m > r$; similarly $m > s$ by this and by Corollary 28 (2). Then, since $m > r$, we just need to compare $M'$ with the two main subterms of $M''$. That $M' >_{LPO} cut_4^s(M''', x.M)$ follows because $m > s$ and $M''', M$ are variables properly occurring in $M'$; that $M' >_{LPO} Ms$ is trivial, the latter being a variable properly occurring in $M'$. It follows that $M' >_{LPO} M''$. ■

**Theorem 31.** *Every bounded term is $(ES + CC + B)$-SN.*

*Proof.* By the well-foundedness of $>_{LPO}$ and Proposition 30. ■

**Corollary 32.** *Every typed term is $(ES + CC + B)$-SN.* ■

*Proof.* By Proposition 23 (1) and the above theorem. ■

(This answers Herbelin's question.)

**Corollary 33.** *The calculus of terms $Ms, M$ with the $(ES + CC + B)$-reduction rules preserves strong normalisation w.r.t. the calculus of pure terms under $\beta$-reduction.* ■

## 11 Comments

It would be interesting to find a direct normalisation proof (in sequent calculus notation) for the simply-typed $\lambda$-calculus and compare it with those of [10, 12].

Our cut-reductions 5(c), 5(d) for Herbelin's explicit substitution calculus already appeared, in different notation, as the rules $\overline{\lambda}22$ and $\overline{\lambda}44$ in Espírito Santo's [7]; this paper *inter alia*

1. establishes a 1-1 correspondence, concerning both terms and reductions, between the lambda calculus and his calculus $\lambda_H$ of terms (similar to our calculus of pure terms);
2. shows that $\lambda_H$ can be extended to a calculus $\lambda_H^+$ of terms which, in our notation, have no instances of $cut_1$ and $cut_2$. These two operators are treated as defined functions; our category of terms $Ms$ can thus be replaced by that of term lists. This corresponds to a certain strategy of reducing $cut_1$ and $cut_2$ terms by terms normal w.r.t. our rules 1 and 2; our rules 5(a) and 5(b) are then superfluous. Moreover, $cut_3$ and $cut_4$ terms are, following a reduction step, dealt with immediately by auxiliary functions, defined by equations, rather than by use of explicit operations.

However, the issue of whether $B$-reductions can be combined in an *arbitrary* fashion with $(ES + CC)$-reductions is not addressed; this appears to us to be the main issue raised by Herbelin's paper, with its emphasis on explicit concatenations and explicit substitutions. We gratefully acknowledge José Espírito Santo's helpful comments illuminating the content of his paper.

Vestergaard and Wells [14] have considered explicit substitution calculi based on Gentzen's L-systems, with de Bruijn indices rather than variable names and with "weak correspondences" with some known explicit substitution calculi.

Herbelin (private commmunication, March 2001) conjectured that our SN result for $(ES + CC + B)$ also follows from the SN result [9] for the $\overline{\lambda}\mu\overline{\mu}$-calculus, mentioning however that a similar conjecture for the strong normalisability of Parigot's $\lambda\mu$-calculus turned out to be mistaken. We have no opinion on this conjecture; in any case, it is good to have a more elementary proof.

An early version of this paper showed that "garbage reduction" rules (as in [2]) were admissible, as part of a (regrettably) faulty proof of Proposition 20. Such rules can be added as primitive rules without loss of confluence or termination.

The calculus of Herbelin also appears in the work of Cervesato and Pfenning [4], in the guise of a "spine calculus"; no theory of explicit substitutions therein appears to have been worked out, although some implementation of such substitutions is apparently in the Twelf implementation. We thank Iliano Cervesato for bringing this report to our attention.

There are of course issues in the explicit substitution world that are not addressed by the above, such as the questions of optimality, of sharing and of confluence on open terms. We make no claims about superiority of the system $ES + CC + B$ over other explicit substitution calculi; we remark merely that it has an impeccable proof-theoretic pedigree.

In Appendix D we show that our calculus can simulate the $\lambda\mathbf{x}$-calculus of [3] if we add another two simple rules (that can be added without losing confluence or termination). Issues arising with this simulation will be addressed in future work.

## References


1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
2. R. Bloo and H. Geuvers. Explicit Substitution: On the Edge of Strong Normalisation. *Theoretical Computer Science*, 211(1–2):375–395, 1999.
3. R. Bloo and K. H. Rose. Preservation of Strong Normalisation in Named Lambda Calculi with Explicit Substitution and Garbage Collection. In Proceedings of CSN'95, 1995, pp 62–72.
4. I. Cervesato and F. Pfenning. A Linear Spine Calculus. School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., CMU-CS-97-125, 1997.
5. R. Dyckhoff and L. Pinto. Proof Search in Constructive Logic. In S. B. Cooper and J. K. Truss, editors, *Proceedings of the Logic Colloquium 1997*, volume 258 of *London Mathematical Society Lecture Note Series*, pages 53–65. Cambridge University Press, 1997.
6. R. Dyckhoff and L. Pinto. Cut-Elimination and a Permutation-Free Sequent Calculus for Intuitionistic Logic. *Studia Logica*, 60(1):107–118, 1998.
7. J. C. Espírito Santo. Revisiting the Correspondence between Cut Elimination and Normalisation. In *Proceedings of ICALP 2000*, volume 1853 of *LNCS*, pages 600–611. Springer Verlag, 2000.
8. H. Herbelin. A $\lambda$-calculus Structure Isomorphic to Sequent Calculus Structure. In *Proceedings of the 1994 conference on Computer Science Logic*, volume 933 of *LNCS*, pages 67–75. Springer Verlag, 1995.
9. H. Herbelin. Explicit Substitutions and Reducibility. Journal of Logic and Computation, Vol 11 (3), pages 429–449, 2001.
10. F. Joachimski and R. Matthes. Short Proofs of Normalisation. Archive for Mathematical Logic, to appear. (Preprint, 1999.)
11. K. H. Rose. Explicit Substitution: Tutorial & Survey. Technical report, BRICS, Department of Computer Science, University of Aarhus, 1996.
12. F. van Raamsdonk and P. Severi. On Normalisation. Technical report, TR CS-R9545, Centrum voor Wiskunde en Informatica, Amsterdam, 1996. (incorporated into [13].)
13. F. van Raamsdonk, P. Severi, M. H. Sörensen and H. Xi. Perpetual Reductions in Lambda Calculus. Information and Computation **149**, pages 173–225, 1999.
14. R. Vestergaard and J. Wells. Cut Rules and Explicit Substitutions. Mathematical Structures in Computer Science, to appear.

## A Proof of Proposition 5

We define a function $h : (Ms \cup M) \longrightarrow \mathbb{N}$ as follows:

$$\begin{aligned} h([]) &= 1 \\ h((M :: Ms)) &= h(M) + h(Ms) + 1 \\ h(cut_1(Ms, Ms')) &= h(Ms') + 2 * h(Ms) + 1 \\ h(cut_2(M, x.Ms)) &= h(Ms) * (3 * h(M) + 1) \end{aligned}$$

$$\begin{aligned} h((x; Ms)) &= h(Ms) + 1 \\ h((\lambda x.M)) &= h(M) + 1 \\ h(cut_3(M, Ms)) &= h(Ms) + 2 * h(M) + 1 \\ h(cut_4(M, x.M')) &= h(M') * (3 * h(M) + 1) \end{aligned}$$

and observe that for every rule of $ES + CC$, $h(L) > h(R)$.

## B Proof of Proposition 6

Here we show systematically that each critical pair is joinable, considering pairs of rules $R_1, R_2$ where the LHS of $R_1$ unifies with a non-variable subterm of the LHS of $R_2$, in which case we say that $R_1$ *overlaps* with $R_2$.

