Electronic Notes in Theoretical Computer Science 36 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume36.html 23 pages

CINNI – A Generic Calculus of Explicit Substitutions and its Application to λ-, ς- and π-Calculi

Mark-Oliver Stehr

SRI International
Computer Science Laboratory
Menlo Park, CA 94025, U. S. A.
Email: stehr@csl.sri.com

Abstract

We approach the general problem of representing higher-order languages, that are usually equipped with special variable binding constructs, in a less specialized first-order framework such as membership equational logic and the corresponding version of rewriting logic. The solution we propose is based on CINNI, a new calculus of explicit substitutions that makes use of a term representation that contains both the standard named notation and de Bruijn's indexed notation as special subcases. The calculus is parametric in the syntax of the object language, which allows us to apply it to different object languages such as λ-calculus, Abadi and Cardelli's object calculus (ς-calculus) and Milner's calculus of communicating mobile processes (π-calculus). As a practical result we obtain executable formal representations of these object languages in Maude with a representational distance close to zero.

Key words: Higher-Order Languages, Explicit Substitutions, Logical Frameworks, Rewriting Logic, Maude, Lambda-Calculus, Sigma-Calculus, Pi-Calculus

1 Introduction

A common feature of higher-order languages is that essential entities which operate on data receive a first-class status so that variables can range over these entities. For instance, in higher-order logics or higher-order functional programming languages, variables can range over functions; in object-oriented programming languages, variables can range over objects (which can contain methods); and in languages for mobile processes, variables can range over processes or channels (which are references to processes). In order to express the essential entities directly as terms in the language, higher-order languages typically provide a syntax for abstractions, such as abstractions for functions,

methods or processes in the examples above. Abstractions are essentially binding constructs that bind the free variables in the abstracted term with the intention to be instantiated later by means of substitution.

Currently there are two major approaches to representing an object language with binding constructs in first-order frameworks such as membership equational logic [28,10] or rewriting logic [27]. We distinguish betweeen representations with names and representations based on de Bruijn indices. Representations with names have the very desirable feature to be close to the object language, but a major drawback is that they lack a canonical way to treat names and potential name clashes. Calculi based on de Bruijn indices have the advantage of a canonical representation, but they are more abstract, since information about names is not represented so that the gap between the object language and its representation is considerable.

In addition to the representation of the object language, another important issue is capture-free substitution, the main operation that is needed for terms of the object language. For a definition of substitution on terms with binding constructs, it turns out to be useful to treat substitutions as first-class citizens, since substitutions usually have to be adjusted as they are propagated through the term they are applied to. Since substitutions receive the same formal status as terms, the calculi that deal with substitution in this way are called *explicit substitution calculi*.

Research in this area has led to a rich collection of calculi (overviews and comparisons can be found in [25,35,8,30,22]) with quite different properties and motivations. Most of this research is focused on λ-calculus (notable exceptions are [9] and [33]) and the use of explicit substitutions to express β-reduction in terms of a more primitive concept. Among the motivations for using explicit substitutions we can find the following: the need for a rigorous and simple explanation of capture-free substitution [4], the quest for a notion of computation that is more fine-grained and more implementation-oriented than standard β-reduction [14,2], the interest in analysis of evaluation strategies and efficiency of computation [19], the application of first-order techniques to higher-order languages [18], and the use of algorithms that operate on incomplete terms such as higher-order unification [17], type checking/inference [36], and proof synthesis [31,30].

Our primary motivation to propose a new calculus of explicit substitutions in this paper is to obtain a first-order representation of terms with binding and capture-free substitution that is as close as possible to the standard named notation. Beyond that, we want this calculus to be executable, in the sense that it can be executed using a rewriting engine such as Maude [12,11], and furthermore we are interested in a general solution, in the sense that the theory does not restrict ourselves to a particular object language. More precisely, the objective of this paper is to develop a calculus of names and explicit substitutions that takes names seriously and completely removes the gap between the object language and its representation (often called representational dis-

tance) without losing the possibility of canonical representations. A solution that is closely related to de Bruijn's representation [16] but has been developed independently by Berkling [6,7] is a unification of named and indexed notation. Despite of its advantages, which have also been recognized more recently in [34], it is an unconventional representation that has not attracted much attention so far. Therefore, we devote the next section to an introduction and motivation of Berkling's representation, which will serve as a basis for the CINNI substitution calculus.

