On Explicit Substitutions and Names

Article · June 1997

Source: CiteSeer

CITATIONS
2

READS
73

2 authors:

Eike Ritter
University of Birmingham
75 PUBLICATIONS **1,074** CITATIONS
SEE PROFILE

Valeria De Paiva
Topos Institute
203 PUBLICATIONS **2,901** CITATIONS
SEE PROFILE

On Explicit Substitutions and Names

Eike Ritter and Valeria de Paiva [*]

March 20, 1997

Abstract

This paper investigates the difference between giving a semantics of λ-calculi with explicit substitution (*i.e.*, substitution is part of the calculus and not a meta-operation) and using these calculi to design abstract machines. We show that a calculus for explicit substitutions derived according to principles used in type theory and in semantics differs from a calculus derived for implementations. We establish the equivalence between these two versions, and hence obtain an important link between the semantics and implementations. These two different versions coincide in the simply-typed λ-calculus, but again not in calculi with dependent types.

We also show in this paper how the type-theoretic view gives rise to a λ-calculus with explicit substitution and variable names. So far, all calculi with explicit substitution use nameless dummies, the so-called de Bruijn numbers, instead of variables. Calculi with de Bruijn numbers are well-suited for implementations but very difficult to read for humans.

We also sketch applications of this work to the design of abstract machines and theorem proving.

1 Introduction

Explicit substitution calculi (or $\lambda\sigma$-calculi for short) first appeared in a seminal paper by Abadi et al. [ACCL91]. The basic idea is that instead of having substitutions as a meta-level operation, as in traditional λ-calculus, we should make them part of the object-level calculus. The advantages of this approach are twofold. Firstly, it makes it possible to design much more efficient abstract machines as we are allowed to delay substitutions, and secondly it makes it much easier to prove them correct since the calculus and its implementation are closer.

There are several variants of calculi with explicit substitutions. Some of these variants are geared towards semantics, others are derived with implementations in mind. Rather than listing all variants, we explain in this paper

[*]School of Computer Science, University of Birmingham. e-mail: {exr,vdp}@cs.bham.ac.uk.

what we take to be the principal differences between them. This way we describe what appears at first sight as various "design choices" for lambda-calculi. But we then justify why we have to develop calculi for each possible choice if we want to prove semantics and syntax equivalent. Moreover, by using the context handling of type theory as a guide, we are able to define a confluent calculus with explicit substitutions and names—something that Abadi et al. were not able to do.

1.1 Equations first versus Reductions first

There are two main approaches when defining typed λ-calculi, with or without explicit substitutions. The first one, in the spirit of Martin Löf's type theory [ML84], defines the calculus with equations-in-context. Reduction is then a derived notion, obtained by orienting the equations. The second approach considers the set of typed terms as a subset of the set of raw terms, and hence reduction is defined on raw terms, which are not necessarily well-formed. Equality is now the derived notion, namely it is the symmetric and transitive closure of the relation generated by the reduction rules.

The first approach is required when giving semantics to λ-calculi because only well-formed objects have a meaning. The second approach avoids the need to check for well-formedness during reduction, which is incorporated in the first approach. As a consequence, this approach is well-suited for implementations, but a semantics for terms can only be given by showing the equivalence of this presentation to the Martin Löf-style presentation. Whereas this equivalence is easy to prove in the case of the simply-typed λ-calculus (and hence it is not really necessary to differentiate between the two approaches in this case), the difference becomes crucial as soon as we add, for example, dependent types [Str89][1]. This difference becomes crucial again when we consider calculi with explicit substitutions.

This paper presents calculi for both approaches and shows their equivalence (see section 3). This is because we want to connect the implementation, which is based on the second approach, with the semantics, which is based on the first approach.

1.2 Typed versus untyped calculi

There are typed and untyped calculi with explicit substitutions, both of which are presented already in [ACCL91]. The typing rules enforce two different restrictions: firstly, they eliminate expressions with misuse of variables, *e.g.*, ones where we try to substitute two different terms for the same variable simultaneously. Secondly, they ensure that the only well-typed λ-terms are the ones of the simply-typed λ-calculus.

[1]The equivalence proofs can still be done [Geu93], but some of the required properties of the type theories, like confluence and subject reduction, are very hard to establish.

The first restriction is independent from the second: if one takes a typing system consisting of one universal base type Ω (a type Ω such that $\Omega \cong \Omega \Rightarrow \Omega$), the well-formed λ-terms are the terms of the untyped λ-calculus. Hence it is possible to give a semantics for such a calculus by extending the semantics for the untyped λ-calculus. However, such a semantics does not assign a meaning to the expression which tries to substitute two different terms for the same variable. This is a syntactic issue which has to be checked *before* a semantics can be defined. So, contrary to the untyped λ-calculus, there is no semantics for each raw expression of the untyped $\lambda\sigma$-calculus.

1.3 Names versus de Bruijn numbers

Another important kind of choice the designer of a explicit substitution λ-calculus can make concerns the difference between variable names and de Bruijn numbers. De Bruijn numbers were initially considered, as an implementational trick for Automath: instead of using variables like x, y, z de Bruijn proposed to use natural numbers (that correspond to the binding level of the variable), in such a way that a class of α-congruent terms correspond to a single syntactic object. Hence two expressions with variable names are α-equivalent if and only if the corresponding terms with de Bruijn numbers are syntactically equal. More than simply an implementational trick, de Bruijn numbers are helpful when defining the semantics of the calculus in question. The point is that a de Bruijn-number n corresponds exactly to the n-th projection $A_n \times \cdots \times A_1 \rightarrow A_n$.

There is a trade-off between a version of the calculus with de Bruijn numbers and a version with names. Expressions with variable names are much easier to read. The difference becomes apparent even for relatively small terms (*e.g.*, compare the expressions $\lambda x.(\lambda yz.x)(\lambda z.x)$ and $\lambda.(\lambda.\lambda.3)(\lambda.2)$). The main drawback of the version with names is the need to identify terms which only differ in the name of bound variables: the semantics of terms can only be defined modulo α-equivalence. This complicates the definition of the syntax significantly, as the definition of α-equivalence is rather involved (see section 3). On the other hand, α-conversion is not needed for the version with de Bruijn numbers, and the absence of α-equivalence makes this better suited for implementations.

So a judicious use of both versions seems the best option: for the presentation of results in the meta-theory, the version with names is used, and for implementations one uses de Bruijn-terms to handle variable access. Of course a good implementation keeps the variable names as extra information during reduction so that terms can be printed with names rather than with de Bruijn numbers.

1.4 Iterated Substitutions

The fourth choice concerns the need (or not) for composition of substitutions.

The precursor of the $\lambda\sigma$-calculus, Curien's $\lambda\rho$-calculus [Cur91], was designed to capture environment machines and had no notion of iterated substitutions. This is rather restrictive, as nested substitutions arise in several situations: during reduction to normal form rather than weak head normal form, when modelling sharing in environment machines, when modelling instantiation in theorem provers, and as the counterpart of composition in the categorical semantics of λ-calculi. The $\lambda\sigma$-calculus was developed by Abadi et al. [ACCL91] with these applications in mind. Iterated substitutions seem to us an essential part of any $\lambda\sigma$-calculus.

1.5 Explicit weakening

The final kind of difference between variants of explicit substitution calculi concerns an operator for explicit weakening. (This is a minor point compared to the other ones.) An explicit weakening operation is mandatory for the version with de Bruijn numbers because a term which is well-formed in context Γ is not necessarily well-formed in an extended context Γ, A. (This operation is represented in the syntax by the so-called "shifting" $\uparrow$ operator.) Such an operation is convenient in a calculus with variable names as it makes the correspondence with de Bruijn numbers easier, but explicit weakening is not necessary to establish the correspondence with the λ-calculus. In the simply typed λ-calculus case, logical weakening holds, so $\Gamma \vdash t\colon A$ implies $\Gamma, x\colon B \vdash t\colon A$.

Summing up

Summarising, it seems to us that the first "design choices" are not choices at all. We must have both the equations-in-context and the reductions-first versions, both the typed and untyped versions and both the de Bruijn and the names versions, as our goal is the implementation of abstract machines. It also seems essential to have composition of substitution for the reasons outlined above. Explicit weakening or not is, as far as this paper is concerned, a matter of taste.

The paper is structured as follows. We define our calculus of explicit substitutions and equations in context in the next section. Next we discuss issues relating binding operations and α-equivalences in explicit substitutions calculi. We prove the necessary syntactical properties (confluence and normalisation) of our calculus and then we examine the equivalence between the versions of the $\lambda\sigma$-calculus with typed and untyped reduction rules. We conclude by briefly discussing implementations and applications, which are mostly future work.