1. Rule 1(a) overlaps with 5(a): thus,

   $$cut_1^A(cut_1^B([], Ms), Ms')$$

   reduces by 1(a) to

   $$cut_1^A(Ms, Ms');$$

   we can also reduce it by 5(a) to

   $$cut_1^B([], cut_1^A(Ms, Ms'))$$

   which reduces by 1(a) to

   $$cut_1^A(Ms, Ms').$$

2. Rule 1(a) overlaps with 5(b): thus,

   $$cut_2^A(M, x.cut_1^B([], Ms))$$

   reduces by 1(a) to

   $$cut_2^A(M, x.Ms);$$

   we can also reduce it by 5(b) to

   $$cut_1^B(cut_2^A(M, x.[]), cut_2^A(M, x.Ms))$$

   which reduces by 2(a) to

   $$cut_1^B([], cut_2^A(M, x.Ms))$$

   which reduces by 1(a) to

   $$cut_2^A(M, x.Ms).$$

3. Rule 1(b) overlaps with 5(a): thus,

$$cut_1^A(cut_1^B((M :: Ms), Ms'), Ms'')$$

reduces by 1(b) to

$$cut_1^A((M :: cut_1^B(Ms, Ms')), Ms'')$$

which reduces by 1(b) to

$$(M :: cut_1^A(cut_1^B(Ms, Ms'), Ms''))$$

which reduces by 5(a) to

$$(M :: cut_1^B(Ms, cut_1^A(Ms', Ms'')));$$

we can also reduce it by 5(a) to

$$cut_1^B((M :: Ms), cut_1^A(Ms', Ms''))$$

which reduces by 1(b) to

$$(M :: cut_1^B(Ms, cut_1^A(Ms', Ms''))).$$

4. Rule 1(b) overlaps with 5(b): thus,

$$cut_2^A(M, x.cut_1^B((M' :: Ms), Ms'))$$

reduces by 1(b) to

$$cut_2^A(M, x.(M' :: cut_1^B(Ms, Ms')))$$

which reduces by 2(b) to

$$(cut_4^A(M, x.M') :: cut_2^A(M, x.cut_1^B(Ms, Ms'))$$

which reduces by 5(b) to

$$(cut_4^A(M, x.M') :: cut_1^B(cut_2^A(M, x.Ms), cut_2^A(M, x.Ms')));$$

we can also reduce it by 5(b) to

$$cut_1^B(cut_2^A(M, x.(M' :: Ms)), cut_2^A(M, x.Ms'))$$

which reduces by 2(b) to

$$cut_1^B((cut_4^A(M, x.M') :: cut_2^A(M, x.Ms)), cut_2^A(M, x.Ms'))$$

which reduces by 1(b) to

$$(cut_4^A(M, x.M') :: cut_1^B(cut_2^A(M, x.Ms), cut_2^A(M, x.Ms'))).$$

5. Rules 2(a), 2(b) have no overlaps.
6. Rule 3(a) overlaps with 5(c): thus,

$$cut_3^A(cut_3^B((x; Ms), Ms'), Ms'')$$

reduces by 3(a) to

$$cut_3^A((x; cut_1^B(Ms, Ms')), Ms'')$$

which reduces by 3(a) to

$$(x; cut_1^A(cut_1^B(Ms, Ms'), Ms''))$$

which reduces by 5(a) to

$$(x; cut_1^B(Ms, cut_1^A(Ms', Ms'')));$$

we can also reduce it by 5(c) to

$$cut_3^B((x; Ms), cut_1^A(Ms', Ms''))$$

which reduces by 3(a) to

$$(x; cut_1^B(Ms, cut_1^A(Ms', Ms''))).$$

7. Rule 3(a) overlaps with 5(d). Consider two cases:
   (a)
   $$cut_4^A(M, x.cut_3^B((x; Ms), Ms'))$$
   reduces by 3(a) to
   $$cut_4^A(M, x.(x; cut_1^B(Ms, Ms')))$$
   which reduces by 4(b) to
   $$cut_3^A(M, cut_2^A(M, x.cut_1^B(Ms, Ms')))$$
   which reduces by 5(b) to
   $$cut_3^A(M, cut_1^B(cut_2^A(M, x.Ms), cut_2^A(M, x.Ms')));$$
   we can also reduce it by 5(d) to
   $$cut_3^B(cut_4^A(M, x.(x; Ms)), cut_2^A(M, x.Ms'))$$
   which reduces by 4(b) to
   $$cut_3^B(cut_3^A(M, cut_2^A(M, x.Ms)), cut_2^A(M, x.Ms'))$$
   which reduces by 5(c) to
   $$cut_3^A(M, cut_1^B(cut_2^A(M, x.Ms), cut_2^A(M, x.Ms'))).$$
   (b) Let $y \neq x$.
   $$cut_4^A(M, x.cut_3^B((y; Ms), Ms'))$$
   reduces by 3(a) to
   $$cut_4^A(M, x.(y; cut_1^B(Ms, Ms')))$$
   which reduces by 4(a) to
   $$(y; cut_2^A(M, x.cut_1^B(Ms, Ms')))$$
   which reduces by 5(b) to
   $$(y; cut_1^B(cut_2^A(M, x.Ms), cut_2^A(M, x.Ms')));$$
   we can also reduce it by 5(d) to
   $$cut_3^B(cut_4^A(M, x.(y; Ms)), cut_2^A(M, x.Ms'))$$
   which reduces by 4(a) to
   $$cut_3^B((y; cut_2^A(M, x.Ms)), cut_2^A(M, x.Ms'))$$
   which reduces by 3(a) to
   $$(y; cut_1^B(cut_2^A(M, x.Ms), cut_2^A(M, x.Ms'))).$$
8. Rule 3(b) overlaps with 5(c): thus,
   $$cut_3^A(cut_3^B(\lambda x.M, []), Ms)$$
   reduces by 3(b) to
   $$cut_3^A(\lambda x.M, Ms);$$
   we can also reduce it by 5(c) to
   $$cut_3^B(\lambda x.M, cut_1^A([], Ms))$$
   which reduces by 1(a) to
   $$cut_3^B(\lambda x.M, Ms).$$
   Note the apparent change of type information; if we are ignoring types, this is no problem; and if terms are typed, then, in this case, because of the [] argument, $A = B$.

9. Rule 3(b) overlaps with 5(d): thus,

$$cut_4^A(\lambda y.M, x.cut_3^B(\lambda z.M', []))$$

reduces by 3(b) to

$$cut_4^A(\lambda y.M, x.\lambda z.M')$$

which reduces by 4(c) to

$$\lambda z.cut_4^A(\lambda y.M, x.M');$$

we can also reduce it by 5(d) to

$$cut_3^B(cut_4^A(\lambda y.M, x.\lambda z.M'), cut_2^A(\lambda y.M, x.[]))$$

which reduces by 2(a) to

$$cut_3^B(cut_4^A(\lambda y.M, x.\lambda z.M'), [])$$

which reduces by 4(c) to

$$cut_3^B(\lambda z.cut_4^A(\lambda y.M, x.M'), [])$$

which reduces by 3(b) to

$$\lambda z.cut_4^A(\lambda y.M, x.M').$$

10. Rules 4(a), 4(b), 4(c) and 4(d) have no overlaps.
11. Rule 5(a) overlaps with itself. Consider

$$cut_1^A(cut_1^B(cut_1^C(Ms, Ms'), Ms''), Ms''')$$

which reduces by 5(a) at a non-root position to

$$cut_1^A(cut_1^C(Ms, cut_1^B(Ms', Ms'')), Ms''')$$

which reduces by 5(a) again to

$$cut_1^C(Ms, cut_1^A(cut_1^B(Ms', Ms''), Ms'''))$$

which reduces by 5(a) again to

$$cut_1^C(Ms, cut_1^B(Ms', cut_1^A(Ms'', Ms'''))).$$

Reduction by 5(a) at the root position, however, produces

$$cut_1^B(cut_1^C(Ms, Ms'), cut_1^A(Ms'', Ms'''))$$

which reduces by 5(a) to

$$cut_1^C(Ms, cut_1^B(Ms', cut_1^A(Ms'', Ms'''))).$$

12. Rule 5(a) overlaps with 5(b). Consider

$$cut_2^A(M, x.cut_1^B(cut_1^C(Ms, Ms'), Ms''))$$

which reduces by 5(a) to

$$cut_2^A(M, x.cut_1^C(Ms, cut_1^B(Ms', Ms'')))$$

which reduces by 5(b) to

$$cut_1^C(cut_2^A(M, x.Ms), cut_2^A(M, x.cut_1^B(Ms', Ms'')))$$

which reduces by 5(b) to

$$cut_1^C(cut_2^A(M, x.Ms), cut_1^B(cut_2^A(M, x.Ms'), cut_2^A(M, x.Ms'')));$$

we can also reduce it by 5(b) to

$$cut_1^B(cut_2^A(M, x.cut_1^C(Ms, Ms')), cut_2^A(M, x.Ms''))$$

which reduces by 5(b) to

$$cut_1^B(cut_1^C(cut_2^A(M, x.Ms), cut_2^A(M, x.Ms')), cut_2^A(M, x.Ms''))$$

which reduces by 5(a) to

$$cut_1^C(cut_2^A(M, x.Ms), cut_1^B(cut_2^A(M, x.Ms'), cut_2^A(M, x.Ms''))).$$

13. Rule 5(b) has no (further) overlaps.
14. Rule 5(c) overlaps with itself. Consider

$$cut_3^A(cut_3^B(cut_3^C(M, Ms), Ms'), Ms'')$$

which reduces by 5(c) at a non-root position to

$$cut_3^A(cut_3^C(M, cut_1^B(Ms, Ms')), Ms'')$$

which reduces by 5(c) to

$$cut_3^C(M, cut_1^A(cut_1^B(Ms, Ms'), Ms''))$$

which reduces by 5(a) to

$$cut_3^C(M, cut_1^B(Ms, cut_1^A(Ms', Ms''))).$$

But also, it reduces by 5(c) at the root position to

$$cut_3^B(cut_3^C(M, Ms), cut_1^A(Ms', Ms''))$$

and again by 5(c) to

$$cut_3^C(M, cut_1^B(Ms, cut_1^A(Ms', Ms''))).$$

15. Rule 5(c) overlaps with 5(d):

$$cut_4^A(M, x.cut_3^B(cut_3^C(M', Ms), Ms'))$$

reduces by 5(c) to

$$cut_4^A(M, x.cut_3^C(M', cut_1^B(Ms, Ms')))$$

which reduces by 5(d) to

$$cut_3^C(cut_4^A(M, x.M'), cut_2^A(M, x.cut_1^B(Ms, Ms')))$$

which reduces by 5(b) to

$$cut_3^C(cut_4^A(M, x.M'), cut_1^B(cut_2^A(M, x.Ms), cut_2^A(M, x.Ms')));$$

we can also reduce it by 5(d) to

$$cut_3^B(cut_4^A(M, x.cut_3^C(M', Ms)), cut_2^A(M, x.Ms'))$$

which reduces by 5(d) to

$$cut_3^B(cut_3^C(cut_4^A(M, x.M'), cut_2^A(M, x.Ms)), cut_2^A(M, x.Ms'))$$

which reduces by 5(c) to

$$cut_3^C(cut_4^A(M, x.M'), cut_1^B(cut_2^A(M, x.Ms), cut_2^A(M, x.Ms'))).$$

16. Rule 5(d) has no (further) overlaps.

## C Proof of Proposition 9

Note that in each case the term on the RHS is a pure term; by definition the functions @, [./.]. and {.}. all produce pure terms from pure arguments. The proposition follows from the following cases formulated as lemmata.

**Lemma 34.** *For pure terms $Ms, Ms'$,*

$$cut_1(Ms, Ms') \leadsto^*_{ES+CC} Ms@Ms'.$$

*Proof.* Routine, by induction on the length of $Ms$ which, being pure, is already known to be a list. ■

**Lemma 35.** *For pure terms $M, Ms$,*

$$cut_3(M, Ms) \leadsto^*_{ES+CC} \{M\}Ms.$$

*Proof.* By induction on the height of the term $M$ and case analysis:

1. When $M$ is of the form $(y; Ms')$ the LHS is

   $$cut_3((y; Ms'), Ms)$$

   which reduces by 3(a) to $(y; cut_1(Ms', Ms))$, which reduces by Lemma 34 to $(y; Ms'@Ms)$, which is just $\{(y; Ms')\}Ms$.
2. When $M$ is of the form $\lambda x.M'$, there are two cases:
   (a) When $Ms = []$: this is easy.
   (b) When $Ms = (M'' :: Ms')$, the LHS of (3) is $cut_3(\lambda x.M', (M'' :: Ms'))$ which is (by definition) the same term as $\{\lambda x.M'\}Ms$ without any reduction.
3. When $M$ is of the form $cut_3(\lambda y.M', M'' :: Ms')$, the LHS of (3) is

   $$cut_3(cut_3(\lambda y.M', M'' :: Ms'), Ms)$$

   which reduces by 5(c) to

   $$cut_3(\lambda y.M', cut_1(M'' :: Ms', Ms))$$

   which reduces by 1(b) to

   $$cut_3(\lambda y.M', (M'' :: cut_1(Ms', Ms))).$$

   which by Lemma 34 reduces to

   $$cut_3(\lambda y.M', M'' :: (Ms'@Ms)).$$

   which is just $\{M\}Ms$. ■

**Lemma 36.** *For pure terms $M, Ms, M'$,*

1. $cut_2(M, x.Ms) \leadsto^*_{ES+CC} [M/x]Ms$;
2. $cut_4(M, x.M') \leadsto^*_{ES+CC} [M/x]M'$.

*Proof.* By simultaneous induction on the heights of the terms and case analysis. We consider the cases systematically:

1. (a) When $Ms$ is $[]$, the LHS reduces by 2(a) to $[]$, which is just $[M/x][]$.
   (b) When $Ms$ is $(M' :: Ms')$, the LHS reduces by 2(b) to $cut_4(M, x.M') :: cut_2(M, x.Ms')$. The first part of this reduces, by inductive hypothesis, to $[M/x]M'$; the second reduces, by inductive hypothesis, to $[M/x]Ms'$; their combination by :: is just $[M/x](M' :: Ms')$, i.e. $[M/x]Ms$.

2. (a) When $M'$ is of the form $(y; Ms')$ (for $y \neq x$), the LHS reduces by 4(a) to

$$(y; cut_2(M, x.Ms'))$$

which by inductive hypothesis reduces to

$$(y; [M/x]Ms')$$

which is just

$$[M/x](y; Ms').$$

(b) When $M'$ is of the form $(x; Ms')$, the LHS reduces by 4(b) to

$$cut_3(M, cut_2(M, x.Ms'))$$

which by inductive hypothesis reduces to

$$cut_3(M, [M/x]Ms')$$

which by Lemma 35 reduces to

$$\{M\}[M/x]Ms'$$

which is just

$$[M/x](x; Ms').$$

(c) When $M'$ is of the form $\lambda y.M''$, this is easy.

(d) When $M'$ is of the form $cut_3(\lambda y.M'', (M''' :: Ms'))$, the LHS of (4) is

$$cut_4(M, x.cut_3(\lambda y.M'', (M''' :: Ms')))$$

which reduces by 5(d) to

$$cut_3(cut_4(M, x.\lambda y.M''), cut_2(M, x.(M''' :: Ms')))$$

which reduces by 4(c) and 2(b) to

$$cut_3(\lambda y.cut_4(M, x.M''), (cut_4(M, x.M''') :: cut_2(M, x.Ms')))$$

which by inductive hypothesis (three times) reduces to

$$cut_3(\lambda y.[M/x]M'', ([M/x]M''' :: [M/x]Ms'))$$

which is just

$$[M/x]cut_3(\lambda y.M'', (M''' :: Ms')).$$

## D Simulation of Lambda-x

In this section we show that the $\lambda\mathbf{x}$-calculus of Bloo and Rose [3] can be simulated by the reduction system $ES + CC + B$, if we add the following reduction rules

$$cut_1^A(M\,s, []) \leadsto Ms$$
$$cut_3^A(M, []) \leadsto M$$

These rules are harmless with respect to confluence and strong normalisation.