2 Indexed Names and Named Indices

Consider the standard treatment of binding constructs, say in the context of first-order logic, where α-equivalent terms, i.e. terms that can be transformed into each other by consistent renaming of bound variables, are identified, i.e. not distinguished for essential parts of the metatheory. An obvious first step towards a named representation is to give up this identification that we also refer to as α-equality. Unfortunately, this rather naive approach leads to the following difficulty that we refer to as *accidental hiding*.

Consider for instance the formula

$$\forall X.(A \wedge \forall Y.(B \Rightarrow \forall X.C(X)))$$

for distinct names X and Y. Assume the subformula $C(X)$ contains X free. Then each free occurrence of X in $C(X)$ is *captured* by the inner $\forall$ quantifier, so that the name bound by the outermost $\forall$ quantifier is hidden from the viewpoint of $C(X)$. Indeed there is no way to refer to the outermost $\forall$ quantifier within $C(X)$.

Hence, we are faced with the following problem: a calculus without α-equality is not only less abstract, which is an unavoidable consequence of giving up identification by α-conversion, but also, depending on the (accidental) choice of names, visibility of (bound) variables may be restricted. It is important to emphasize that visibility is not restricted in the original calculus with α-equality, since renaming can be performed *tacitly* at any time.

Clearly, the phenomenon of hiding that occurs in the example above is undesirable [1], because it is not present in the original calculus with α-equality. It is merely an accident caused by giving up identification by α-conversion without adding a compensating flexibility to the language.

This suggests tackling this general problem by migrating to a more flexible syntax, where we express a binding constraint by annotating each name X with an index $i \in \mathrm{N}$, written X_i, that indicates how many X-binders should be skipped before we reach the one that X_i refers to. For instance, we write

$$\forall X.(A \wedge \forall Y.(B \Rightarrow \forall X.C(X_0)))$$

[1] Of course, in general hiding is important but it is not an issue of binding; it should be treated independently.

to express that X_0 is bound by the inner $\forall$, and

$$\forall X.(A \wedge \forall Y.(B \Rightarrow \forall X.C(X_1)))$$

meaning that X_1 is bound by the outermost $\forall$. To make the language a conservative extension of the traditional notation, we can identify X and X_0.

The use of indexed names is equivalent to a representation introduced by Berkling [6,7] in the context of λ-calculus [2] which is why we refer to the notation based on indexed names also as Berkling's notation. As indicated by the example above we use Berkling's representation not (only) for λ-calculus but as the core syntax of CINNI, the *Calculus of Indexed Names and Named Indices* which is generic in the sense that it can be instantiated for a wide range of object languages.

Obviously, there is some similarity to a notation based on de Bruijn indices [16], but notice that there is an essential difference: the index m in the occurrence X_m is *not* the number of binders to be skipped; it states that we have to skip m binders for the particular name X, *not* counting binders for other names. Still a formal relationship to de Bruijn's notation can be established: if we restrict ourselves to terms that contain only a single name X, then we can replace each X_i by the index i without loss of information and we arrive at de Bruijn's purely indexed notation. [3] In other words, if we restrict the available names to a single one, we obtain de Bruijn's notation as a very special case. In this sense, Berkling's representation can be formally seen as a proper generalization of de Bruijn's notation. Pragmatically, however, the relationship to de Bruijn's syntax plays only a minor role, since a typical user will exploit the dimension of names much more than the dimension of indices. Hence, in practice the notation can be used as a standard named notation, with the additional advantage that accidental hiding and weird renamings [4] are avoided.

The pragmatic advantage of Berkling's notation is that it can be used to reduce the distance between the formal system and its implementation: it can be directly employed by the user who wants to think in terms of names, so that the need for a translation between an internal representation (e.g., using de Bruijn indices) and a user-friendly syntax (e.g., using ordinary names) disappears completely.

Usually, this translation between an internal and an external representation is not considered to be a problem, and indeed, in the case of terms where all parts are known or accessible, solutions are straightforward. However, even in this case this gap is not desirable; consider, for example, a tactic-based theorem prover where the user is confronted with an internal representation which reflects the theory only in a very indirect way. More seriously, the

[2] An indexed variable X_i is represented in Berkling's representation as $\#^i X$ where $\#$ is the so-called unbinding operation.