2 A calculus with equational judgements

In this section we present (with minor modifications) Martin-Löf's λ-calculus with explicit substitutions. This calculus is the $\lambda\sigma$-calculus by Abadi et al. but with names and equations-in-context. Tasistro [Tas93] describes this calculus and gives ample motivation about the form of the judgements and their interpretation. [2]

2.1 Well-formed expressions

We start by presenting raw expressions and defining the judgements for well-formed expressions and then give a few intuitions about the calculus.

Definition 1 (Raw Expressions). *The types of the $\lambda\sigma$-calculus with names are base types and function types $A \Rightarrow B$. The raw expressions of the calculus are given by the following grammar:*

$$\begin{array}{lcl} t & ::= & x \mid \lambda x\colon A.t \mid tt \mid f * t \\ f & ::= & \langle\rangle \mid \langle f, t/x\rangle \mid f; f \end{array}$$

We call expressions of the first kind terms *and expressions of the second kind* substitutions[3]*. Moreover, we write*

$$\langle t_n/x_n, \ldots, t_1/x_1\rangle$$

for

$$\langle \cdots \langle\langle\langle\rangle, t_n/x_n\rangle, t_{n-1}/x_{n-1}\rangle, \ldots, t_1/x_1\rangle$$

.

We identify terms which are identical up to change of bound variables. Because not only the λ-abstraction but also the explicit substitution $f * t$ binds variables, the definition of bound variable is significantly more complex than in the λ-calculus; for a precise definition of the notion of bound variable and of α-equivalence see Section 3.

Judgements for well-formed expressions require an additional kind of raw expressions, namely contexts. Such a context is a list $x_1\colon A_1, \ldots, x_n\colon A_n$ of assignments of a type to a variable. (Contexts are called environments in [ACCL91].) We call a context well-formed if no variable occurs twice in it. From now on we tacitly assume contexts to be well-formed. We denote the empty context, which is the special case of $n = 0$, by $[\ ]$. Note that

[2]We use the term $\lambda\sigma$-calculus as a generic term for any variant of the calculi presented in [ACCL91].

[3]The reader should notice that we also have an *explicit substitution operator*, denoted by $*$, which takes a substitution f and a term t and returns a term $f * t$.

contexts are lists rather than multisets; in other words the order is relevant. This approach generalises to dependent type theory and is compatible with categorical semantics. Because contexts like $x\colon A, y\colon B$ and $y\colon B, x\colon A$ are not identified, there is an explicit representation of the exchange rule. This avoids problems with the existence of normal forms of substitutions; for details see Section 4.

We have two judgements for the well-formedness of raw expressions, namely $\Gamma \vdash t\colon A$, the usual "t is a term of type A in context Γ", and $\Gamma \vdash f\colon \Delta$. The last judgement should be interpreted as "f is an (explicit) substitution for variables in Δ where the free variables of the terms to be substituted are contained in Γ". Such a substitution roughly corresponds to a list of substitutions in the λ-calculus. We call any context Γ' arising from Γ by deleting some assignments $x_i\colon A_i$ a *subcontext*; in that case we write $\Gamma' \subseteq \Gamma$ and call Γ an extension of Γ'.

Definition 2 (Typing Judgements). *The inference rules for the judgements* $\Gamma \vdash t\colon A$ *and* $\Gamma \vdash f\colon \Delta$ *are as follows:*

(i) On terms:

$$\frac{}{\Gamma, x\colon A, \Gamma' \vdash x\colon A} \qquad \frac{\Gamma, x\colon A \vdash t\colon B}{\Gamma \vdash \lambda x\colon A.t\colon A \Rightarrow B}$$

$$\frac{\Gamma \vdash t\colon A \Rightarrow B \qquad \Gamma \vdash s\colon A}{\Gamma \vdash ts\colon B} \qquad \frac{\Gamma \vdash f\colon \Delta \qquad \Delta \vdash t\colon A}{\Gamma \vdash f * t\colon A}$$

(ii) On substitutions:

$$\frac{}{\Gamma \vdash \langle\rangle\colon \Gamma'}\ (\Gamma' \subseteq \Gamma) \qquad \frac{\Gamma \vdash f\colon \Delta \qquad \Gamma \vdash t\colon A}{\Gamma \vdash \langle f, t/x\rangle\colon \Delta, x\colon A}$$

$$\frac{\Gamma \vdash f\colon \Gamma' \qquad \Gamma' \vdash g\colon \Gamma''}{\Gamma \vdash f;g\colon \Gamma''}$$

The new syntax is best explained by relating the terms with explicit substitutions to terms with the usual implicit substitution of the simply-typed λ-calculus. The basic idea is that a substitution $\Gamma \vdash f\colon \vec{y}\colon \vec{B}$ [4] in the $\lambda\sigma$-calculus corresponds to a list of terms $\vec{t} = (t_1, \ldots, t_n)$ such that $\Gamma \vdash t_i\colon B_i$ in the λ-calculus. Moreover, a term $f * t$ in the $\lambda\sigma$-calculus corresponds to a term $t[t_i/x_i]$ in the λ-calculus. For example, the term $\langle t/x\rangle * x$ corresponds to the λ-term $x[t/x] \equiv t$, and the $\lambda\sigma$-term $\langle t/x, s/y\rangle * (xy)$ corresponds to the λ-term $(xy)[t/x, s/y] \equiv ts$. If we analyse the typing of the term $\langle t/x, s/y\rangle * (xy)$, we see that x is a variable and t a term of some function type $A \Rightarrow B$, and y a variable and s a term of type A. The

[4]We abbreviate a context $x_1\colon A_1, \ldots, x_n \cdots A_n$ to $\vec{x}\colon \vec{A}$. Similarly we write $t[\vec{s}/\vec{x}]$ for $t[s_1/x_1, \ldots, s_n/x_n]$.

substitution $\langle t/x, s/y\rangle$ corresponds to the lists of terms (t, s) and has the typing $\Gamma \vdash \langle t/x, s/y\rangle \colon (x \colon A \Rightarrow B, y \colon A)$. As this example demonstrates, the operation $*$ in the $\lambda\sigma$-calculus models the explicit substitution.

The operations ";" and "$\langle \_, \_\rangle$" model sequential and parallel composition of substitutions respectively. If $\Gamma \vdash f \colon (\vec{x} \colon \vec{A})$ and $\vec{x} \colon \vec{A} \vdash g \colon \Delta$ and f and g correspond to the lists $\vec{t}$ and $\vec{s}$ respectively, then the substitution $f; g$ corresponds to the list $(s_1[\vec{t}/\vec{x}], \ldots, s_m[\vec{t}/\vec{x}])$ and hence models sequential composition of the substitutions f and g. It is important to note that the existence of sequential composition in the $\lambda\sigma$-calculus implies that substitutions in the $\lambda\sigma$-calculus are not only lists of terms but more generally expressions $f_1; \cdots; f_n$ where each f_i is a list of terms. The expression $\langle f, t/x\rangle$ models parallel composition of substitutions: if $\Gamma \vdash t \colon A$, then the substitution $\langle f, t/x\rangle$ corresponds to the list of terms $(\vec{t}, t)$. The substitution $\langle\rangle$ acts not only as the identity substitution in the sense that the term $\langle\rangle * t$ corresponds to t but also as weakening: If $\Gamma \vdash t \colon A$ and Γ' is an extension of Γ then the term $\Gamma' \vdash \langle\rangle * t \colon A$ corresponds to the λ-term $\Gamma' \vdash t \colon A$ in the extended context Γ'.

2.2 Equations and Reductions

Now we turn to the equations-in-context, which are judgements $\Gamma \vdash f = g \colon \Delta$ and $\Gamma \vdash t = s \colon A$. This notion of equality is sometimes called *judgemental equality.* If a judgement $\Gamma \vdash f = g \colon \Delta$ can be stated for any contexts Γ and Δ such that $\Gamma \vdash f \colon \Delta$ implies $\Gamma \vdash g \colon \Delta$, we will write $f = g$ for $\Gamma \vdash f = g \colon \Delta$. Similarly, if a judgement $\Gamma \vdash t = s \colon A$ can be stated for any context Γ and type A such that $\Gamma \vdash t \colon A$ implies $\Gamma \vdash s \colon A$, we will write $t = s$ for this judgement. In section 5 we will relate this version of the calculus to a version with equations derived from reduction defined on raw terms.