The terms of $\lambda\mathbf{x}$ are given by the following grammar:

$$N ::= x \mid (\lambda x.N) \mid NN \mid N\langle x := N\rangle.$$

In $\lambda\mathbf{x}$ the beta-reduction

$$\beta \qquad (\lambda x.N)N' \leadsto_\beta [N'/x]N \tag{1}$$

is replaced by the reduction

$$\mathtt{b} \qquad (\lambda x.N)N' \leadsto_\mathtt{b} N\langle x := N'\rangle \tag{2}$$

where the reduct contains the constructor for explicit substitutions. The following reduction rules apply to this term constructor.

$$\begin{array}{ll}
\mathbf{x1} & x\langle x := N\rangle \leadsto_{\mathbf{x1}} N \\
\mathbf{x2} & y\langle x := N\rangle \leadsto_{\mathbf{x2}} y \\
\mathbf{x3} & (\lambda y.N')\langle x := N\rangle \leadsto_{\mathbf{x3}} \lambda y.\, N'\langle x := N\rangle \\
\mathbf{x4} & (N'N'')\langle x := N\rangle \leadsto_{\mathbf{x4}} N'\langle x := N\rangle\, N''\langle x := N\rangle
\end{array}$$

We translate $\lambda\mathbf{x}$-terms into Herbelin's calculus as follows:

$$\begin{array}{rcl}
(x)^* & =_{def} & (x; []) \\
(\lambda x.N)^* & =_{def} & \lambda x.N^* \\
(NN')^* & =_{def} & cut_3(N^*, N'^* :: []) \\
(N\langle x := N'\rangle)^* & =_{def} & cut_4(N'^*, x.N^*)
\end{array}$$

The simulation is then as follows:

1. Rule $\mathtt{b}$ is mapped onto the reduction sequence

$$cut_3(\lambda x.N^*, N'^* :: []) \leadsto cut_3(cut_4(N'^*, x.N^*), []) \leadsto cut_4(N'^*, x.N^*)$$

2. Rule $\mathbf{x1}$ is mapped onto the reduction sequence

$$cut_4(N^*, x.(x; [])) \leadsto cut_3(N^*, cut_2(N^*, x.[])) \leadsto cut_3(N^*, []) \leadsto N^*$$

3. Rule $\mathbf{x2}$ is mapped onto the reduction sequence

$$cut_4(N^*, x.(y; [])) \leadsto (y; cut_2(N^*, x.[])) \leadsto (y; [])$$

4. Rule $\mathbf{x3}$ is mapped onto the reduction sequence

$$cut_4(N^*, x.\lambda y.N'^*) \leadsto \lambda y.cut_4(N^*, x.N'^*)$$

5. Rule $\mathbf{x4}$ is mapped onto the reduction sequence

$$\begin{array}{rcl}
cut_4(N^*, x.cut_3(N'^*, N''^* :: [])) & \leadsto & cut_3(cut_4(N^*, x.N'^*), cut_2(N^*, x.N''^* :: [])) \\
& \leadsto & cut_3(cut_4(N^*, x.N'^*), cut_4(N^*, x.N''^*) :: cut_2(N^*, x.[])) \\
& \leadsto & cut_3(cut_4(N^*, x.N'^*), cut_4(N^*, x.N''^*) :: [])
\end{array}$$

# Generalized Director Strings and Explicit Substitutions

Maribel Fernández
DI-LIENS (CNRS UMR 8548)
École Normale Supérieure
45 Rue d'Ulm, 75005 Paris, France
maribel@di.ens.fr

Ian Mackie
CNRS-LIX (UMR 7650)
École Polytechnique
91128 Palaiseau Cedex, France
mackie@lix.polytechnique.fr

**Abstract**

In this paper we give a name free $\lambda$-calculus with explicit substitutions. The calculus is based on a generalized notion of director strings: we annotate a term with information about how each substitution should be propagated through the term. Our calculus is based on a strategy for explicit substitutions (and thus we do not simulate $\beta$-reduction in full generality), but is nevertheless adequate for weak reduction (reduction to weak head normal form). Moreover we allow certain reductions to take place inside $\lambda$-abstractions which offers the potential of allowing additional sharing of redexes; thus we see this explicit substitution calculus as a contribution towards using such calculi as an implementation technique.

## 1 Introduction

In [5] we introduced a calculus of explicit substitutions, which implements *closed reduction* in the $\lambda$-calculus. This is a calculus with names, which follows a specific strategy for reduction defined by a set of conditional rewrite rules: substitutions must be closed before they can be moved (hence the name *closed reductions*). Although this calculus uses names, one of the surprising properties is that $\alpha$-conversion is not needed during reduction. At the end of [5] we also hinted at an alternative presentation of the calculus which, on one hand internalizes the conditions on the reduction rules, and on the other hand offers a name free version of the calculus (which is also syntactically simpler).

The purpose of this present paper is simply to present in detail the name free version of the calculus for closed reduction, and study its properties. We consider this important for three reasons:

- Part of the culture of explicit substitutions is name free.
- The calculus that we have obtained offers an alternative to de Bruijn notation [4], which has become the standard name-free syntax for such calculi. Since many calculi based on this notation lack certain properties, this motivates the search for alternative formalisms.
- In addition, we provide a generalization of director strings, which were introduced in [9] for combinator reduction. In our generalized director strings reduction is allowed under abstractions.

We thus see the calculus presented in this paper as a first step towards an alternative syntax for explicit substitution calculi.

Over the last few years a whole range of explicit substitution calculi have been proposed, starting from the $\lambda\sigma$-calculus [1], with the general aim of making the substitution process exist at the same level as $\beta$-reduction. The motivation for such calculi is to have a handle on the process of substitution, and to be able to control it in various ways. Although many different applications of such calculi exist, one of the main motivations, which we follow, is to express in low level terms the process of $\beta$-reduction. We are thus interested in the implementation perspective of such calculi. Specifically:

- Control the process of substitution: substitutions should not be duplicated unless we really need to. Thus we focus on strategies for the substitution process. We remark that the $\lambda$-calculus, in addition to substitution, lacks explicit information about sharing and evaluation orders. This point becomes more subtle when one considers it in the framework of explicit substitutions, since the order in which substitutions are performed can have dramatic consequences on the efficiency of the reduction process. To ensure that we have a tight control over the way substitutions are performed, we also make explicit the copying and erasing phases of substitution. This will also allow us to control (and avoid) the issues of duplicating and erasing free variables in the substitution process.

- The substitution process should be natural: we wish to avoid introducing a wealth of substitution manipulation rules (for instance building lists of substitutions) which have proved problematic.

- Efficiency will be one of our main considerations.

In particular, we point out that our goals are not to obtain a calculus for simulating $\beta$-reduction in full generality: many of the standard properties are indeed of very little interest to us, but nevertheless do hold for this calculus. One notable exception is confluence on open terms: our strategy for reduction is defined only on ground terms, and thus the application to higher-order unification, for instance, will not be possible in the present formulation of the calculus.

From an implementation perspective, weak explicit substitution calculi have been proposed. In the $\lambda$-calculus, by weak reduction we often mean no reduction under abstraction. In the language of explicit substitutions, this weak form of reduction is often interpreted as not pushing substitutions through an abstraction [3]. This form of weak reduction has the convenience that the most awkward part of the substitution process is removed from the system (prohibiting substitution through an abstraction avoids name clashes and $\alpha$-conversions). However this benefit is achieved at a price because terms with substitutions (closures) may be copied, which can cause redexes to be duplicated. In our calculus we address this problem by allowing closed substitutions to be made, and moreover we never copy a term (or closure) which contains a free variable.

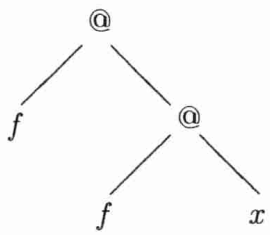
**Related work.** Our work is clearly related to the general work on explicit substitution calculi (starting from the $\lambda\sigma$-calculus [1]). However, is much more inline with the use of explicit substitutions for controlling substitutions in the $\lambda$-calculus, with an emphasis on implementation, for instance the call-by-need $\lambda$-calculus of Ariola et al. [2] and calculi with shared environments of Yoshida [13]. Other work in this area includes [8] and [12]. The notion of closed reduction for the $\lambda$-calculus was inspired by a strategy for cut-elimination in linear logic, used in a proof of soundness of the geometry of interaction by Girard [7]. Many of the ideas come from director strings [9], which we use as a starting point for this present work.

**Overview.** The rest of this paper is structured as follows. In the following section we provide the background material, specifically we recall director strings, and give intuitions about how we intend to generalize these. In Section 3 we introduce the calculus: syntax and reduction rules, together with some examples. In Section 4 we give a number of properties (confluence, preservation of strong normalization, etc.). In Section 5 we include a typed version of the calculus. Finally, in Section 6 we conclude the paper and state out current work in this area.

## 2 Background Material: Director Strings

In this section we briefly recall the basic ideas of director strings, which were introduced in [9] (see also [11]), and give the intuition to the generalization used in the rest of this paper.

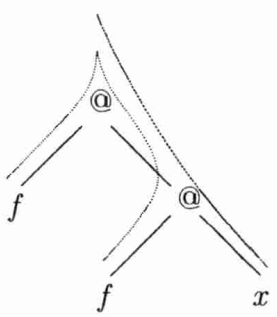
As an example, assume that we have a term $f(fx)$, which contains two free variables. We can draw this in the following way:



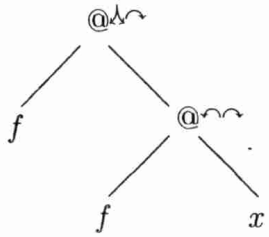

Assume also that we have substitutions for both $f$ and $x$, thus the term is $((f(fx))[F/f])[X/x]$. The best way to perform these substitutions would be to propagate them only to the place in the tree where they are required. For instance, for the $[F/f]$ substitution, we would like to have the following sequence of reductions:

$$\begin{array}{rcl}(f(fx))[F/f] & \to & (f[F/f])((fx)[F/f]) \\ & \to & (f[F/f])((f[F/f])x) \\ & \to & F((f[F/f])x) \\ & \to & F(Fx)\end{array}$$

Note that substitutions are copied only when they need to be. The reduction sequence for $[X/x]$ would similarly be directed to the unique occurrence of the variable $x$ in the term. In terms of graphs, we see this as each substitution being guided towards the destination, thus the diagram below shows the *paths* which the substitutions must follow.




There are several ways of guiding the substitution to the correct destination: for instance we can ask which are the free variables of each sub-term. However, a more natural way is given by director strings, which simply annotate each node in the graph with information about where the substitution must go. In the diagram below we show an annotated graph.




- When the substitution for $f$ passes the root of this term, we send a copy to both sub-terms, and erase the $\curlywedge$ director.
- The second substitution can then pass the root, where it is directed uniquely to the right branch.

This process is continued until substitutions reach their final destination, which are the variable occurrences. One of the reasons that this works is because substitutions do not overtake, that is to say that the order of the substitutions is preserved under reduction. This comment is important, and will play a role later in our explicit substitution calculus: substitutions will not be allowed to overtake each other.

There are several important remarks that must be made about this idea:

- This idea works easily when the substitution is closed (does not contain free variables). This is because as each substitution passes a given director, we must add the additional directors for each free variable in the substitution. If the term is closed, then we simply erase the director: the length of the director string indicates how many free variables there are. In this paper we adopt this latter choice, which additionally offers a good strategy for reduction.
- If we perform a $\beta$-reduction step, then we must make sure that we keep the path consistent: some reorganization of director strings must be made. However, there are several conditions that one can impose which keep these reorganizations local to the sub-term, and thus we do not require any global rewrite steps.
- Finally, we remark that we would like these reductions to take place anywhere in the term: in particular within substitutions and under abstractions.

We end this section by briefly recalling the relationship between director strings and combinator reduction. The reduction rules for the $S$,$B$,$C$ combinators are the following:

$$\begin{array}{lcl} Sxyz & \to & xz(yz) \\ Bxyz & \to & x(yz) \\ Cxyz & \to & xzy \end{array}$$

where $Sxy$ takes an argument and directs it to both $x$ and $y$; $Bxy$ takes an argument and directs it just to $y$; and finally $Cxy$ takes an argument and directs it just to $x$.