[3] With the slight difference that de Bruijn's indices start at 1 instead of 0.

[4] See the discussion on weird renaming in the next section.

translation between internal and external representations becomes impossible, or at least requires certain restrictions, as soon as we use terms containing metavariables, holes or placeholders, which are useful for many applications including unification algorithms and representation of incomplete proofs.

3 Explicit Substitutions

In the previous section we discussed Berkling's first-order representation for expressions which contains the conventional named notation as well as de Bruijn's indexed notation as special cases. The most important operation to be performed on such terms represented in this way is capture-free substitution. Therefore, we now present the CINNI substitution calculus, a first-order calculus that can be seen as an (operational) refinement of an external (i.e. metalevel) substitution function such as the one given in [7].

Strictly speaking, CINNI is a family of explicit substitution calculi, parameterized by the syntax of the language we want to represent. For a language L given by its syntax we denote the corresponding instantiation of CINNI by CINNI_L. The syntax defines the term constructors together with their binding constraints which are expressed by associating a binary relation to each term constructor f as follows: We say that f binds argument i in argument j iff for each term $f(P_1, \ldots, P_n)$ of the object language, P_i is a name and this name is bound in the subterm P_j. So each P_i is either a term or a name (names are not considered to be object language terms).

As an example we use the untyped λ-calculus to present the concrete instantiation CINNI_λ of the substitution calculus. CINNI_λ-terms are generated by the syntax

$$X_m \quad | \quad (M\ N) \quad | \quad [X]M$$

with the constraint that $[\_]\_$ binds argument 0 in argument 1.

As a motivation for the substitution calculus given below, consider the following example of a β-reduction step in the traditional λ-calculus with distinct names X and Y, again taking names literally, i.e., not presupposing identification by α-conversion:

$$(([X][Y]X)Y) \rightarrow_\beta [Z]Y$$

Clearly, the bound variable Y must be renamed to Z, a name different from Y, to avoid capturing of the free variable Y. Unfortunately, there is no canonical choice if all names should be treated as being equal. We call this phenomenon *weird renaming* of bound variables. It is actually a combination of two undesirable effects: (1) names that have been carefully chosen by the user have to be changed, and (2) the enforced choice of a new name collides with the right of names to be treated as equal citizens. These effects are avoided in the CINNI calculus. It is specified by the first-order equational theory given below. Indeed, the only operation assumed on names is equality.

More formally, we assume that the syntax of the object language is given

by a signature in membership equational logic which introduces a sort of names (assumed to be nonempty), a sort of natural numbers (with zero 0 and successor +1 as the only operations), and a sort of object language terms. In addition, we have a constructor for variables, i.e. terms of the form X_m, as well as additional constructors for terms together with their binding constraints and optional structural equations (see below).

To present the actual calculus we need to extend the notion of term by explicit substitutions. To this end, we introduce a sort of substitutions together with the following operators: In addition to the two basic kinds of substitutions, namely *simple* substitutions $[X{:=}M]$ and *shift* substitutions $\uparrow_X$, substitutions can be *lifted* using $\Uparrow_X(S)$, where the variable S ranges over substitutions. The application $S\ M$ of an explicit substitution to a term is again a term. Now CINNI_L has the signature just described and the following equations:

$$[X{:=}M]\ X_0 \ = \ M \qquad \text{(FVar)}$$
$$[X{:=}M]\ X_{m+1} \ = \ X_m \qquad \text{(RVarEq)}$$
$$[X{:=}M]\ Y_n \ = \ Y_n \ \text{ if } \ X \neq Y \qquad \text{(RVarNEq)}$$

$$\uparrow_X X_m \ = \ X_{m+1} \qquad \text{(VarShiftEq)}$$
$$\uparrow_X Y_n \ = \ Y_n \ \text{ if } \ X \neq Y \qquad \text{(VarShiftNEq)}$$

$$\Uparrow_X(S)\ X_0 \ = \ X_0 \qquad \text{(FVarLift)}$$
$$\Uparrow_X(S)\ X_{m+1} \ = \uparrow_X \ (S\ X_m) \qquad \text{(RVarLiftEq)}$$
$$\Uparrow_X(S)\ Y_n \ = \uparrow_X \ (S\ Y_n) \ \text{ if } \ X \neq Y \qquad \text{(RVarLiftNEq)}$$