Definition 3 ($\lambda\sigma$ Equations). *The equations of the $\lambda\sigma$-calculus with names are as follows:*

(i) Equations modelling (traditional) λ-calculus-reductions:

$$\begin{array}{rcl} (\lambda x \colon A.t)s & = & \langle\langle\rangle, s/x\rangle * t \\ \lambda x \colon A.tx & = & t \quad \textit{if } x \textit{ not free in } t \end{array}$$

(ii) Equations for substitutions (In the third rule, $y = x$ if y is neither a free variable nor a substitution variable in f, or y is a variable which

is neither a free variable of t and f nor a substitution variable in f) [5]*:*

$$
\begin{array}{rcll}
\langle f, t/x\rangle * x & = & t & (1)\\
\langle f, t/y\rangle * x & = & f * x \text{ if } x \neq y & (2)\\
f * \lambda x\colon A.t & = & \lambda y\colon A.\langle f, y/x\rangle * t & (3)\\
f * (ts) & = & (f * t)(f * s) & (4)\\
\langle\rangle; f & = & f & (5)\\
\langle\rangle * t & = & t & (6)\\
f; \langle g, t/x\rangle & = & \langle f; g, f * t/x\rangle & (7)\\
f; (g; h) & = & (f; g); h & (8)\\
f * (g * t) & = & (f; g) * t & (9)
\end{array}
$$

$$\frac{\Gamma \vdash f\colon \Delta = x_1\colon A_1, \ldots, x_n\colon A_n}{\Gamma \vdash f = \langle f * x_1/x_1, \ldots, f * x_n/x_n\rangle\colon \Delta}$$

The first two equations are the equations corresponding to β-and η-reduction in the λ-calculus respectively. The equation for the β-rule has a term with an explicit substitution on the right hand side rather than an implicit substitution as in the λ-calculus. This is the place where explicit substitutions are introduced during the reduction of λ-terms to normal form in order to make the delay of substitution possible. The equations (1)–(4) push substitutions over the constructors of λ-terms. The equation $\langle f, t/x\rangle * x = x$ is the one where the replacement of the term t for x actually takes place. The equations $f; (g; h) = (f; g); h$ and $f * (g * t) = (f; g) * t$ express associativity of substitution. The last equation for substitution expresses the fact that substitution is determined by its effect on variables. This equation can be thought of as an η-rule for the explicit substitutions. It is necessary for the definition of an extensional semantics, *e.g.*, a categorical semantics.

Definition 4 (Reduction Relations). *The (typed) reduction relations $\Gamma \vdash t \rightsquigarrow t'\colon A$ (over terms), and $\Gamma \vdash f \rightsquigarrow f'\colon \Delta$ (over substitutions) are defined by orienting the above equations from left to right.*

Again, if a reduction rule can be stated for any contexts Γ, Δ and types A such that $\Gamma \vdash f\colon \Delta$ implies $\Gamma \vdash f'\colon \Delta$ and $\Gamma \vdash t\colon A$ implies $\Gamma' \vdash t'\colon A$, we will write $f \rightsquigarrow f'$ and $t \rightsquigarrow t'$ respectively.

Before we investigate the meta-theoretical properties of this calculus, we examine α-equivalence in detail in the next section.

[5]the substitution variables in a substitution f are all variables x occurring in an expression $\langle g, t/x\rangle$; for a precise definition see Section 3.

3 α-equivalence

In this section we examine α-equivalence in a $\lambda\sigma$-calculus with names, which is more complex than in the λ-calculus.

When defining type theories, after the definition of the raw expressions one usually lists the binding operations and makes a remark like "We identify terms which are α-equivalent, *i.e.*, equal up to renaming of bound variables". The most commonly used way of justifying this informal statement goes back to Curry [CF58]. He defines substitution in a way that variable capture is avoided: α-equivalence can then be defined as the smallest congruence such that $\lambda x\colon A.t = \lambda y\colon A.t[y/x]$. Reduction preserves α-equivalence in the sense that if $(\lambda x\colon A.t)s$ is α-equivalent to $(\lambda y\colon A.t')s'$, so are $t[s/x]$ and $t'[s'/x']$.

A completely different approach was proposed by De Bruijn' [De 72], who replaces variables by numbers which indicate the nesting level of the variables, *i.e.*, the number of other λ-abstractions which are in the syntax tree on a path between the λ-abstraction binding this variable and the root of the tree. In this way α-equivalence becomes syntactic identity, *i.e.*, if one replaces variables by the appropriate de Bruijn-numbers, two terms with variables are α-equivalent iff the corresponding de Bruijn-terms are syntactically equal.

Here we transfer Curry's definition of α-equivalence to the $\lambda\sigma$-calculus. We aim to retain the results for the λ-calculus, in particular we want two expressions to be α-equivalent iff their corresponding de Bruijn-terms are equal, and reduction should preserve α-equivalence. The latter causes problems which are not apparent in the λ-calculus. If we define α-equivalence to be the smallest congruence such that $\lambda x\colon A.t = \lambda y\colon A.t[y/x]$, then β-reduction does not preserve α-reduction: the two terms $(\lambda x\colon A.t)s$ and $(\lambda y\colon A.t[y/x])s$ are α-equivalent, but the contracta $\langle s/x\rangle * t$ and $\langle s/y\rangle * t[y/x]$ are not.

Hence we have to define α-equivalence in such a way that terms like $\langle s/x\rangle * t$ and $\langle s/y\rangle * t[y/x]$ are α-equivalent. This means that the substitution operator $*$ acts as another binding operator. However, this is a different kind of binding from the one λ-abstraction provides: the substitution operator binds in any expression $f * t$ those variables in t where there is a term contained in f which is to be substituted in t. In the example $\langle s/x\rangle * t$, the variable x is bound by $*$. Note that the substitution operator $*$ does not indicate the scope nor the name of the variables that it binds.

To give the details of how free variables, bound variables and *substitution variables* (which are all those variables in a substitution f which are bound in a term $f * t$ or in a substitution $f; g$) we need two definitions.

Definition 5. *For raw terms t and raw substitutions f we define the sets $\mathtt{FV}(t)$ and $\mathtt{FV}(f)$ of free variables and $\mathtt{SV}(t)$ and $\mathtt{SV}(f)$ of substitution variables* *by induction over the structure of raw expressions:*

$$\begin{array}{llcl}
(x) & \mathtt{FV}(x) & = & \{x\} \\
(\lambda) & \mathtt{FV}(\lambda x\colon A.t) & = & \mathtt{FV}(t) \setminus \{x\} \\
(ts) & \mathtt{FV}(ts) & = & \mathtt{FV}(t) \cup \mathtt{FV}(s) \\
(f * t) & \mathtt{FV}(f * t) & = & \mathtt{FV}(f) \cup (\mathtt{FV}(t) \setminus \mathtt{SV}(f)) \\
(\langle\rangle) & \mathtt{FV}(\langle\rangle) & = & \emptyset \\
(\langle -, - \rangle) & \mathtt{FV}(\langle f, t/x \rangle) & = & \mathtt{FV}(f) \cup \mathtt{FV}(t) \\
(;) & \mathtt{FV}(f;g) & = & \mathtt{FV}(f) \cup (\mathtt{FV}(g) \setminus \mathtt{SV}(f))
\end{array}$$

$$\begin{array}{llcl}
(\langle\rangle) & \mathtt{SV}(\langle\rangle) & = & \emptyset \\
(\langle -, - \rangle) & \mathtt{SV}(\langle f, t/x \rangle) & = & \mathtt{SV}(f) \cup \{x\} \\
(;) & \mathtt{SV}(f;g) & = & (\mathtt{SV}(f) \setminus \mathtt{FV}(g)) \cup \mathtt{SV}(g)
\end{array}$$

A variable occurring in t is called bound *in the term t if it is not a free variable in t. A variable occurring in f is called* bound *in the substitution f if it is neither a free variable nor a substitution variable in f.*

In the λ-calculus Curry defines substitution before he defines α-equivalence. As the substitution has been made explicit, we only need to define *renaming* (*i.e.*, the replacement of one variable by another) as an operation in the meta-theory to state α-equivalence. This definition of renaming requires an auxiliary notion to change the name of the substitution variable x in $\langle f, t/x \rangle$ to y, *i.e.*, we define an operation $f\,\{y/x\}$, which satisfies

$$\langle f, t/x \rangle \, \{y/x\} = \langle f, t/y \rangle$$

Definition 6. *(i) We define the name-changing substitution $f\,\{y/x\}$ by induction over the structure of f as follows:*

$$\begin{array}{rcl}
\langle\rangle \, \{y/x\} & = & \langle\rangle \\
(f;g)\,\{y/x\} & = & \left\{ \begin{array}{ll} f;(g\,\{y/x\}) & \textit{if } x \in \mathtt{SV}(g) \\ f\,\{y/x\}\,;g & \textit{if } x \notin \mathtt{SV}(g) \end{array} \right. \\
\langle f, t/x \rangle \, \{y/x\} & = & \langle f, t/y \rangle \\
\langle f, t/z \rangle \, \{y/x\} & = & \langle f\,\{y/x\}\,, t/z \rangle \ \textit{if } x \neq y
\end{array}$$

(ii) We define the renaming of the variable x by the variable y in t or f by induction over the structure of raw expressions.

$$\begin{array}{rcl}
x[y/x] & = & y \\
z[y/x] & = & z \ \textit{if } z \neq x \\
(\lambda x\colon A.t)[y/x] & = & \lambda x\colon A.t \\
(\lambda z\colon A.t)[y/x] & = & \lambda w\colon A.t[w/z][y/x] (\ z \neq x) \\
(tu)[y/x] & = & (t[y/x])(u[y/x]) \\
(f * t)[y/x] & = & f\,\{z_i/y_i\}\,[y/x] * t[z_i/y_i][y/x] \\
\langle\rangle[y/x] & = & \langle\rangle \\
\langle f, t/z \rangle[y/x] & = & \langle f[y/x], t[y/x]/z \rangle \\
(f;g)[y/x] & = & (f\,\{z_i/y_i\})[y/x]; (g[z_i/y_i])[y/x]
\end{array}$$

In the second rule for λ-abstraction, w is equal to y if $x \notin \mathtt{FV}(t)$ or $y \notin \mathtt{FV}(s)$, otherwise w occurs neither free nor bound in t or s. In the rule for $f * t$, the variable z_i is equal to y_i if $y_i \notin \mathtt{FV}(s)$, otherwise it is a fresh variable. The same condition applies for the case $f; g$.