Thus we can annotate the $xy$ application with the combinators, which are just the directors given above ($S$ is $\curlywedge$, $B$ is $\curvearrowright$, and $C$ is $\curvearrowleft$).

# 3 An explicit substitution calculus

Here we present the main contribution of the paper, which is the explicit substitution calculus. We begin by giving the syntax of terms, then the reduction rules, and finally several examples. We conclude this section with a type system for the calculus.

**Definition 3.1 ($\lambda$-calculus with director string annotations)** *We define three syntactic categories:*

**Characters (Directors)** *: We use four directors:*

1. $\curvearrowright$*: indicates that the substitution occurs only in the right branch of a binary construct (application or substitution, as given below).*
2. $\curvearrowleft$*: indicates that the substitution occurs only in the left branch of a binary construct.*
3. $\curlywedge$*: indicates that the substitution occurs in both the left and right branches of a binary construct.*
4. $\downarrow$*: indicates that the substitution occurs in a unary construct (abstraction and variables, see below).*

*which will be the alphabet that we use to construct director strings. Additionally, we could include a fifth one "$-$" to indicate that the substitution is not required, but we prefer to encode this in the abstraction, as shown below.*

**Strings:** *A director string is either empty, denoted by $\epsilon$, or built from the above characters. We use $s, s_1, \ldots$ to range over strings, and the length of a string $s$ is denoted by $\mid s \mid$. We shall read strings from left to right.*

**Annotated Terms:** *Let $s$ range over strings, and $t, u$ range over annotated terms, then the following are valid terms:*

$$t ::= \Box \mid (\lambda t)^s \mid (\lambda^- t)^s \mid (tu)^s \mid (t[u])^s$$

*where $\Box$ represents variables (a place holder), $(\lambda t)^s$ is an abstraction, where the bound variable occurs in the term $t$, whereas $(\lambda^- t)^s$ is an abstraction where no variables are bound. $(tu)^s$ is an application, and finally $(t[u])^s$ is our notation for explicit substitution.*

Some intuitions are in order, many of which will become apparent when we give the reduction rules later.

- Variables, represented by $\Box$, are simply place holders in the term, which may be replaced by a term during reduction. The name of the variable is of no interest to us, since the director strings give the path that the substitution must follow through the term to ensure that it gets to the right place. Remark that variables are not annotated, since in practice the substitution is always required in the variable.
- There are two abstractions, depending on whether the abstracted variable occurs free in the body of the abstraction or not. There is an alternative presentation of this calculus which adds the "$-$" director which serves the same purpose. Combining the erasing with the abstraction indicates a choice in the calculus: we erase terms as soon as possible, which is a practical consideration, and similarly, we try to postpone duplication of a term to as late as possible.

As with most $\lambda$-calculi, we will adopt several syntactic conventions: we will drop parentheses whenever we can, and not write the empty string $\epsilon$ unless it is essential.

Our use of this calculus is rather as an object language: the image of a translation of correctly formed $\lambda$-terms, thus we shall not enter here into a possible set of conditions on when an annotated term is a valid one. The following definition of a compilation of the usual $\lambda$-calculus into this calculus also serves as an indication on how the strings are built.

**Definition 3.2 (Compilation)** *Let $M$ be a $\lambda$-term with $\mathsf{fv}(M) = \{x_1, \ldots, x_n\}$, its compilation is defined as: $[x_1]\ldots[x_n]M^\circ$ with $(\cdot)^\circ$ given by:*

$$\begin{array}{lcll} x^\circ & = & x & \\ (MN)^\circ & = & M^\circ N^\circ & \\ (\lambda x.M)^\circ & = & \lambda[x]M^\circ & \text{if } x \in \mathsf{fv}(M) \\ (\lambda x.M)^\circ & = & \lambda^- M^\circ & \text{otherwise} \end{array}$$

*and $[\cdot]\cdot$ given by:*

$$\begin{array}{lcll} [x]x & = & \Box & \\ [x](\lambda t)^s & = & (\lambda[x]t)^{\downarrow s} & \\ [x](\lambda^- t)^s & = & (\lambda^-[x]t)^{\downarrow s} & \\ [x](tu)^s & = & (([x]t)([x]u))^{\curlywedge s} & x \in \mathsf{fv}(t), x \in \mathsf{fv}(u) \\ & = & (([x]t)u)^{\curvearrowleft s} & x \in \mathsf{fv}(t), x \notin \mathsf{fv}(u) \\ & = & (t([x]u))^{\curvearrowright s} & x \in \mathsf{fv}(u), x \notin \mathsf{fv}(t) \end{array}$$

**Example 3.3** *We give several examples to give a flavour of terms in this calculus:*

$$\begin{array}{lcl} \mathbf{S} = (\lambda xyz.(xz)(yz))^\circ & = & \lambda(\lambda(\lambda((\Box\Box)^{\curvearrowleft\curvearrowleft}(\Box\Box)^{\curvearrowleft\curvearrowleft})^{\curvearrowleft\curvearrowleft\curlywedge})^{\downarrow\downarrow})^{\downarrow} \\ \mathbf{K} = (\lambda xy.x)^\circ & = & \lambda(\lambda^-\Box)^{\downarrow} \\ \mathbf{I} = (\lambda x.x)^\circ & = & \lambda\Box \\ \mathbf{2} = \lambda fx.f(fx)^\circ & = & \lambda(\lambda(\Box(\Box\Box)^{\curvearrowleft\curvearrowleft})^{\curlywedge\curvearrowleft})^{\downarrow} \end{array}$$

*We remark that in many cases, the graphical representation of these terms (as we gave in the previous section) helps to see the significance of the director strings.*

**Lemma 3.4 (Length of Strings)**

$$\mathsf{fv}(M) = \{x_1, \ldots, x_n\} \iff M^\circ = t^s, \mid s \mid = n$$

*In particular, note the case for closed terms, which always have the $\epsilon$ director string.*

We now proceed to give the reduction rules for this calculus. We begin with some intuitions.

- Pushing closed substitutions through a term has a limited effect on the director strings: we simply erase the leftmost director.
- A reduction in the $\lambda$-calculus changes the positions of the free variables, and thus changes some of the director strings in the redex sub-term. We thus want to restrict to reductions which only have a local effect (we will mention this choice again in the conclusions).

As a consequence of these remarks, many of our rules will only apply when the director string is empty.

**Definition 3.5** *The reduction rules for this calculus are given as follows, where if $d$ is a director, then $d^n$ is a string of $d$'s of length $n$:*

| Name | Reduction | | |
|---|---|---|---|
| Beta | $((\lambda t)^\epsilon u)^s$ | $\rightsquigarrow$ | $(t[u])^s$ |
| BetaErase | $((\lambda^- t)^\epsilon u^\epsilon)^\epsilon$ | $\rightsquigarrow$ | $t$ |
| Var | $(\Box[v^{s_1}])^s$ | $\rightsquigarrow$ | $v^{s_1}$ |
| App1 | $((t^{s_2}u)^{\curvearrowleft s_1}[v^\epsilon])^s$ | $\rightsquigarrow$ | $((t^{s_2}[v^\epsilon])^{\curvearrowleft\lvert s_2\rvert}u)^{s_1}$ |
| App2 | $((tu^{s_2})^{\curvearrowright s_1}[v^\epsilon])^s$ | $\rightsquigarrow$ | $(t(u^{s_2}[v^\epsilon])^{\curvearrowleft\lvert s_2\rvert})^{s_1}$ |
| App3 | $((t^{s_2}u^{s_3})^{\curlywedge s_1}[v^\epsilon])^s$ | $\rightsquigarrow$ | $((t^{s_2}[v^\epsilon])^{\curvearrowleft\lvert s_2\rvert}(u^{s_3}[v^\epsilon])^{\curvearrowleft\lvert s_3\rvert})^{s_1}$ |
| Lam | $((\lambda t)^{\downarrow s_1}[v^\epsilon])^s$ | $\rightsquigarrow$ | $(\lambda(t[v^\epsilon])^{\curvearrowleft\lvert s_1\rvert})^{s_1}$ |
| Comp | $((t[w^{s_2}])^{\curvearrowright s_1}[v^\epsilon])^s$ | $\rightsquigarrow$ | $(t[(w^{s_2}[v^\epsilon])^{\curvearrowleft\lvert s_2\rvert}])^{s_1}$ |

Again, some intuitions are in order.

- The leading theme of the reduction is that of closed reduction: only closed substitutions can be propagated through a term, which is captured in this calculus by the fact that the director string is empty for each substitution. This means that we do not have to update many of the strings during reduction.
- Substitution through an abstraction is permitted, but only when the substitution is closed. Remark that in terms of a calculus with names, no $\alpha$-conversion would be needed. It is the inclusion of this rule in the calculus which allows us to have a calculus which is less weak than many other weak $\lambda$-calculi for explicit substitutions.
- The *Beta* rule requires that the function is closed: this places a priority on pushing substitutions.
- The *BetaErase* rule is simply a *Beta* rule where we erase the argument if the abstraction does not bind any variable. Remark that this rule only erases closed terms, and thus we (from a named calculus perspective) preserve free variables under reduction.
- The rules for application are the main rules which use the director strings: substitutions here are propagated only to the places where they are needed, and copied only when there is more than one occurrence of a given variable.
- The *Comp* rule allows substitutions to move inside others, but note that they do not overtake.

**Example 3.6** *We give one example reduction sequence for this calculus, which demonstrates that we can reduce under abstractions. Consider the $\lambda$-term $\lambda x.(\lambda y.y)x$, which contains a single redex.*

$$(\lambda x.(\lambda y.y)x)^\circ = (\lambda((\lambda\Box)^\epsilon\Box)^\frown)^\epsilon \rightsquigarrow \lambda(\Box[\Box])^\frown \rightsquigarrow \lambda\Box = (\lambda x.x)^\circ$$

*We remark that an encoding into combinators, using director strings as presented in [9], would not allow this redex to be contracted, and thus if used as an argument could potentially be duplicated. In this sense, our director strings are a generalization of [9].*

Since we prefer to think of this calculus as some form of intermediate language, we also provide a notion of read-back, which simply puts the names back in, and completes the substitution process.

**Definition 3.7 (Readback)** *Let $t^s$ be a term, where $\mid s \mid = n$. We extend the syntax of our calculus with ground constants $x_1, \ldots, x_n$ representing the variables, and allow these in the rewrite rules.*

*Define $(t^s)^\star$ as follows (we omit the strings since they do not play any role in the definition):*

$$\begin{array}{lcll}
(\Box)^\star & = & \Box & \\
(\lambda t)^\star & = & \lambda x.(t)^\star[x] & x \text{ fresh} \\
(\lambda^- t)^\star & = & \lambda x.(t)^\star & \\
(tu)^\star & = & (t)^\star(u)^\star & \\
(t[u])^\star = (t)^\star[(u)^\star] & & &
\end{array}$$

*The term $(t^s)^\star[x_1]\cdots[x_n]$ can then be reduced to normal form with the rewrite rules given in Definition 3.5, to obtain a $\lambda$-term.*

**Example 3.8** *We give one small example of the readback procedure:*

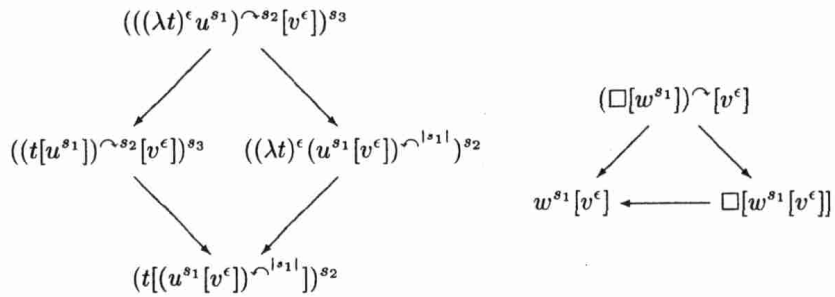
$$(\lambda\Box)^\star = \lambda x.\Box[x] \rightsquigarrow \lambda x.x$$

# 4 Properties

In this section we show some basic properties of this calculus. Many of the results here also apply (with suitable change of syntax) to the version with names (see [5]).