For each syntactic constructor f of L we add a *syntax-specific equation*

$$S\ f(P_1, \ldots, P_n) = f(\Uparrow_{P_{j_{1,1}}}(\ldots \Uparrow_{P_{j_{1,m_1}}}(S))\ P_1, \ldots, \Uparrow_{P_{j_{n,1}}}(\ldots \Uparrow_{P_{j_{n,m_n}}}(S))\ P_n)$$

where $j_{i,1}, \ldots, j_{i,m_i}$ are all the arguments (necessarily of the name sort) that f binds in argument i (necessarily of term sort). If P_k is of name sort we identify $S\ P_k$ and P_k (abuse of notation).

Often the terms of the object language are not just freely generated by the syntactic constructors, but are subject to additional *structural equations*. Admissible equations in this paper are the laws of associativity, commutativity and identity for binary operators, and we assume throughout the paper that structurally equivalent terms are identified. If f is a binary operator with identity e we add a condition to the syntax-specific equation above which ensures that none of the P_i is equal to e (cf. the specifications of ς-calculus and π-calculus in Section 5). This ensures that the left hand sides of valid instances of syntax-specific equations do not overlap and is also needed to ensure termination of the corresponding rewrite system.

The syntax-specific equations are the only equations that depend on the syntax

of L. For instance, CINNI$_\lambda$ has the following syntax-specific equations:

$$S\ (MN)\ =\ (SM)(SN) \qquad \text{(App)}$$
$$S\ ([X]M)\ =\ [X](\Uparrow_X(S)\ M) \qquad \text{(Lambda)}$$

The equations of CINNI can be justified by the following algebraic substitution semantics: a substitution S is interpreted as a function from variables to terms. Application of substitution is interpreted as function application. The substitutions $[X{:=}M]$, $\uparrow_X$ and $\Uparrow_X(S)$ are then uniquely defined by the equations above. Finally, substitutions are extended from variables to terms by the syntax-specific equations. Each time a substitution moves into a new scope it has to be adjusted using a lift substitution.

CINNI is not only an equational calculus with an algebraic semantics, but as usual for explicit substitution calculi it can be equipped with an operational semantics by regarding equations as rewrite rules. We refer to the resulting term rewrite system as the *CINNI rewrite system* and we introduce the following relations: The *characteristic relation* of the equations of CINNI is denoted by $\Rightarrow_{\mathrm{S}}$, i.e. $M \Rightarrow_{\mathrm{S}} N$ holds iff $M = N$ is a valid instance of one of the equations. The *rewrite relation* induced by $\Rightarrow_{\mathrm{S}}$, i.e. its compatible closure, is denoted by $\rightarrow_{\mathrm{S}}$. The induced equivalence on terms is denoted by $=_{\mathrm{S}}$.

We can now define the explicit substitution version of the *β-rule* by

$$([X]N)M \Rightarrow_{\mathrm{B}} [X{:=}M]N.$$

Notice that weird renaming of bound variables as in the previous example is avoided with the new notion of β-reduction. For instance, we have

$$(([X][Y]X_0)Y_0) \rightarrow^*_{\mathrm{SB}} ([Y]Y_1)$$

where $\rightarrow_{\mathrm{SB}}$ denotes the compatible closure of $\Rightarrow_{\mathrm{S}} \cup \Rightarrow_{\mathrm{B}}$. Notice also that we do not view application-specific computation rules as a part of the substitution calculus (which is CINNI$_\lambda$ in this case). The substitution calculus only depends on the syntax of the object language.

As another application of substitution, consider *renaming of a bound variable* X by $\bullet$ as in the following explicit substitution version of *α-reduction*

$$([X]N) \Rightarrow_{\mathrm{A}} ([\bullet][X{:=}\bullet] \uparrow_\bullet N) \ \text{ if } \ X \neq \bullet$$

where $\bullet$ is an arbitrary but fixed name. Using this rule and the rules for explicit substitutions, every CINNI$_\lambda$ term can be reduced to a nameless *α-normal form* which is essentially its de Bruijn index representation. α-reduction is a new concept that becomes expressible due to the use of a unified syntax with indices and names.