The definition of α-equivalence can now be stated.

Definition 7. *We define α-equivalence in the $\lambda\sigma$-calculus to be the smallest congruence relation on raw expressions including*

$$\begin{array}{rcl} \lambda x \colon A.t & \equiv_\alpha & \lambda y \colon A.t[y/x] \\ f * t & \equiv_\alpha & f\,\{y/x\} * t[y/x] \\ f; g & \equiv_\alpha & f\,\{y/x\}\,; g[y/x] \end{array}$$

The variable y is either x or it is not free in t, f and g nor is it contained in $\mathtt{SV}(f)$.

Next we examine the interaction between α-equivalence which is defined on raw expressions, and the judgements. For the typing judgements, $\Gamma \vdash t \colon A$ means there exists an α-equivalent term t' such that $\Gamma \vdash t' \colon A$ according to the rules presented in Section 2. A similar convention is adopted for all other judgements and for reduction on raw expressions. The next theorem justifies this convention.

Theorem 8. *Assume that t_1 and t_2 are two α-equivalent $\lambda\sigma$-terms, and assume that f_1 and f_2 are two α-equivalent substitutions.*

(i) If $\Gamma \vdash t_1 \colon A$, then also $\Gamma \vdash t_2 \colon A$, and similarly if $\Gamma \vdash f_1 \colon \Delta$, then also $\Gamma \vdash f_2 \colon \Delta$.

(ii) If $\Gamma \vdash t_1 = s \colon A$, then also $\Gamma \vdash t_2 = s \colon A$, and if $\Gamma \vdash f_1 = g \colon \Delta$, then $\Gamma \vdash f_2 = g \colon \Delta$.

(iii) If t and s are α-equivalent terms in the λ-calculus, then they are α-equivalent in the $\lambda\sigma$-calculus, too.

Proof. For (i) and (ii), show a suitable substitution lemma, *i.e.*, $\Gamma \vdash s \colon A$ and $\Gamma, x \colon A \vdash t \colon B$ implies $\Gamma \vdash t[s/x] \colon B$ and $\Gamma, x \colon A \vdash t = s \colon B$ and $\Gamma \vdash u \colon A$ implies $\Gamma \vdash t[u/x] = s[u/x]$. Now an induction over the derivation verifies the claim. (iii) is obvious from the definition of α-equivalence. $\square$

Remark In any implementation one wants to operate on representatives for the α-equivalence classes and avoid renaming (*i.e.*, change to a different representative of the same equivalence class) as much as possible. One often used way is to do the renaming during typechecking if necessary. In particular whenever a λ-abstraction is typechecked and the newly introduced bound variable is already in the context, a renaming operation is carried out along with typechecking the body of the λ-abstraction. In this way all bound variables are different from all free variables, and hence no renaming is necessary during reduction.

The $\lambda\sigma$-calculus with de Bruijn numbers has no variable names and hence also no α-equivalence. The intuition is that α-equivalence is in fact only a consequence of the existence of names and does not affect the type theory in any other way. The following theorem shows that this is indeed the case, by showing that equality modulo α-equivalence in the calculus with names and equality in the $\lambda\sigma$-calculus with de Bruijn numbers coincide. The translation from the $\lambda\sigma$-calculus with names into the $\lambda\sigma$-calculus with de Bruijn-numbers is defined by an induction over the derivation and replaces each variable x in a context $\Gamma, x\colon A, \Gamma'$ by the length $|\Gamma'|$ of the context Γ'.

Theorem 9. *(i) Assume that e_1 and e_2 are two $\alpha-$equivalent $\lambda\sigma$-expressions (either two terms or two substitutions). Then the translation of e_1 and e_2 into expressions with de Bruijn-numbers yields two syntactically equal expressions.*

(ii) Conversely, assume that s_1 and s_2 are two $\lambda\sigma$-terms with names such that their translations into de Bruijn numbers are syntactically equal, and that s_1 and s_2 are well-formed terms in the same context. Then s_1 and s_2 are α-equivalent.

(iii) If f_1 and f_2 are two substitutions with names such that their translations into de Bruijn numbers are syntactically equal and $\Gamma \vdash f_1\colon \Delta$ and $\Gamma \vdash f_2\colon \Delta$, then f_1 and f_2 are α-equivalent.

Proof. Induction over the structure of terms. The additional hypothesis in the second part ensures that the names of free variables that correspond to the same de Bruijn-number coincide. Otherwise the theorem is false: consider the two terms $x\colon A \vdash x\colon A$ and $y\colon A \vdash y\colon A$. Both are translated to the de Bruijn-number 0, but are not α-equivalent. □

The results of this section imply that Barendregt's variable convention can be adopted in the rest of this paper when we prove meta-theoretic properties. To be precise, we consider α-equivalent terms to be syntactically equal, and in the sequel we assume that all bound variables occur nowhere else in a given mathematical context (*e.g.*, neither as free variables as in $x(\lambda x\colon A.x)$ nor as substitution variables as in $\langle f, t/x\rangle * \lambda x\colon A.s$).

4 Confluence and Normalisation

This section investigates confluence and normalisation for the (equational) $\lambda\sigma$-calculus. We deduce confluence from the confluence of the simply-typed λ-calculus, using a modularity argument, first described by [Har89] and familiar under the name "interpretation method". The argument is well-known, here we just make an effort to present it in its generic form.

Definition 10 (Modularity Properties). *We call a translation* $\llbracket - \rrbracket$ *of the extended calculus into the confluent one* modular *if it satisfies the following properties:*

(i) If $t \rightsquigarrow s$ *in the extended system, then also* $\llbracket t \rrbracket \rightsquigarrow^* \llbracket s \rrbracket$.

(ii) For each term t *in the extended system we have* $t \rightsquigarrow^* \llbracket t \rrbracket$.

(iii) For each reduction $t \rightsquigarrow s$ *in the confluent system there exists a reduction sequence* $t \rightsquigarrow^* s$ *in the extended system.*

Confluence of the extended system can now be deduced as follows, where all arrows denote an arbitrary number of reductions: Each fork

$$\begin{tikzcd}
& t \arrow[dl] \arrow[dr] & \\
t_1 & & t_2
\end{tikzcd}$$

can be extended by assumption (ii) to a fork

$$\begin{tikzcd}
& t \arrow[dl] \arrow[dr] & \\
t_1 \arrow[d, dashed] & & t_2 \arrow[d, dashed] \\
\llbracket t_1 \rrbracket & & \llbracket t_2 \rrbracket
\end{tikzcd}$$

By assumption (i), this fork can be translated into a fork

$$\begin{tikzcd}
& \llbracket t \rrbracket \arrow[dl] \arrow[dr] & \\
\llbracket t_1 \rrbracket & & \llbracket t_2 \rrbracket
\end{tikzcd}$$

in the confluent calculus. The confluence property implies the existence of a term s such that

$$\begin{tikzcd}
& \llbracket t \rrbracket \arrow[dl] \arrow[dr] & \\
\llbracket t_1 \rrbracket \arrow[dr, dashed] & & \llbracket t_2 \rrbracket \arrow[dl, dashed] \\
& s &
\end{tikzcd}$$

By assumption *(iii)*, this implies that the fork in the extended calculus can be further extended to

$$\begin{tikzcd}
 & t \arrow[dl] \arrow[dr] & \\
t_1 \arrow[d, dashed] & & t_2 \arrow[d, dashed] \\
\llbracket t_1 \rrbracket \arrow[dr, dashed] & & \llbracket t_2 \rrbracket \arrow[dl, dashed] \\
 & s &
\end{tikzcd}$$

which is the claim.