**Theorem 4.1 (Confluence)** *If $t \rightsquigarrow u$ and $t \rightsquigarrow v$ then there is a term $s$ such that $u \rightsquigarrow^* s$ and $v \rightsquigarrow^* s$.*

**Proof.** There are just two critical pairs for this calculus, between the *Beta* and *App2* rules, and the *Var* and *Comp* rules, which are easily resolved:




Thus by a standard result in rewriting (Diamond Lemma), we obtain confluence of the reduction system. We remark that other critical pairs are eliminated by the condition that substitutions must be closed, in particular the potential critical pair between the *Beta* and *App1* is eliminated in this way. □

Confluence also holds even if we remove some of the conditions on the rewrite system (we can drop the closedness condition on *App1* and *App2*).

**Lemma 4.2 (Completion of closed substitutions)** *If $v$ is a closed term, then $(t[v^{\epsilon}])^s$ is not a normal form.*

**Proof.** The compilation function gives a correctly annotated term, and reduction preserves this correctness. If $t$ is $\Box$, application, or abstraction, then we can apply one of the rules for substitution. If $t = u[w]$ then the substitution will go to $w$ (it cannot be moved to $u$ since the *Beta* rule that created the substitution could not be applied with an open function), therefore we can apply the *Comp* rule. □

**Lemma 4.3 (Substitutions do not overtake)** *The order of substitutions is preserved by the reduction rules.*

**Theorem 4.4 (Termination of substitutions)** *There are no infinite reduction sequences starting from $t[u]$ using only the rules for substitution.*

**Proof.** We define an interpretation that associates to each term a multiset with one element for each sub-term $v[w]$ occurring in $t$, which is the distance from the root of $v$ to the final destination. Each application of a substitution rule decreases this interpretation, since we always apply a substitution to a sub-term of $t$ or erase it, and the distance is reduced. □

**Theorem 4.5 (Preservation of Strong Normalization)** *If $t$ is the translation of a strongly normalizable $\lambda$-term then $t$ is strongly normalizable.*

The proof of this result is similar to the corresponding result for Closed Reduction [5], which in turn was inspired by the proof for $\lambda\upsilon$ [10].

We complete this section with an important result, which is that this calculus is adequate for the evaluation of closed terms:

**Theorem 4.6 (Adequacy)** *If $t$ is a closed term, then $t \rightsquigarrow^* \lambda t'$ (weak head normal form).*

Thus we can use this calculus to obtain a lazy evaluator for the $\lambda$-calculus. What is even more surprising is that for closed terms we can simulate both call-by-name and call-by-value reduction sequences: note that when we copy a term (which is clearly identified in a single rule of the calculus) then we can decide to reduce the substitution to normal form first to obtain sharing of redexes. However, this is a strategy that we can impose on top of our calculus, and is not forced.

# 5 A typed version of the calculus

In this section we present a typed version of this calculus. The types assigned are simple types as used in the $\lambda$-calculus: function types $A \to B$, and type variables $A, B, \ldots$

**Definition 5.1 (Typed Terms)** *Variables $\square$ are annotated with types, and each term $t$ is then assigned a type $t : A$, as given by the following set of rules:*

$$\frac{}{\square_A : A}\,(Ax) \qquad \frac{t^{s_1} : A \to B \qquad u^{s_2} : A}{(t^{s_1} u^{s_2})^s : B}\,(App) \qquad \frac{t^{s_1} : B \qquad (A \text{ fresh})}{(\lambda^- t^{s_1})^s : A \to B}\,(\lambda^-)$$

$$\frac{t^{s_1} : B \qquad P(t, s_1) = A}{(\lambda t^{s_1})^s : A \to B}\,(\lambda) \qquad \frac{t^{s_1} : B \qquad u^{s_2} : A \qquad P(t, s_1) = A}{(t^{s_1}[u^{s_2}])^s : B}\,(Subst)$$

*where the function $P(t, s)$ is given by the following:*

$$\begin{array}{lcl} P(\square_A, s) & = & A \\ P(\lambda^- t^{s_1}, \downarrow s) & = & P(t, s_1) \\ P(\lambda t^{s_1}, \downarrow s) & = & P(t, s_1) \\ P(t^{s_1} u, \curvearrowleft s) & = & P(t, s_1) \\ P(t u^{s_2}, \curvearrowright s) & = & P(u, s_2) \\ P(t^{s_1} u^{s_2}, \curlywedge s) & = & P(t, s_1) = P(u, s_2) \end{array}$$

*The function $P(t, s)$ used in the rules $(\lambda)$ and $(Subst)$ finds the type of the variable position associated to the first substitution arriving to a term (in the case of an abstraction, this is the bound variable). Also note that for a term to be typable all the occurrences of the same variable must have the same type.*

**Example 5.2** *We give two small examples of type derivations in this system.*

1. $(\lambda \square_A) : A \to A$

$$\frac{\dfrac{}{\square_A : A} \qquad P(\square_A, \epsilon) = A}{(\lambda \square_A) : A \to A}$$

2. $\lambda(\lambda^- \square_A)^\downarrow : A \to B \to A$

$$\frac{\dfrac{\dfrac{}{\square_A : A}}{(\lambda^- \square_A)^\downarrow : B \to A}\,(B \text{ fresh}) \qquad P(\lambda^- \square_A, \downarrow) = A}{\lambda(\lambda^- \square_A)^\downarrow : A \to B \to A}$$

The most basic result that we have for this typed calculus is that reduction preserves types

**Theorem 5.3 (Subject Reduction)** *If $t : A$ and $t \rightsquigarrow u$ then $u : A$.*

Finally, we have a strong normalization result for this calculus:

**Theorem 5.4 (Termination)** *If $t : A$ then $t$ is strongly normalizable.*

We end this section by stating that there are several variants of the type system given which can perform additional checks on the structure of the term. In the above we have always assumed that terms are those given by the translation function: typed $\lambda$-terms can be translated into our calculus and can be given the same type. One specific variant also gives types for the directors, thus we can additionally check that the term is well-formed.

# 6 Conclusions

There are a number of issues that we are currently investigating with respect to this calculus:

Can our calculus be generalized to deal with $\beta$-reduction in full generality? As we pointed out in Section 3 reductions cause changes in the string, thus our choice was to minimize this effect. Nevertheless, it seems a possible extension of the idea, although it is not clear that the resulting system will be of practical interest (the reduction rules will be global, in that we have to modify many director strings not directly involved in the redex).

Our calculus can be seen as providing one possible way of identifying the path which a substitution must follow to reach its destination. In other words, substitutions are directed towards the leafs of the term. Proof nets for linear logic [6] have an alternative approach in that abstractions contain a pointer directly to the occurrence of the variable. The Geometry of Interaction [7] for proof nets is also about paths, and the strategy used there requires closed reductions too. An important aspect of our future work is to try to resolve this gap.

Finally, some of the current questions that we are investigating for this calculus are the length of reduction sequences with respect to other explicit substitution calculi, and environment machines for closed reduction.

# References


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

[2] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 233–246. ACM Press, January 1995.

[3] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.

[4] N. G. deBruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.

[5] M. Fernández and I. Mackie. Closed reduction in the $\lambda$-calculus. In J. Flum and M. Rodríguez-Artalejo, editors, *Proceedings of Computer Science Logic (CSL'99)*, number 1683 in Lecture Notes in Computer Science, pages 220–234. Springer-Verlag, September 1999.

[6] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[7] J.-Y. Girard. Geometry of interaction 1: Interpretation of System F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium 88*, volume 127 of *Studies in Logic and the Foundations of Mathematics*, pages 221–260. North Holland Publishing Company, Amsterdam, 1989.

[8] T. Hardin, L. Maranget, and B. Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–176, March 1998.

[9] J. Kennaway and M. Sleep. Director strings as combinators. *ACM Transactions on Programming Languages and Systems*, 10:602–626, 1988.

[10] P. Lescanne and J. Rouyer-Degli. The calculus of explicit substitutions lambda-upsilon. Technical Report RR-2222, INRIA, 1995.

[11] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International, 1987.

[12] K. Rose. Explicit substitution - tutorial and survey. Lecture Series LS-96-3, BRICS, Dept. of Computer Science, University of Aarhus Denmark, 1996.

[13] N. Yoshida. Optimal reduction in weak lambda-calculus with shared environments. *Journal of Computer Software*, 11(6):3–18, November 1994.

# Axiomatic Rewriting Theory and Explicit Substitution

— Invited paper —

Paul-André Melliès

# Verifying Explicit Substitution Calculi in Binding Structures with Effect-Binding

**Burkhart Wolff**

Universität Freiburg
Institut für Informatik
wolff@informatik.uni-freiburg.de

**Abstract.** *Binding structures* enrich traditional abstract syntax by providing support for representing binding mechanisms (based on deBruijn indices), term-schemata and a very clean algebraic theory of substitution. We provide a novel binding structure with the following main results:
1) The formalisation of a *generic* binding structure with the novel concept of *effect-binding* that enables the explicit representations of both contexts and terms inside one term meta-language,
2) The foundation of a formal (machine-assisted) *substitution theory* of effect-binding that is well-suited for mechanisation. This can be used for the systematic and correct development of new calculi with explicit substitutions.
The substitution theory is formally proven in Isabelle/HOL; the implementation may serve as (untyped) framework for *deep embeddings.*

## 1 Introduction

The concept of *variable binding* is omnipresent in symbolic data-structures representing programs, logical formulae and specifications. If these data-structures are manipulated in program transformers, rewriters, theorem provers and partial evaluators, the *integrity of binding* must be maintained, i.e. variable clashes or variable captures must be avoided during operations like *substitution, matching* or *unification.*

Since the problem is so fundamental, it is very tempting to provide a generic meta-language providing such operations and to solve the technical and theoretical problems once and for all. Among the numerous approaches to these meta-languages, the λ-calculus plays a key role. While traditional λ-calculus treats substitution as an atomic operation hidden inside the β-reduction rule, modern variants such as ([ACCL 91], [Les 94]) provide *explicit substitution* operations based on de Bruijn's indexes representing bound and free variables.

The λ-calculus is a block-structured language: a variable $x$ bound in the abstraction $C[\lambda x.E]$ is only visible inside $E$ and not in its surrounding term $C[]$. As consequence, when representing the *context* of a term, i.e. a special data-structure associating additional information, e.g. values, proof-terms or type information, with its free variables, it is necessary to nest abstractions arbitrarily deep. A *value context* $x_1=v_1, ..., x_n=v_n$ for a term $E$ for example (also called *environment* or *definitional context* for $E$), may be represent this as:

$$(\lambda x_1 ... x_n.\ E)\ v_1\ ...\ v_n$$

As for a type context $x_1:T_1, ..., x_n:T_n$, it is common to represent it by the following nesting of abstractions:

$$\lambda x_1:T_1, ..., \lambda x_n:T_n\ .\ E$$

While this nesting-technique is adequate and widely used when *representing* type-contexts inside proof-terms in logical frameworks like LF, ECC or Isabelle, it is quite complex to *manipulate* contexts and to *reason* over them in this representation. The necessity for operations over contexts arise, for example, when merging two theories (and thus generating a common context and coercing the formulas to it), when deciding that one term is valid in a sub-context, or when reasoning over "skeletons" of contexts or parametric contexts. As a consequence, many implementations such as HOL or Isabelle represent bindings into the context differently, namely by introducing explicit constant symbols into the λ-calculus and separating the constant-symbol context (called *signature*) from the variable context.

In [Wol99], a meta-language with substitution operations — called binding structure — is presented, that allows for the representation of contexts in a non-nesting way that is closer to efficient implementations while maintaining a uniform and clean algebraic theory of substitution both for variables and constants. Besides the usual block-structured binding, the structure offers the concept of effect-binding that mimicks the binding behaviour of global declaration operators. In this paper, we use a slightly simplified version of this binding structure to develop standard and effect-binding calculi with explicit substitution to verify their correctness, i.e. their binding integrity. Since our binding structure is implemented in Isabelle/HOL (see [Isabelle]), this verification task can be assisted mechanically (which is in fact fairly easy[1]). However, the investigation of (ground)-confluence and normalization properties of the verified calculi is not in the scope of this paper.

This paper proceeds as follows: In the next section, we will introduce a generic binding structure, called T-structure, and motivate the key-concept *effect-bindings* that enable the explicit representation of signatures (aka type-contexts or environments). In the following sections, the theory for substitutions is developed. Finally, two applications are demonstrated: First, by *interpreting* an existing substitution calculus in the T-Structre theory, we inherit a correctness result (with respect to binding integrity) for this calculus; second, we derive a conceptually new calculus with explicit substitutions from T-Structures — called λC — exploiting effect bindings.

## 2 T-Structures and Effect-Bindings

In this section, we will introduce a simplified version of T-Structures described in [Wol99][2] and will demonstrate the concept of effect binding with some examples. The basic terms of T-Structures — called T-terms — consist of deBruijn indices, an empty term ◊ and a kind of generalized application — called *construction* — built over a set of binding operators *Op* that will be used to control the binding inside and outside a T-term (built over *Op*).

---

1 the proof documentation for all proofs of this paper can be found at http://www.informatik.uni-freiburg.de/~wolff/isa_doc/TC/index.html.

2 In this presentation, we ommitted metavariables that turn T-structures into an extension of Combinatory Rewrite Systems [KOR93] by effect binding.

**Definition 2.1: T-Terms.** Let *Op* be a (possibly infinite) set of binding operators, then the set of T-terms *T(Op)* is freely constructed over the variants:

| | |
|---|---|
| deBruijn variables | V : Nat → T(Op) |
| empty term | ◇ : T(Op) |
| construction | _⟨_,_⟩: Op × T(Op) × T(Op) → T(Op) |

Throughout this paper, the underscore "_" symbol in declarations will be used to denote infix-notations; thus, given a binding operator $p$ and two T-terms $s$ and $t$, we will write for $p\langle s,t\rangle$ instead of $(\_\langle\_,\_\rangle)(p,s,t)$ for a construction.

We will associate with *Op* two functions: *depth:Op → nat* and *effect: Op → nat*. The triple *B=(Op, depth, effect)* is called the *binding signature* of an object-language; the set $T_B$ contains all T-terms of a language according to *B*. Let $p$ be an operator in a construction $p\langle s,t\rangle$, then *depth p* characterises the number of local variables declared according block-structured visibility in t, while *effect p* characterises the number of variables declared globally in a linear visibility style known from SML or ADA "after" the construction, i.e. in its right context. The scopes induced by a construction $p\langle s,t\rangle$ are turned dark in the following diagram:

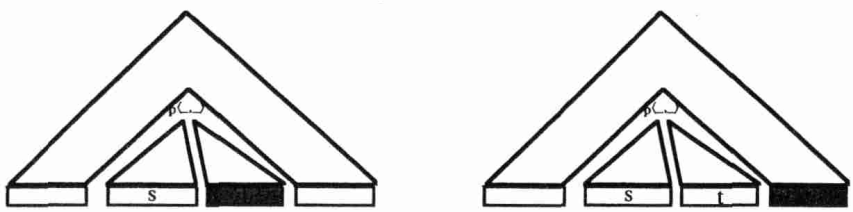



Fig 1.2: block structured binding effect binding

**Example 1**: We define the binding signature for the untyped λ-calculus: Let $Op_\lambda$ = {LAM, APP}, let *depth x = if x=LAM then 1 else 0* and *effect x = 0*. Then we can represent the term $\lambda x.\ y\ x$ by $LAM\langle\diamond, APP\langle V\ 1,\ V\ 0\rangle\rangle$ in the T-Structure $T_{Op_\lambda}$. Again, we will relax the notation and write *LAM(APP(V 1, V 0))* instead.

For convenience, we will use a convention in the following to define a binding signature: the depth of a binding operator $p$ will be denoted by a superscript, while the effect will be defined by a subscript of $p$. Acoording to this convention, we can declare the binding signature of Example 1 by: $Op_\lambda = \{LAM^1_0,\ APP^0_0\}$.

**Example 2**: In order to demonstrate the flexibility of effect-binding structures, we present a fragment of the raw-term language of a variant of the calculus of constructions CC [CH 88]:

$$Op_{ECC} = \{APP(\_,\_)^0_0,\ LAM(\_,\_)^1_0,\ ALL(\_,\_)^1_0,\ \_\times\_^1_0,\ \_\to\_^1_0,$$
$$SORT(\_)^0_1,\ CON(\_)^0_1,\ \_;\_^0_0,\ \_\vdash\_^0_0\}$$

The following CC-example (in the original notation) describes a type-context for a type-constructor *set* that depends on a type *a* and on an ordering on that type. This is motivated by efficient implementations in programming practice. A type-context consists of a sequence of declarations, written [_:_]. The symbol * denotes the "type" of all types as well as the empty context; hence, [bool:*] simply says "bool is a type".

$\Gamma_0$ = [bool : *][int : *][ _≤_ :int × int → bool]
[set : (λa : *) (λord. a × a → bool)*]
[_∪_ : (λa : *) (λord: a × a → bool). set(a)(ord)×set(a)(ord)→set(a)(ord)] *

Over such a type-context $\Gamma_0$ the following *judgement* $\Gamma \vdash A$ can be constructed that states the commutativity of the declared union-operator instantiated with the type *int* and its function _≤_ (assumed to be an order):

$\Gamma_0 \vdash \forall$ A, B : set(int)(≤). A $\cup_{(int)(\leq)}$ B = B $\cup_{(int)(\leq)}$ A

This term with its complex binding dependencies between sorts, constants and formulas can is represented in $T_{OPECC}$ as follows:

$\Gamma_0$ = SORT(◇); SORT(◇); CON(V 0 × V 1 → V 3); SORT(LAM(◇, ...));...
$\Gamma_0 \vdash$ ALL(APP(APP(V(3),V(4)), V(5)), ALL(...))

Note that the judgement symbol ⊢ is an ordinary constructor symbol of the object language (in contrast to the original formulation, where it is a notational part of the meta-language). Note, moreover, that it is the top-most operator in the T-term: selecting the context or the formula is just projection into the topmost construction. Adding a new context into a judgement can be implemented simply by appending one context to the next and performing appropriate (but simple) adjustment operations (see next section).

It is a straight forward exercise to embed object languages like SML in T-Structures in a similar way: toplevel-declaration constructions like

let val x = ...;

or

let fun f x = ...;

can be represented by the operators $LETVAL^0_1$ and $LETFUN^1_1$.

## 3 Adjustments

*Adjustments* (following the terminology of [Tal 93]) are ubiquitous in the theory of T-structures. When manipulating a T-term, operations are necessary that set de Bruijn-indices to appropriate values such that they "point" to another location in the term.

A prerequisite of adjustments in binding structures with effect binding is a function called *effect of t* (written $t). It just sums up all effects of binding operators occurring in *t* and is defined as follows:

**Definition 3.1: effect of a term.**

(1) $(V n) = 0
(2) $ ◇ = 0
(3) $(p⟨s, t⟩) = $s + $t + effect p

We are now ready for the definition:

**Definition 3.2: up and down adjustments.**

(1) V n ⊵ (i, k) = (if k ≤ n then V (k + i) else V n)
(2) ◇ ⊵ (i, k) = ◇
(3) p⟨t, u⟩ ⊵ (i, k) = p⟨t ⊵ (i, k), u ⊵ (i, k + depth s + $ t)⟩

(4) V n ⊴ (i, k) = (if k≤n then V(max (n-i) k) else V n)
(5) ◇ ⊴ (i, k) = ◇
(6) p⟨t, u⟩ ⊴ (i, k) = p⟨t ⊴ (i, k), u ⊴ (i, k + depth s + $ t)⟩

The up adjustment over *(i, k)* increments all free variables larger *k* in a term *t*. *k* will also be called the *cutpoint* of the adjustment. In the construction case, all locally

bound variables have to be taken into account: this happens by incrementing the cutpoint with the number of local variables of the construction depth s plus the effect of *t*. The treatment for down adjustments is dual. The operation $t\triangleright(i, k)$ is often denoted $\tau_k^i(t)$ in the literature for the λ-calculus, from which it only differs by the computation $ t in (3) and the fact, that it is based on T-terms and not the λ-calculus. The dual $t\triangleleft(i, k)$ is largely unknown.

We will use the notation $t \overset{*}{\triangleright} M$ with $M = [(i_1,k_1),...,(i_n,k_n)]$[3] for *m-adjustments* $t \triangleright (i_1,k_1) \ldots \triangleright (i_n,k_n)$. The operation *m-adjustment lift* increases all cutpoints in an m-adjustment list by n: $[(i_1,k_1),...,(i_n,k_n)] \uparrow n \equiv [(i_1,k_1+n),...,(i_n,k_n+n)]$.

The main results of the adjustment theory are collected in the following theorem:

**Theorem 3.3: up and down adjustments.**

(1) $ (t ▷ (i, k)) = $ t
(2) t ▷ (0, k) = t
(3) t ▷ (i, k) ◁ (i, k) = t
(4) t ▷ (i, k) ▷ (j, k) = t ▷ (i + j, k)
(5) t ▷ (i, k + l) ▷ (j, l) = t ▷ (j, l) ▷ (i, k + j + l)

These results extend to their duals and m-extensions. Theorem 3.3(5) is called *quasi-commutation* and is the key theorem for rearrangements of the adjustment order. Note, however, that the converse to 3.3(3) t ◁ (i, k) ▷ (i, k) = t does not hold.

We will also define the usual concept of free variables in a term *fv:*

**Definition 3.4: free variables.**

(1) fv (V x) = {x} (2) fv ◇ = {}
(3) fv(p⟨s,t⟩) = fv s ∪ {y | y+$s+depth p : fv t}

We can establish a connection between the converse of 3.3(3) and free variables:

( t ◁ (i, k) ▷ (i, k) = t ) = (∀x. k ≤ x < k+i ⇒ x ∉ fv t)

This means that we can express traditional reasoning over the non-occurrence of free variables in a certain range by an algebraic argument via adjustments.

Finally, we will see how global operations on contexts can be treated efficiently in T-Structures: assume, that we want to merge two judgements $\Gamma_1 \vdash A$ and $\Gamma_2 \vdash B$ with (disjoint) contexts $\Gamma_1$ and $\Gamma_2$. This can be done by constructing the common context simply by $\Gamma_1; \Gamma_2$ and by coercing the formulas by $A\triangleright(\$\Gamma_2, 0)$ resp. $B\triangleright(\$\Gamma_1, \$\Gamma_2)$. On the other hand, if we want to decide if in $\Gamma_1; \Gamma_2 \vdash A$ the formula $A$ is also valid in sub-context $\Gamma_1$, we can simply ask if A has bindings into $\Gamma_2$ or algebraically: $A\triangleleft(\$\Gamma_2, 0)\triangleright(\$\Gamma_2, 0) = A$. Since merges of theories are usually built blockwise in implementations, merges on non-disjoint contexts can be built efficiently from these two cases.

## 4 Substitutions

---

[3] We use ellipses throughout this presentation instead of the precise formulation: $t \overset{*}{\triangleright} M \equiv$ foldl (λt' (n,m). t' ▷ (n, m)) t M.