Hardin's interpretation method [Har89] is based on the same idea: one has a set E and a reduction relation R together with a subsystem R' of R restricted to a subset E' of E. Then R is confluent if the restriction of R is confluent and terminating, and there is a map from E to E' which commutes with the reductions R and R'. In this case R is the whole reduction system, R' is are the σ-rules, and E' are the λ-terms. We only need the weaker hypothesis that each expression reduces via the substitution rules to a λ-term. Her additional condition amounts to requiring that the σ-rules are strongly normalising. This can be shown but is rather hard to establish.

Now we apply the argument given in Definition 10 to our version of the $\lambda\sigma$-calculus. The translation $\llbracket - \rrbracket$ works by "carrying out the substitutions", *i.e.*, $\llbracket \langle t_i/x_i \rangle * t \rrbracket = \llbracket t \rrbracket [\llbracket t_i \rrbracket / x_i]$. The precise definition is as follows.

Definition 11 (Translating away substitutions). *Define a translation $\llbracket - \rrbracket$ from well-formed $\lambda\sigma$-terms to simply-typed λ-terms and from well-formed $\lambda\sigma$-substitutions to lists of terms by induction over the derivation of $\lambda\sigma$-expressions, recursively, as follows.*

(i) On terms:

$$\begin{array}{rcl}
\llbracket \Gamma, x\colon A, \Gamma' \vdash x\colon A \rrbracket & = & x \\
\llbracket \Gamma \vdash \lambda x\colon A.t\colon A \Rightarrow B \rrbracket & = & \lambda x\colon A.\llbracket \Gamma, x\colon A \vdash t\colon B \rrbracket \\
\llbracket \Gamma \vdash ts\colon B \rrbracket & = & \llbracket \Gamma \vdash t\colon A \Rightarrow B \rrbracket \llbracket \Gamma \vdash s\colon A \rrbracket
\end{array}$$

$$\frac{\llbracket \Gamma \vdash f\colon (\vec{x}\colon \vec{A}) \rrbracket = \vec{t}}{\llbracket \Gamma \vdash f * t\colon A \rrbracket = \llbracket \vec{x}\colon \vec{A} \vdash t\colon A \rrbracket [\vec{t}/\vec{x}]}$$

(ii) On substitutions:

$$[\![\Gamma \vdash \langle\rangle \colon \vec{y} \colon \vec{B}]\!] \quad = \quad \vec{y}$$

$$\frac{[\![\Gamma \vdash f \colon \vec{x} \colon \vec{A}]\!] = \vec{t} \qquad [\![\Gamma \vdash t \colon A]\!] = t}{[\![\Gamma \vdash \langle f, t/x\rangle \colon (\vec{x} \colon \vec{A}, x \colon A)]\!] = (\vec{t}, t)}$$

$$\frac{[\![\Gamma \vdash f \colon \vec{x} \colon \vec{A}]\!] = (t_i)_i}{[\![f; g]\!] = [\![g]\!][\vec{t}/\vec{x}]}$$

We omit the typing information in the mapping $[\![-]\!]$ if the translation works for any context Γ and Δ and type A such that $\Gamma \vdash f \colon \Delta$ and $\Gamma \vdash t \colon A$.

Now we have to verify the modularity properties of the translation. All reduction rules except the β-rule $(\lambda x \colon A.t)s \rightsquigarrow \langle s/x\rangle * t$ and the η-rule $\lambda x \colon A.tx \rightsquigarrow t$ model explicit substitution. We call these rules σ-rules, and we denote a σ-reduction by $t \overset{\sigma}{\rightsquigarrow} s$. We expect the translation from the $\lambda\sigma$-calculus to the λ-calculus to map the redex and the contractum of a σ-reduction to the same λ-term. The next lemma shows this, and we obtain the modularity properties as a consequence.

Lemma 12. *Let t be a well-formed $\lambda\sigma$-term in context Γ, and let $\Gamma \vdash f \colon \vec{x} \colon \vec{A}$ be a well-formed substitution. A reduction on lists of λ-terms is defined as a reduction in any of the terms in the list.*

(i) If $t \overset{\sigma}{\rightsquigarrow} t'$, then $[\![t]\!] = [\![t']\!]$. Moreover, if $f \overset{\sigma}{\rightsquigarrow} f'$ then $[\![f]\!] = [\![f']\!]$.

(ii) We have $t \overset{\sigma}{\rightsquigarrow}{}^* [\![t]\!]$. Moreover, if $[\![f]\!] = (t_1, \ldots, t_n)$, then $f \overset{\sigma}{\rightsquigarrow}{}^* \langle t_1/x_1, \ldots, t_n/x_n\rangle$.

(iii) Let $\overset{\rho}{\rightsquigarrow}$ be either a β-or an η-reduction. If $t \overset{\rho}{\rightsquigarrow} t'$, then $[\![t]\!] \rightsquigarrow^* [\![t']\!]$. Similarly, if $f \overset{\rho}{\rightsquigarrow} f'$, then $[\![f]\!] \rightsquigarrow^* [\![f']\!]$.

(iv) Whenever $[\![t]\!] \rightsquigarrow u$, then also $t \rightsquigarrow^* u$. Similarly, if $[\![f]\!] \rightsquigarrow \vec{t}$, then $f \rightsquigarrow^* \langle \vec{t}/\vec{x}\rangle$.

Proof. (i) is shown by a case analysis. For (ii), one shows that the definition of implicit substitution can be turned into σ-reductions. (iii) is then an easy consequence of (ii). To obtain the reduction in (iv), do first a β-or η-reduction and then eliminate all explicit substitutions via (ii). ☐

Formalising the argument given before to establish confluence of the $\lambda\sigma$-calculus we obtain.

Theorem 13 (Typed confluence). *Let $\Gamma \vdash t \colon A$ be any well-formed $\lambda\sigma$-term. If $t \rightsquigarrow^* t_1$ and $t \rightsquigarrow^* t_2$, then there exists a well-formed $\lambda\sigma$-term $\Gamma \vdash u \colon A$ such that $t_1 \rightsquigarrow^* u$ and $t_2 \rightsquigarrow^* u$. Similarly, let $\Gamma \vdash f \colon \Delta$ be any well-formed substitution. If $f \rightsquigarrow^* f_1$ and $f \rightsquigarrow^* f_2$, then there exists a well-formed substitution $\Gamma \vdash g \colon \Delta$ such that $f_1 \rightsquigarrow^* g$ and $f_2 \rightsquigarrow^* g$.*

Proof. We first show confluence for terms. By parts (*i*) and (*iii*) of the Lemma 12, we obtain $[\![t]\!] \rightsquigarrow^* [\![t_1]\!]$ and $[\![t]\!] \rightsquigarrow^* [\![t_2]\!]$. The second part of Lemma 12 implies that $t \rightsquigarrow^* [\![t_1]\!]$ and $t \rightsquigarrow^* [\![t_2]\!]$. Because the simply-typed λ-calculus is confluent, there exists a term u such that $[\![t_1]\!] \rightsquigarrow^* u$ and $[\![t_2]\!] \rightsquigarrow^* u$. Now the fourth part of Lemma 12 yields $t_1 \rightsquigarrow^* u$ and $t_2 \rightsquigarrow^* u$. Now we turn to substitutions. By parts (*i*) and (*iii*) of the Lemma 12, we obtain $[\![\Gamma \vdash f\colon \Delta]\!] \rightsquigarrow^* [\![\Gamma \vdash f_1\colon \Delta]\!]$ and $[\![\Gamma \vdash f\colon \Delta]\!] \rightsquigarrow^* [\![\Gamma \vdash f_2\colon \Delta]\!]$. Assume $[\![\Gamma \vdash f_1\colon \Delta]\!] = (t_1, \ldots, t_n)$ and $[\![\Gamma \vdash f_2\colon \Delta]\!] = (u_1, \ldots, u_n)$. The second part of Lemma 12 implies that $f \rightsquigarrow^* \langle t_1/x_1, \ldots, t_n/x_n\rangle$ and $f \rightsquigarrow^* \langle u_1/x_1, \ldots, u_n/x_n\rangle$. Because the simply-typed λ-calculus is confluent and reduction on lists is defined componentwise, there exists a list of terms $g = s_1, \ldots, s_n$ such that $[\![\Gamma \vdash f_1\colon \Delta]\!] \rightsquigarrow^* g$ and $[\![\Gamma \vdash f_2\colon \Delta]\!] \rightsquigarrow^* g$. Now the fourth part of Lemma 12 yields $f_1 \rightsquigarrow^* \langle s_1/x_1, \ldots, s_n/x_n\rangle$ and $f_2 \rightsquigarrow^* \langle s_1/x_1, \ldots, s_n/x_n\rangle$. □

Normalisation also arises as a consequence of the modularity properties of the translation. Because the proof consists of giving an effective normalisation strategy, we obtain decidability of equality in the $\lambda\sigma$-calculus as a corollary.