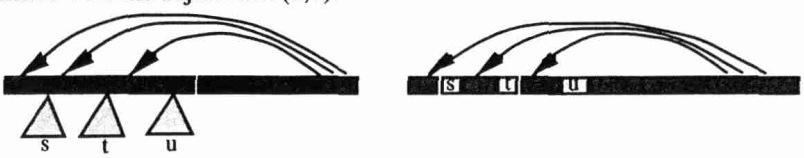
This section is devoted to the definition and the properties of the simultaneous substitution function written t ←| S for variables. Substitutions in deBruijn-indexed Binding Structures are combinations of replacements of some variables and adjustments of all other (free) variables in a term. A data-structure that contains both information is called *closure.*

Closures are well-known in substitution calculi and implementations of $\lambda^{\rightarrow}$. In the presence of effect binding, however, there is a new complication: since a substitution may insert *several* subterms with non-zero effect into a term, a *list* of elementary adjustments, hence a m-adjustment, may be necessary in order to re-establish the correctness of binding.

Let us consider the following example taken from $T_{OPECC}$ — the insertion of a term into an environment as a result of a substitution:

SORT(◇); V 1 ; CON(V 0) ⇝ SORT(◇); CON(◇) ; CON(V 1)

In a right context of this term, variable V 0 would be bound to the constant declaration CON(V 0) and V 1 to the type declaration SORT(◇). After the substitution, there is an "insertion" into the name-space between V 0 and V 1 which can be established with the adjustment (1,1).




In this section, we will first develop the theory of closures and then turn to an auxiliary function that computes these m-adjustments called *drive* of a substitution (written t $←| S). On this basis, the substitution function will be defined as a primitive recursive system.

### 4.1 Closures

Generally speaking, functions that map variables to terms are called *valuations*. They are the material from which substitutions — mappings of terms to terms — were built. Finite, intensional representations of functions are usually called *closures*. A *lookup-function* _^_ is used to convert a closure in its valuation. We will develop for both substitutions and fillings an appropriate notion of closure.

For substitutions, we decided to represent their closures by a pair consisting of a list of substitutes for the first *n* variables and an m-adjustment list in order to control the renaming all other variables. Hence, a substitution closure has type (α T list × (nat*nat)list) and its lookup is defined by:

**Definition 4.1.1: substitution closure lookup.**

(L, M) ^ i ≡ if i < length L then nth i L else V (i - length L) ⊵* M

Closures must be transformed during the substitution process. For example, when entering a subterm representing a new scope (such as t in LAM(t)), the substitution list must be *shifted* by one, i.e. prefixed by the substitute list [V 0] and hence mapping all local variables to themselves. The elements in the substitution list must be adjusted by (1,0) (i.e. all de Bruin indexes will be increased) in order to avoid name capture. This operation is traditionally called *shift* and written S ⇑ n.

Since in a substitution theory with effect binding insertions in a name-space can occur anywhere and not just in front as in substitution theories for languages with block-oriented visibility, shifting must be generalised to *spreading* as follows:

**Definition 4.1.2: spreading.**[4]

$$(L,M)\ \nabla\ (n,d) \equiv \text{if } n = 0 \text{ then } (L,M)$$
$$\text{else (if } d \le \text{length } L \text{ then(map } (\lambda x.\ x \triangleright (n, d))(\text{take } L\ d)\text{++}$$
$$\text{map } V\ ([d\ ..\ (n+d)])\ \text{++}$$
$$\text{map } (\lambda x.\ x \triangleright (n, d))(\text{drop } L\ d),$$
$$M\uparrow\ n),$$
$$\text{else (map } (\lambda x.\ x \triangleright (n, d))\ L,\ M\text{++}[(n,d)])$$

where take and drop are the usual cutting operations on lists in functional languages and [d .. (n+d)] creates the list of numbers from d upto (n+d) (exclusively!). Again, we will also extend spreadings to m-adjustment lists: $A \overset{*}{\nabla} M$.

It is now straight-forward to define the traditional shift as $S \Uparrow n \equiv S \Uparrow (n,0)$. The following theorem collects the major results of the substitution closure theory:

**Theorem 4.1.3: spreadings.**

(1) $A\nabla(0, d) = A$ (2) $A \Uparrow 0 = A$

(3) $A \Uparrow \text{Suc } n = (V\ 0 :: \text{map } (\lambda t.\ t \triangleright (1, 0))\ (\text{fst } (A \Uparrow n)),\ [(1,0)])$

(4) $A \wedge d \triangleright (i, k) = A\nabla(i, k) \wedge \text{lift\_up } d\ (i, k)$

(5) $A\nabla(i, 0)\nabla(a, b + i) = A\nabla(a, b)\nabla(i, 0)$

(6) $A \wedge d \overset{*}{\triangleright} M = A\overset{*}{\nabla}M \wedge \text{lift\_up}^*\ d\ M$ (7) $A\nabla(i, 0)\overset{*}{\nabla}M\uparrow\ i = A\overset{*}{\nabla}M\nabla(i, 0)$

(8) $(\forall x.\ A \wedge x = B \wedge x) \Rightarrow \forall\ y.\ (A\nabla m)\wedge y = (B\nabla m)\wedge y$

**Proof-Sketch:** (1) to (3) follow by simplification from Definition 4.1.2. If lift_up d (i,k) ≡ (if k ≤ nat then V (nat + i) else V nat) (i.e. the V-case in the up adjustment definition 3.3(1)), (4) is a matter of nested case-distinctions i=0, k<fst A and d<fst A and tricky arithmetic reasoning. (5) is a consequence of Theorem 3.4(5). (6) and (7) are generalisations of (4) and (5) via induction over M. □

## 4.2 Drives

Keeping in mind the motivation for drive at the begin of this main section, we give immediately its primitive recursive definition:

**Definition 4.2.1: drive.**

(1) $V\ n\ \$\leftarrow|\ \phi = (\text{if } \$(\phi\wedge n) = 0 \text{ then } [] \text{ else } [(\$(\phi\wedge n),0)])$

(2) $\diamond\ \$\leftarrow|\ \phi = []$

(3) $p\langle t,u\rangle\ \$\leftarrow|\ \phi = (t\ \$\leftarrow|\ \phi)\uparrow\ (\$\ u + \text{effect } p)\ \text{++}$
$(u\ \$\leftarrow|\ \phi \Uparrow (\text{depth } p + \$t))\uparrow\ \text{effect } p$

The first case is the base case: whenever a variable is substituted with a term that has non-zero effect, all variables in the right context must be adjusted by the effect of the

---

[4] This definition is slightly more general than the one used in our actual version in the Isabelle/HOL-theory.

substitute. The tricky case is the third one — obviously, the drives produced by t and u must be concatenated (after some manipulations). The substitution of u (and hence the drive-construction) must be performed in its new block, i.e. its closure must be shifted by depth p + $t. Now, the drives of both t and u must be lifted by the effects occurring in their right local sub contexts within the construction.

**Theorem 4.2.2: drive.**

(1) $(\forall x.\ \$(A \wedge x) = \$(B \wedge x)) \Rightarrow t\ \$\leftarrow| A = t\ \$\leftarrow| B$

(2) $t_0\ \$\leftarrow| A = []$ (3) $t \triangleright^* M\ \$\leftarrow| S \overset{*}{\nabla} M = t\$\leftarrow| S$

where $t_0$ is a term of a block-structured language[5], i.e. a language in which all operators have no effect. (2) suggests that drive does not occur in such languages.

### 4.3 The Substitution Function

The definition of the substitution is now somewhat analogous to the one of drive:

***Definition 4.3.1: substitution.***

(1) $V\ x \quad \leftarrow| \phi = \phi\hat{}\ x$

(3) $\diamond \quad \leftarrow| \phi = \diamond$

(4) $p\langle t,u\rangle \quad \leftarrow| \phi =$ (let M = $(t\ \$\leftarrow| \phi)\uparrow$ depth p

in $p\langle t \leftarrow| \phi,\ (u \leftarrow| \phi \Uparrow (\$t+\text{depth } p)) \triangleright^* M\rangle$)

The first two cases are straight-forward for a substitution definition. The crucial case is of course the case of the construction. However, it is easy to see that the substitution must distribute over construction such that the first subterm is simply $t\leftarrow|\phi$. For the other subterm u, this substitution produces drive M (that must be lifted to the new level of local variables depth p). The substitution closure $\phi$ must be transformed in order to take into account the new "declaration level" ($t+depth p) via a shift.

As a consequence of theorem 4.2.2(3) it is easy to see that in block-structured languages the drive computation can be simply cancelled and the following, more familiar substitution scheme results:

$$p\langle t_0,u_0\rangle \leftarrow| \phi = p\langle t_0 \leftarrow| \phi, u_0 \leftarrow| \phi \Uparrow \text{depth } p\rangle$$

**Theorem 4.3.2: substitution.**

(1) $\$ (t \leftarrow| S) = \$\ t + \text{sum}(\text{map fst } (t\ \$\leftarrow| S))$

(2) $t \triangleright^* M \leftarrow| S \overset{*}{\nabla} M = (t \leftarrow| S) \triangleright^* M$

(3) $t \leftarrow| (\text{map } V\ [0..i],\ i + k) = t \triangleright (k,\ i)$

(4) $(\forall n.\ n \notin fv\ t \vee S\ \hat{}\ n = T\ \hat{}\ n) \Rightarrow t \leftarrow| S = t \leftarrow| T$

(5) $t \leftarrow| (A,m) \leftarrow| (B,m') = t \leftarrow| (\text{map } (\lambda t.\ t \leftarrow| (B,i))\ A,m++m')$

(if length A = length B)

Theorem 4.3.2(1) allows to reduce the computation of the effect of a substitution on the computation of drive; while (2) shows that m-adjustment distributes over substitution via spreading. (3) explains adjustment as a special case of substitution, while

---

5 In the Isabelle/HOL implemention, this informal requirement is made precise with a certain type-constraint called Isabelle type classes.

(4) explain extensionality and (5) fusibility of substitutions. These theorems represent a generalisation of Lemma 1.13/14 of [HS 86], where the substitution theory of the λ-calculus is discussed.

### 4.4 Correctness of Substitution: Binding Integrity.

Now, it arises the crucial question if this general form of substitition is *right* — beyond the fact, that it fulfills sensible generalizations of well-known properties. What we definitively require for a substitution $t \leftarrow| (S,M)$ (in a language based on deBruijn variables) is:

1) Unsubstituted free variables in a term are decreased by length S and adjusted by M

(in a special case, they remain invariant)

2) Free variables in a substitute remain invariant under substitution

3) Bound variables remain bound to the same binding operator in a term

These three postulates — which constitute what we call binding integrity — represent the core of what we expect for a substitution and its correct behaviour. The postulates 1) and 2) hold in all T-Structures, as can be seen more or less directly from the following:

**Theorem 4.4.1: Binding Integrity.**

$$(1)\quad fv\ (t \leftarrow| (S,M)) = \bigcup_{i \mid i<\text{length } S \wedge i \in fv\ t} fv\ (\text{nth } i\ S) \cup (\lambda x.\ x \overset{*}{\triangleright} M)\ ``\ \{i \mid i+\text{length } S \in fv\ t\}$$

$$(1')\quad fv\ (t \leftarrow| (S,[(0,\text{length } S)])) = \bigcup_{i \mid i<\text{length } S \wedge i \in fv\ t} fv\ (\text{nth } i\ S) \cup \{i \in fv\ t \mid \text{length } S \leq i\}$$

We first turn to the special case (1'): Here, the free variables of the substitution that leaves all unsubstituted variables invariant stem either from a substitute (i.e. there must have been a free variable i occurring in t for which exists a replacement in S) or must have occured in the term before and there is no replacement in S.

The reason for the more complex definition of substitution and binding integrity ("decreased by length S and adjusted by M") is a generalization motivated by the β-reduction in the λ-calculus and similar rules (see section 5 for details): In

$$APP(LAM(a),b) \rightarrow a \leftarrow| ([b],[])$$

we eliminate a binding operator, and in order assure binding integrity *of this rule* (in the sense: free variables remain invariant under reduction), all free variables occuring in a and not bound to LAM (i.e. $\{i \in fv\ t \mid 1 \leq i\}$) must be down-adjusted by one. Thus, a substitution must manipulate all free variables in a.

We now turn to the second part of (1) and see that it formalizes the requirement "decreased by length S and adjusted by M".

As for the postulate 3), we can argue that it implicitly also follows from 1) and 2) given the fact that bound variables are in some subterm free variables (remaining suitably invariant) bound by their context.