Theorem 14 (Normalisation). *Every well-formed term t and substitution f of the $\lambda\sigma$-calculus has a normal form, which can be effectively computed. The normal form for a term is a normal λ-term, and the normal form for a substitution is a lists of normal λ-terms.*

Proof. Lemma 12 shows that the following strategy is a normalisation strategy: eliminate all explicit substitutions, then do a β-or η-reduction, and repeat this until no more β-or η-redexes are possible. □

Note that both normalisation and confluence are existential statements (there exists a term or a reduction sequence with certain properties). This makes it possible to establish confluence and normalisation by transferring constructions in the confluent (or normalising) calculus to the extended calculus. Strong normalisation is a universal statement, and hence every possible interaction between the reductions in the extended system and the ones in the confluent system has to be investigated. As Mellies [Mel95] shows, these interactions are powerful enough to destroy strong normalisation. As an example, he gives a λ-term which reduces to the identity but which admits a reduction sequence where a term t reduces to a term t' which contains t as a subterm. But it is possible to show that all reduction strategies that reduce an expression first to one in weak head-normal form (*i.e.*, substitution is pushed under λ-abstraction only if the λ-abstraction is the outermost constructor) lead only to finite sequences of reductions [Rit94b].

5 Reduction on Raw Terms

The main part of this section examines a typed calculus with reduction defined on raw terms. At the end we mention briefly untyped calculi.

5.1 Typed Calculi

Apart from the extensionality rule for substitution $\Gamma \vdash f \rightsquigarrow \langle f{*}x_i/x_i\rangle : \vec{x}\colon \vec{A}$, all reduction rules do not use typing information. Hence we omit this rule, and write $\rightsquigarrow_r$ for the notion of reduction on raw terms given by turning all reduction rules $\Gamma \vdash f \rightsquigarrow g\colon \Delta$ except $\Gamma \vdash f \rightsquigarrow \langle f * x_i/x_i\rangle : \vec{x}\colon \vec{A}$ into rules $f \rightsquigarrow_r u$, and all reduction rules $\Gamma \vdash t \rightsquigarrow s\colon A$ into rules $t \rightsquigarrow_r s$. For this intensional fragment, which suffices for the design of abstract machines, we show in this section that reduction based on raw terms and the reduction derived from equational judgements (see Section 2) coincide. The important properties for this proof are uniqueness of types and subject reduction. The same proofs that work for the simply-typed λ-calculus with reduction defined on raw expressions work also for the system with explicit substititution.

Lemma 15. *Assume t is a $\lambda\sigma$-term $\Gamma \vdash t\colon A$ and f a $\lambda\sigma$-substitution $\Gamma \vdash f\colon \Delta$.*

(i) All free variables of t and f are contained in the context Γ.

(ii) If also $\Gamma \vdash t\colon B$, then $A = B$. If also $\Gamma \vdash f\colon \Delta'$, then Δ and Δ' are subcontexts of some common context Δ''.

(iii) If x does not occur in the context Γ, then also $\Gamma, x\colon B \vdash t\colon A$ and $\Gamma, x\colon B \vdash f\colon \Delta$.

(iv) Assume that $\Gamma = \Gamma', x\colon A$. If x does not occur freely in t and f, then we have also $\Gamma' \vdash t\colon A$, and $\Gamma' \vdash f\colon \Delta$.

We get as an easy consequence that well-formedness is preserved under substitution.

Proposition 16 (Substitution lemma). *Assume $\Gamma, x\colon A \vdash t\colon B$, $\Gamma, x\colon A \vdash f\colon \Delta$ and $\Gamma \vdash s\colon A$. Then $\Gamma \vdash t[s/x]\colon B$ and $\Gamma \vdash f[s/x]\colon \Delta$.*

As in the simply-typed λ-calculus, this lemma suffices to prove subject reduction and the coincidence between reduction defined on raw expressions and the typed equality.

Proposition 17 (Subject Reduction). *Assume $\Gamma \vdash t\colon A$, $\Gamma \vdash f\colon \Delta$ and $t \rightsquigarrow_r t'$ and $f \rightsquigarrow_r f'$. Then we have also $\Gamma \vdash t'\colon A$, $\Gamma \vdash f'\colon \Delta$, and $\Gamma \vdash t = t'\colon A$ and $\Gamma \vdash f = f'\colon \Delta$.*

Now we turn to the confluence proof for the calculus based on reduction on raw terms. The proof follows the general outline established in the previous section but it does not work directly because the previous proof uses the fact (in Lemma 12, part (*ii*)) that every substitution reduces to a list of terms. This is no longer true if we use reduction on raw expressions: substitutions can no longer be reduced to lists of terms in general, but only to so-called *canonical forms*, *i.e.*, lists of terms with an additional weakening at the end. In particular, the substitution $\langle t/x\rangle;\langle\rangle$ is a normal form if t is a normal form. The precise definition is as follows:

(i) $\langle\rangle$ is canonical.

(ii) If f is canonical, so are $\langle f;\langle\rangle, t/x\rangle$ and $\langle f, t/x\rangle;\langle\rangle$.

We say a canonical substitution g *contains a term* t if t appears as a subterm in g. We define *structural equivalence* as the smallest equivalence relation relating two canonical substitutions g and h if

(i) $g \equiv \langle\rangle$ and $h \equiv \langle\rangle$.

(ii) $g \equiv \langle g', t/x\rangle$, $h \equiv \langle h', t/x\rangle$ and g' and h' are structurally equivalent.

(iii) $g \equiv \langle g', t/x\rangle$, $h \equiv \langle h', t/x\rangle;\langle\rangle$ and g' and h' are structurally equivalent.

Informally, if f and g are structurally equivalent, f and g differ only in the occurrences of weakening after lists of terms. For example, the substitution $\langle f, t/x\rangle;\langle\rangle$ is structurally equivalent to $\langle t/x\rangle$.

Because the translation of substitutions in the previous section is extensional (a substitution was always translated to a list of terms) we have to modify the translation: the idea is that we map a substitution $f;\langle\rangle$ into *two* lists $(\vec{t} \mid \vec{s})$ of terms with the intention that $\vec{t}$ denotes the part of f which is thrown away by the weakening $\langle\rangle$, and $\vec{s}$ denotes the part which is kept.

Definition 18. *Define a translation* $[\![-]\!]$ *from well-formed* $\lambda\sigma$*-terms to simply-typed* λ*-terms and from well-formed* $\lambda\sigma$*-substitutions to lists of terms by induction over the derivation of* $\lambda\sigma$*-expressions as follows.*

(i) On substitutions (In the first rule, let $\Gamma' = \vec{x}\colon \vec{A}$ *and* $\vec{y}$ *be the remaining variables of* Γ*):*

$$[\![\Gamma \vdash \langle\rangle : \Gamma']\!] = (\vec{y} \mid \vec{x}) \qquad \frac{[\![\Gamma \vdash f\colon \Delta]\!] = (\vec{t} \mid \vec{s})}{[\![\Gamma \vdash \langle f, t/x\rangle]\!] = (\vec{t} \mid (\vec{s}, t))}$$

$$\frac{[\![\Gamma \vdash f\colon \Gamma']\!] = (\vec{t} \mid \vec{s}) \qquad [\![\Gamma' \vdash g\colon \Gamma'']\!] = (\vec{u} \mid \vec{w})}{[\![\Gamma \vdash f; g\colon \Gamma'']\!] = (\vec{t}, \vec{u}[\vec{s}/\vec{y}] \mid \vec{w}[\vec{s}/\vec{y}])}$$

(ii) On terms:

$$\begin{array}{rcl}[\![x]\!] & = & x\\ [\![\lambda x\colon A.t]\!] & = & \lambda x\colon A.[\![t]\!]\\ [\![ts]\!] & = & [\![t]\!][\![s]\!]\end{array}$$

$$\frac{[\![\Gamma \vdash f\colon \Delta]\!] = (\vec{t} \mid \vec{s})}{[\![f * t]\!] = [\![t]\!][\vec{s}/\vec{x}]}$$

Again, we omit the typing information in the mapping $[\![-]\!]$ if the translation works for any context Γ and Δ and type A such that $\Gamma \vdash f\colon \Delta$ and $\Gamma \vdash t\colon A$.

Lemma 12 can now be stated as follows:

Lemma 19. *Let t be a well-formed $\lambda\sigma$-term in context Γ, and let $\Gamma \vdash f\colon \vec{x}\colon \vec{A}$ be a well-formed substitution. A reduction on lists of λ-terms is defined as a reduction in any of the terms in the list.*

(i) If $t \overset{\sigma}{\leadsto} t'$, then $[\![t]\!] = [\![t']\!]$. Moreover, if $f \overset{\sigma}{\leadsto} f'$ then $[\![f]\!] = [\![f']\!]$.

(ii) We have $t \overset{\sigma}{\leadsto}^* [\![t]\!]$. Similarly, if $[\![f]\!] = (\vec{t} \mid \vec{s})$, then $f \overset{\sigma}{\leadsto}^* g$ where g is canonical and the set of terms t occurring in subexpressions t/x in g is exactly the set of terms in $\vec{t}$ and $\vec{s}$.

(iii) Let $\overset{\rho}{\leadsto}$ be either a β-or an η-reduction. If $t \overset{\rho}{\leadsto} t'$, then $[\![t]\!] \leadsto^* [\![t']\!]$. Similarly, if $f \overset{\rho}{\leadsto} f'$, then $[\![f]\!] \leadsto^* [\![f']\!]$.

(iv) Whenever $[\![t]\!] \leadsto u$, then also $t \leadsto^* u$. Similarly, if $[\![f]\!] \leadsto (\vec{t} \mid \vec{s})$, then $f \leadsto^* g$, where g is canonical and $[\![g]\!] = (\vec{t} \mid \vec{s})$.

(v) If $f \leadsto_r^* g_1$ and $f \leadsto_r^* g_2$ and g_1 and g_2 are canonical, then g_1 and g_2 are structurally equivalent.

Proof. (i) is shown by a case analysis. For (ii), one shows that the definition of implicit substitution can be turned into σ-reductions. (iii) is then an easy consequence of (ii). To obtain the reduction in (iv), do first a β-or η-reduction and then eliminate all explicit substitutions via (ii). For (v) one defines by induction over the structure of f its skeleton, which is the position of weakening expressions $\langle\rangle$ in any canonical substitution to which f reduces and shows that this notion is invariant under reduction. □

As before, we obtain the main theorem:

Theorem 20 (Confluence on Raw Terms). *Let $\Gamma \vdash t\colon A$ be any well-formed $\lambda\sigma$-term. If $t \leadsto_r^* t_1$ and $t \leadsto_r^* t_2$, then there exists a well-formed $\lambda\sigma$-term $\Gamma \vdash u\colon A$ such that $t_1 \leadsto_r^* u$ and $t_2 \leadsto_r^* u$. Similarly, let $\Gamma \vdash f\colon \Delta$ be any well-formed substitution. If $f \leadsto_r^* f_1$ and $f \leadsto_r^* f_2$, then there exists a well-formed substitution $\Gamma \vdash g\colon \Delta$ such that $f_1 \leadsto_r^* g$ and $f_2 \leadsto_r^* g$.*

The result of good design now follows: the judgemental equality presentation of our $\lambda\sigma$-calculus with names is equivalent to its presentation based on reduction on raw terms.

Theorem 21. *The $\lambda\sigma$-calculus with judgemental equality is equivalent to the $\lambda\sigma$-calculus based on reduction on raw terms. Thus $\Gamma \vdash t = s\colon A$ if and only if $\Gamma \vdash t\colon A$, $\Gamma \vdash s\colon A$ and $t \leftrightarrow^* s$, where $\leftrightarrow^*$ is the equivalence relation generated by $\rightsquigarrow$, and similarly for substitutions.*

Proof. Same argument as in the previous section. $\square$

5.2 Untyped Calculi

The untyped λ-calculus arises by dropping all type information from the simply typed λ-calculus. As a consequence every raw term is meaningful. The untyped λ-calculus can also be obtained by adding a base type Ω together with the equality $\Omega \Rightarrow \Omega$ and considering all terms of type Ω. It is the addition of such a universal type Ω that generalises to the calculus with explicit substitution. The result is a calculus with explicit substitutions which has as the well-formed λ-terms exactly the terms of the untyped λ-calculus. However, not every expression is meaningful in this calculus: the substitution $\langle t/x, s/x\rangle$ is meaningless (and does not typecheck) as it assigns two terms to the same variable [6].

The confluence proof of the previous subsection still applies to this calculus with explicit substitutions and a universal base type. The reason is that the universal type does not affect the translation of expressions with explicit substitutions nor the reduction rules that eliminate explicit substitutions. In this way the confluence of the untyped λ-calculus can still be lifted to yield confluence of the calculus with explicit substitutions. Obviously, normalisation fails as any counterexample to normalisation in the untyped λ-calculus can be reproduced in the calculus with explicit substitutions.

6 Implementation Issues

The $\lambda\sigma$-calculus with de Bruijn numbers has been used as a basis for deriving Krivine's machine [ACCL91]. A semantical basis for that calculus and an abstract machine for a much more expressive extension of that calculus (the Calculus of Constructions) were given in an earlier paper by Ritter [Rit94a]. This paper shows that it is not essential to use de Bruijn numbers for the explanation of Krivine's machine but that a calculus with names, which is more readable, could have been used instead. Normally one does

[6]Note that this substitution has no equivalent in a calculus with de Bruijn-numbers. Indeed all raw expression in an untyped calculus with de Bruijn-numbers are meaningful and typecheck.

not implement η-reduction in abstract machines, as they do not have computational significance. In applications for theorem proving, η-expansions are often useful to ensure that normal terms of function type are actually λ-abstractions.

The first application we envisage for our $\lambda\sigma$-calculus is as the basis for a typed, explicit substitution version of Axford and Joy's Aladin abstract machine. Aladin[AJ96] is a minimal pure functional abstract machine, with lazy evaluation. There are no built-in primitive functions in Aladin; instead one has to specify the strictness of the arguments and the results of all primitive functions as part of their definitions. The explicit substitution calculus is well-suited to model the differences between the various evaluation mechanisms. We expect the necessary modifications to Aladin to produce a typed, explicit substitution abstract machine (called x-Aladin) to be relatively straightforward.

The second application we envisage is the use of the abstract machine corresponding to the $\lambda\sigma$-calculus, as a first-order theorem prover with terms. The explicit substitution models the instantiation of quantifiers. In this case we have to use names rather than de Bruijn numbers as the variable names are an integral part of the theorem prover.

7 Related Work

The original paper [ACCL91] on the $\lambda\sigma$-calculus left open two main syntactic problems. First, the authors did not prove that the typed $\lambda\sigma$-calculus with de Bruijn-numbers was strongly normalising. Second, they did not show confluence for open expressions, *i.e.*, expressions with metavariables for substitutions. Later on, Mellies [Mel95] showed that strong normalisation fails for the typed $\lambda\sigma$-calculus in a big way: small syntactic modifications are not enough to overcome this problem.

But since these two problems made it difficult to apply standard term rewriting technology to the calculus, they resulted in the definition of many calculi of explicit substitutions. With respect to the confluence, Curien et al. [CHJJ96] showed that confluence on open terms fails for the untyped $\lambda\sigma$-calculus. To obtain confluence they introduce a special syntactic construction, which describes the effect of pushing a substitution under a λ-abstraction. (They consider a version with de Bruijn-numbers, but the idea should work as well with a calculus with variables.)

Lescanne [Les94] introduces the $\lambda\upsilon$-calculus, which is essentially the calculus Curien et al. present without composition of substitution. This way he obtains a calculus which preserves strong normalisation and has the confluence property for open terms.

Bloo and Rose [BR95] present the λx-calculus which has fewer syntactic constructions than the $\lambda\upsilon$-calculus. The only constructor for substitutions is $\langle t/x \rangle$, with no parallel or sequential composition of substitutions. He uses his calculus, together with a global rule for eliminating garbage, to describe abstract machines and sharing.

Our presentation of a explicit substitution calculus has been driven by the correspondence with the semantics. Of the calculi discussed above, the original $\lambda\sigma$-calculus of Abadi et al. [ACCL91] is the only one with a closer correspondence to a categorical semantics. We see our calculus as a direct improvement of theirs. Moreover, in earlier work [Rit94b] Ritter showed that every reduction strategy that is used in abstract machines terminates, thus strong normalisation is not necessary for their design. He also established confluence for terms without metavariables, which is all one needs for the design of abstract machines. Thus, we are not interested in confluence for metavariables. Designing an abstract machine that is easy to prove correct with respect to a well-established semantics was our main goal: for that the calculus in this paper seems the appropriate one.

8 Conclusions

We examined choices for designing calculi with explicit substitutions. We presented our own version of a calculus of explicit substitutions, for the simply typed λ-calculus, for which we showed the equivalence between its version arising from semantical (equations-in-context) considerations and syntactic (reduction on raw term) ones. (This equivalence is crucial in establishing the correspondence between the semantics of the calculus and its implementations.) We discussed its typed and untyped variants and the names and de Bruijn flavours of the calculus. Also we proved all the necessary, standard, properties of our calculus. The proofs are also standard.

This calculus contains what we take to be the essential points of our approach of using categorical type theory to inform the implementation of abstract machines. Ritter's PhD thesis is perhaps a more impressive example of the same approach, dealing with the Calculus of Constructions. But the point of the paper is to show how "inevitable" this calculus is, given our original goals. This is to be contrasted with the multitude of other explicit substitution calculi. Also it was necessary to clarify the case of the simply typed-lambda-calculus, to modify it appropriately, to deal with the *linear lambda-calculus*. Linearity introduces several new challenges that we are tackling at the moment.