## 5 Application: Deriving Standard Substitution Calculi

It is now an easy exercise to derive and justify calculi with explicit substitutions for λ-calculi made popular by [ACCL 91] and further developed for example in [Les 94], whose calculus λυ is used here because of its brievity — we expect the verification of other calculi with explicit substitution to be completely analogous. For our derivation, the trick is done by giving the substitution operator symbols like

$$s ::= a/ \mid \Uparrow(s) \mid \uparrow$$

an interpretation in our substitution theory. For this purpose, we interprete the sort *s* above by $T_{OP_\lambda}$, hence the type of substitution closures for the untyped λ-calculus and view the substitution t[S] just as a notation for $t \leftarrow| S$. The interpretation of the substitution operators on *s* is done by the following constant definitions:

| | | | |
|---|---|---|---|
| inc_def: | a / | ≡ | ([a],[]) |
| shift_def: | (⇑S) | ≡ | S ⇑ 1 |
| up_def: | (↑) | ≡ | ([],[(1,0)]) |

On this basis we can *derive* λυ from our substitution theory:

**Theorem 5.1: λυ (without beta) lives in $T_{OP_\lambda}$.**

| | | |
|---|---|---|
| App: | APP(a,b) [S] | = APP(a[S],b[S]) |
| Lambda: | LAM(a)[S] | = LAM(a[⇑S]) |
| FVar: | (V 0)[a/] | = a |
| RVar: | V(Suc n)[a/] | = V n |
| FVarLift: | V(0)[⇑S] | = V 0 |
| RVarLift: | V(Suc n)[⇑S] | = V n [S][↑] |
| VarShift: | V(n)[↑] | = V(Suc n) |

**Proof-Sketch:** Elementary simplification with the definitions for depth and effect for $T_{OP_\lambda}$ and the equations constituting the interpretation inc_def,shift_def, up_def, the laws on substitution, lookup, and the Theorem 4.3.2(4) and the mentioned simplifications for block-structured languages (such as $T_{OP_\lambda}$). □

β-reduction APP(LAM(a),b) → a[b/] does not hold as equality — β-reduction has to be introduced as a quotient structure over T-terms. In order to generate a proper model for the complete calculus in Isabelle/HOL, we define an equivalence relation $\leftrightarrow_\beta$: $(T_{OP_\lambda} \times T_{OP_\lambda})$set inductively:

| | | |
|---|---|---|
| (varI) | $V(n) \leftrightarrow_\beta V(n)$ | |
| (lamI) | $t \leftrightarrow_\beta t'$ | $\Rightarrow$ $LAM(t) \leftrightarrow_\beta LAM(t')$ |
| (appI) | $s\Box \leftrightarrow_\beta s', t \leftrightarrow_\beta t'$ | $\Rightarrow$ $APP(s,t) \leftrightarrow_\beta APP(s',t')$ |
| (betaI) | $s\Box \leftrightarrow_\beta s', t \leftrightarrow_\beta t'$ | $\Rightarrow$ $APP(LAM(s),t) \leftrightarrow_\beta s'[t'/]$ |

Based on this (straight-forward) equivalence we can now use the Isabelle mechanism of a type-definition modulo $\leftrightarrow_\beta$ that produces a perfect model of full λυ. Since we *derived* the rules of λυ, we inherit the correctness (binding integrity) from the correctness of the underlying T-Structure; thus, we have a *machine assisted* proof of correctness here.

## 6 Application: Deriving Substitution Calculi with Contexts

We have now established a basis for the development and verification of a conceptually novel substitution canlculus. For this purpose, we present a λ-calculus with *explicit type contexts*, called λC, which is partly an abstraction of the calculus of constructions — and partly close to λυ; this choice is mainly for the sake of a short presentation.

As binding signature of $\lambda_c$ we define: $Op_{\lambda C} = \{LAM^1_0, APP^0_0, CON^0_1\}$ where CON is the construct for constant declaration, hence the basic building block of a type context. Contexts like $x_1{:}T_1, ..., x_n{:}T_n$ can hence be represented inside λC by:

$$APP(CON(T_1), APP( ..., APP(CON(T_{n-1}),CON(T_n))...))$$

We reuse the operators inc, shift and up from the previous section, but add an additional operator mux (written to s<<t,S,n>>) to the substitution calculus as a tribute to effect binding caused by contexts. We define mux in terms of T-Structures:

mux_dri_def $\quad t[\![a,S,n]\!] \equiv t \rhd^* (a\ \$\leftarrow|\ S)\uparrow n$

This definition was found as a kind of invariant when experimenting with the App-theorem. Intuitively, mux takes a term t and adjusts it to a left context a into which a substitution S was performed; this is done by computing the drive produced by this substitution and by lifting it to some level n. This particular operator represents the abstraction from T-Structures and its details which is possible due to the specialization of the genericity to the concrete syntax of λC.

Now we derive the rules of the substitution calculus:

**Theorem 6.1: The rules of λC (without beta) lives in $T_{OP_\lambda}$.**

| | |
|---|---|
| Con: | CON(a) [S] = CON(a [S]) |
| Lambda | LAM(a)[S] = LAM(a[⇑S]) |
| App | APP(a,b) [S] = APP(a [S], (b [S ⇑ $ a]) [[a, S, 0]]) |
| | |
| mux_V | (t [[V x, S, n]]) = t ⊳ ($(V x [S]),n) |
| mux_LAM | (s [[LAM(t), S, n]]) = (s [[t, ⇑S, n]]) |
| mux_CON | (s [[CON(t), S, n]]) = (s [[t, S, Suc n]]) |
| mux_APP | (s [[APP(a,b), S, n]]) = ((s [[a, S, n+$b]])[[b,S ⇑ $a, n]]) |
| | |
| eff_V | $(V x) = 0 |
| eff_LAM | $(LAM(x)) = $(x) |
| eff_APP | $(APP(x,y)) = $(x) + $(y) |
| eff_CON | $(CON(x)) = Suc($(x)) |
| | |
| add0 | 0 + X = X |
| add_Suc | Suc X + Y = Suc (X+Y) |

Additionally, there are the rules FVar, RVar, FVarLift, RVarLift, and VarShift as in the previous section which remain unchanged. The proofs for all of these equations are straight-forward simplifications along the laws of the T-Structure theory; none of the proof-scripts for these equations is longer than 5 lines.

In the next step, we can now generate a model by repeating the equivalence-class construction of the previous section; we propose to restrict betal to effect-free terms (i.e. $(t)=0) in order not to over-generalize the β-reduction in our calculus. Note that within this quotient term construction, we have the freedom to add some structural rules like:

$$APP(CON(T_1),APP(CON(T_2), X) = APP(APP(CON(T_1),CON(T_2)),X)$$

that represent for an elegant algebraic means to model the reorganization in contexts in logics or programming languages.

## 7 Conclusion

We have seen a formal development (i.e. based on theorem prover assistance) of a substitution theory with effect binding based on a implementation-oriented generic data structure. Effect binding can be used to model e.g. type- and constant declarations in object languages, hence are vital for the explicit representation of type contexts or evaluation environments (for operational semantics etc.). Applications for this explicitness range from the theoretical study of implementations of typed λ-calculi (hence systems like Coq, that inspired the author to consider effect binding) to the modelling of module-systems, where signatures of two modules must be "merged".

This work is based on two major design decisions:

1) The substitution theory *with* effect binding is build as a conservative extension of substitution theories *without*; it has been demonstrated, that for block structured languages, all effect-related computations can be cancelled and the result is a generic theory similar to substitution calculi or Klop/Talcott-style meta-languages.
2) It is a distinguishing feature of this theory to be close to a functional program. Applying a sequence of finite differencings — computing the increments to effects, drives, closures and substitution results with one sweep — leads even to an efficient algorithm.

The complexity of effect binding substitution theories is clearly considerable. We would argue, though, that it is not excessive; and most complexity (myriad's of case-distinctions, tricky arithmetic reasoning) is hidden inside the proofs that were done once and for all in a proof assistant such as Isabelle.

Further, we argue that the complexity of the substitution calculi resulting from an *instantiation of T-Structures with a concrete object language* is quite standard; the calculus λC has more or less the same number of rules as, say, λφ, while offering completely new (binding) features.

### 7.1 Comparison to Related Work

When restricting T-Structures to block-structured languages (i.e. for all languages with *effect* $x = 0$)while adding a notion of rule, rewriting notions like "transformations"(such as in [Wol99]) are equivalent to Klop's Combinatory Rewrite Systems and strictly weaker than pattern-rewrite systems ([Nip 91],[MN 97]).

In Coquand's and Huét's description of CC (in [CH 88]), effect-binding was "hidden" in an inductive structure inside the raw term language $\Lambda_i^k$ where *i* corresponds to the effect of all terms in $\Lambda_i^k$ and *k* to their maximal free variable. $\Lambda_i^k$ is constructed over a fixed operator set and hence not generic in our sense; the calculus was built by several inductive definitions that make explicit reference to the so-called *relocation function* $\xi$ (corresponding to $\lambda t.\ t\triangleright(1,0)$)).

Explicit substitution theories have been already extensively discussed throughout this paper. Section 5 shows, that our work can be used as a formal meta-theory for their correct construction. This extends to substitution calculi with explicit type contexts. We are aware of some (informal) work on the correctness of substitution calculi, [LR95] for example, and on formal work on other meta-theoretic properties, but to our knowledge their is no formal (machine assisted) verification work of binding integrity. We are not terribly worried about incorrectness of existing calculi, but we are surprised how *easy* the relevant proofs can be done with a framework like ours, and how the development of (non-trivial) calculi can be turned into a routine task.

### 7.2 Future Work

We see essentially three lines of extension of this work:

1) automated generation of calculi (based on the definitions of new operators),
2) derivation of optimised matching- and unification algorithms for special object languages via partial evaluation techniques, and
3) theoretical study of confluence and normalization properties of the resulting calculi.

Especially 3) is a crucial point for $\lambda C$, good meta-theoretic properties are clearly a prerequisite for its usefulness. It would be an interesting, but also quite challenging goal to give an equally powerful proof support to the solution of these tasks for a suitably wide range of substitution calculi.

## References


[ACCL 91] M. Abadi, L. Cardelli, P.-L. Curien, J.-J. Lévy. Explicit substitutions. J. of Functional Programming, 1(4), pp. 375-416, 1991.

[And 86] P.B. Andrews: *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Academic Press, 1986.

[CH 88] T. Coquand, G. Huét: The Calculus of Constructions. Information and Computation, 76:95 – 120, 1988.

[Chu 40] A. Church: A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940, pp. 56-68.

[Isabelle] The Isabelle-documentation page: www4.informatik.tu-muenchen.de /~nipkow/isabelle. Alternatively: http//www.cl.cam.ac.uk/Research-/HVG/isabelle.html.

[HS 86] J.R. Hindley, J.P. Seldin: *Introduction to Combinators and the λ-Calculus*. CUP, 1986.

[KOR 93] J. W. Klop, V. v. Oostrom, F. v. Raamsdonk: Combinatory reduction systems: introduction an survey. TCS 121 (1-2), pp. 279-308, 1993.

[Les 94] P. Lescanne: From λσ to λυ — a journey through calculi with explicit substitutions. POPL pp. 60-69,1994.

[LR95] P. Lescanne, J. Rouyer-Degli: Explicit Substitutions with deBruijn's levels.

[MN 97] R. Mayr, T. Nipkow: Higher Order Rewrite Systems and their Confluence. *Theoretical Computer Science*, 97.

[Nip 91] T. Nipkow: Higher Order Critical Pairs. In: Proc. 6th IEEE Symp. Logics in Computer Science, pp. 342-349. IEEE Press, 1991.

[Tal 93] C. Talcott: A theory of binding structures and applications to rewriting. Theoretical Computer Science, 112 (1993), pp. 99—143, 1993.

[Wol 99] B. Wolff: A Generic Calculus of Transformations. Dissertation at the University of Bremen. 1997/1999.