9 Acknowledgements

We thank Neil Ghani for valuable comments on earlier drafts of this paper. This research was supported in part by the EPSRC-grant GR/L28296 under the title "The eXplicit Subsitution Linear Abstract Machine".

References

[ACCL91] Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jaques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[AJ96] Tom Axford and Mike Joy. Aladin: An abstract machine for integrating functional and procedural programming. *Journal of Programming Languages*, 4:63–76, 1996.

[BR95] R. Bloo and K.H. Rose. Preesrvation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *Proc. CSN'95—Computer Science in the Netherlands*, pages 62–72, 1995.

[CF58] H. B. Curry and R. Feys. *Combinatory logic*, volume 1. North Holland, 1958.

[CHJJ96] P.-L. Curien, Th. Hardin, and J.-J.Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43:362–397, March 1996.

[Cur91] Pierre-Louis Curien. An abstract framework for environment machines (Note). *Theoretical Computer Science*, 82(2):389–402, 1991.

[De 72] N.G. De Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math*, 34:381–392, 1972.

[Geu93] Herman Geuvers. *Logics and Type Systems*. PhD thesis, University of Nijmegen, 1993.

[Har89] Thérèse Hardin. Confluence results for the pure strong categorical logic CCL. λ-calculi as subsystems of CCL. *Theoretical Computer Science*, 65:291–342, 1989.

[Les94] Pierre Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$: a journey through calculi of explicit substitutions. In *21st ACM Symposium on Principles of Programming Languages*, pages 60–69, Portland, Oregon, 1994.

[Mel95] P.-A. Mellies. Typed λ-calculi with explicit substitution may not terminate. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the Second Conference on Typed Lambda Calculi and Applications*, pages 328–334. Lecture Notes in Computer Science No. 902, Berlin, Heidelberg, New York, 1995.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[Rit94a] Eike Ritter. Categorical abstract machines for higher-order lambda calculi. *Theoretical Computer Science*, 136(1):125–162, 1994.

[Rit94b] Eike Ritter. Normalization for typed lambda calculi with explicit substitution. In *Proceedings of the 1993 Annual Conference of the European Association for Computer Science Logic*, pages 295–304. Lecture Notes in Computer Science No. 832, Berlin, Heidelberg, New York, 1994.

[Str89] Thomas Streicher. *Correctness and Completeness of a Categorical Semantics of the Calculus of Constructions.* PhD thesis, Universität Passau, June 1989.

[Tas93] A. Tasistro. Formulation of martin-löf's theory types with explicit substitutions. Technical report, Chalmers University, Dept. of Computer Science, May 1993.

A The $\lambda\sigma$-calculus with de Bruijn numbers

We present in this section the version of the $\lambda\sigma$-calculus with de Bruijn-numbers [De 72].

The $\lambda\sigma$-calculus with de Bruijn-numbers has four categories of raw expressions, where G is any ground type: contexts Γ, substitutions f, types A and terms t. They are given by the following BNF:

$$\begin{array}{rcl} \Gamma & ::= & [\,] \mid \Gamma \cdot A \\ f & ::= & \langle\rangle \mid \mathsf{Id} \mid f;f \mid \mathsf{Fst} \mid \langle f,t\rangle \\ A & ::= & G \mid A \Rightarrow A \\ t & ::= & f * t \mid 0 \mid \lambda A.t \mid tt \end{array}$$

We define $\mathsf{Fst}^{k+1};f$ recursively as $\mathsf{Fst};\mathsf{Fst}^k;f$, and $\mathsf{Fst};f$ as $\mathsf{Id};f$. Similarly, $\mathsf{Fst}^{k+1} * t$ is defined as $\mathsf{Fst}^k * (\mathsf{Fst} * t)$, and $\mathsf{Fst}^0 * t$ as $\mathsf{Id} * t$.

The judgements for well-formed expressions are as follows:

(i) Contexts:
All contexts are well-formed.

(ii) Substitutions

$$\frac{}{\Gamma \vdash \langle\rangle : [\,]} \qquad \frac{\Gamma \vdash f : \Gamma' \qquad \Gamma' \vdash g : \Gamma''}{\Gamma \vdash f;g : \Gamma''} \qquad \frac{}{\Gamma \vdash \mathsf{Id} : \Gamma}$$

$$\frac{}{\Gamma \cdot A \vdash \mathsf{Fst} : \Gamma} \qquad \frac{\Gamma \vdash f : \Delta \qquad \Gamma \vdash t : A}{\Gamma \vdash \langle f,t\rangle : \Delta \cdot A}$$

(iii) Types

All types are well-formed.

(iv) Terms

$$\frac{\Gamma \vdash f\colon \Delta \qquad \Delta \vdash t\colon A}{\Gamma \vdash f * t\colon A} \qquad\qquad \frac{}{\Gamma \cdot A \vdash 0\colon A}$$

$$\frac{\Gamma \cdot A \vdash t\colon B}{\Gamma \vdash \lambda A.t\colon A \Rightarrow B} \qquad\qquad \frac{\Gamma \vdash t\colon A \Rightarrow B \qquad \Gamma \vdash s\colon A}{\Gamma \vdash ts\colon B}$$

The reduction rules are as follows:

(i) Substitutions

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\mathsf{Id}; f & \rightsquigarrow & f & f; \mathsf{Id} & \rightsquigarrow & f & f;(g;h) & \rightsquigarrow & (f;g);h \\
f;\langle\rangle & \rightsquigarrow & \langle\rangle & \langle f,t\rangle;\mathsf{Fst} & \rightsquigarrow & f & f;\langle g,t\rangle & \rightsquigarrow & \langle f;g, f*t\rangle \\
\langle f;\mathsf{Fst}, f*0\rangle & \rightsquigarrow & f & \langle \mathsf{Fst}, 0\rangle & \rightsquigarrow & \mathsf{Id} & & &
\end{array}$$

(ii) Terms

$$\begin{array}{rcl@{\qquad}rcl}
f*(ts) & \rightsquigarrow & (f*t)(f*s) & f*(g*t) & \rightsquigarrow & (f;g)*t \\
\mathsf{Id}*t & \rightsquigarrow & t & f*(\lambda A.t) & \rightsquigarrow & \lambda A.\langle \mathsf{Fst}; f, 0\rangle * t \\
(f*(\lambda A.t))s & \rightsquigarrow & \langle f,s\rangle * t & \langle f,t\rangle * 0 & \rightsquigarrow & t
\end{array}$$

The translation from a $\lambda\sigma$-calculus-calculus with names to a $\lambda\sigma$-calculus-calculus with de Bruijn numbers is given next.

Definition 22. *By induction over the structure of derivations, define an expression $(e)^n$ in the $\lambda\sigma$-calculus with de Bruijn numbers as follows, where Γ' denotes the length of the context Γ':*

(i) On terms:

$$\begin{array}{rcl}
(\Gamma, x\colon A, \Gamma' \vdash x\colon A)^n & = & |\Gamma'| \\
(\Gamma \vdash \lambda x\colon A.t^n) & = & \lambda.(\Gamma, x\colon A \vdash t)^n \\
(\Gamma \vdash ts\colon A)^n & = & ((\Gamma \vdash t\colon B \Rightarrow A)^n)((\Gamma \vdash s\colon B)^n)n \\
(\Gamma \vdash f * t\colon A)^n & = & (\Gamma \vdash f\colon \Delta)^n * (\Delta \vdash t\colon A)^n
\end{array}$$

(ii) On substitutions

$$\begin{array}{rcl}
(\Gamma \vdash \langle\rangle : \vec{y}\colon \vec{B})^n & = & \langle i_1, \ldots, i_n\rangle \\
(\Gamma \vdash \langle f, t/x\rangle\colon \Delta, x\colon A)^n & = & \langle (\Gamma \vdash f\colon \Delta)^n, (\Gamma \vdash t\colon A)^n\rangle \\
(\Gamma \vdash f;g\colon \Delta)^n & = & (\Gamma \vdash f\colon \Delta')^n; (\Delta' \vdash g\colon \Delta)^n
\end{array}$$

In the first rule for substitutions the number i_j is the place of the variable y_j in the context Γ.

This calculus with de Bruijn-numbers is equivalent to the calculus with names: the substitution

$$A_n, \dots, A_1, A \vdash \mathsf{Fst}\colon A_n, \dots, A_1$$

corresponds to the substitution

$$x_n\colon A_n, \dots, x_1\colon A_1, x\colon A \vdash \langle x_i/x_i\rangle\colon x_n\colon A_n, \dots, x_1\ ,$$

and the de Bruijn-number n corresponds to the variable

$$x_k\colon A_k, \dots, x_n\colon A_n, \dots, x_1\colon A_1 \vdash x_n\colon A_n\ .$$