Just as Berkling's notation contains de Bruijn's notation as a very special case, the instantiation of CINNI for the λ-calculus reduces to the calculus

$\lambda\upsilon$ of explicit substitutions proposed by Pierre Lescanne [25,26,4], but only in the degenerate case where we restrict the set of names to a singleton set. It is noteworthy that $\lambda\upsilon$ is the smallest known indexed substitution calculus enjoying good theoretical properties like confluence and preservation of strong normalization. It seems that its simplicity is inherited by CINNI although in practice the dimension of names will be much more important than the dimension of indices. Hence, we tend to think of CINNI more as a substitution calculus with names than as one with indices.

4 Metatheoretic Properties of CINNI

In this secion we give a number of important operational properties concerning both the CINNI calculus in isolation, and the composition of CINNI with application-specific rules such as the explicit substitution version of the β-rule in the case of CINNI$_\lambda$. The present section generalizes the results of [25] for $\lambda\upsilon$ in two orthogonal dimensions.

The first dimension is the scope of applicability:

(i) Instead of considering a fixed object language such as λ-calculus, we consider an arbitrary object language given by a syntax L with binders.

(ii) Instead of considering a fixed set of computation rules, such as β-reduction in λ-calculus, we allow arbitrary computation rules R as long as they satisfy a certain well-formedness property.

The second dimension of generalization is concerned with the representation of the object language:

(i) The de Bruijn index representation is generalized to a richer representation with indexed names.

(ii) The substitution calculus and its properties are generalized accordingly.

Due to space limitations proofs of the metatheoretic properties cannot be given in this section, but detailed proofs can be found in the extended version of this paper [37].

4.1 CINNI in Isolation

We first consider the CINNI rewrite system in isolation, instantiated for the syntax L of an arbitrary object language.

Definition 4.1 *We define the following two functions on terms to express iterated shift and lift substitutions:*

$$\begin{array}{ll} \uparrow_\emptyset(M) = M & \Uparrow_\emptyset(S)(M) = S\ M \\ \uparrow_{\overline{Y},X}(M) = \uparrow_{\overline{Y}}(\uparrow_X M) & \Uparrow_{\overline{Y},X}(S)(M) = \Uparrow_{\overline{Y}}(\Uparrow_X(S))(M) \end{array}$$

Here and in the following we use variables $\overline{X}$, $\overline{Y}$, $\overline{Z}$, $\overline{U}$, $\overline{V}$, $\overline{W}$ to range over lists of names, $\emptyset$ denotes the empty list and comma denotes concatenation. Furthermore, we use $|\overline{Y}|_X$ to denote the number of occurrences of X in $\overline{Y}$.

4.1.1 Operational Properties

Confluence of CINNI can be easily established by transforming the rewrite system into an equivalent orthogonal, i.e. left-linear and non-overlapping, rewrite system without conditions. This is done by replacing each possibly conditional equation by all its valid instances obtained by instantiating X and Y by concrete names. Notice that the resulting system becomes infinite if the set of names is infinite.

Theorem 4.2 *The relation $\rightarrow_{\mathrm{S}}$ is confluent.*

Some mathematical evidence that CINNI_λ is a nontrivial generalization of $\lambda\upsilon$ and its metatheory seems to be given by the observation that the proof of strong normalization in [4,26], which makes use of elementary interpretations [24], cannot be applied to CINNI_λ. Indeed the problematic equations are (RVarLiftNEq) and the syntax-specific equations which make it unlikely that a proof of strong normalization can be obtained by a modified elementary interpretation. Furthermore, the syntax-specific equations seem to prevent us from giving a proof based on recursive path orderings. Hence, we pursue a different approach which makes use of the fact that in orthogonal term rewrite systems all maximal computations that do not erase redices are essentially equivalent. In particular, computations that only perform innermost reductions are in a certain sense representative computations and can be used to prove strong normalization as already observed in [32]. Hence the following theorem can be proved by exhibiting an innermost-reduction strategy and showing that it always terminates.

Theorem 4.3 *The relation $\rightarrow_{\mathrm{S}}$ is strongly normalizing.*

As a consequence of this theorem, each CINNI_L-term M and each CINNI_L-substitution S has a unique *substitution normal form* which will be denoted by $\mathrm{NF}_{\mathrm{S}}(M)$ and $\mathrm{NF}_{\mathrm{S}}(S)$, respectively. Notice that $\mathrm{NF}_{\mathrm{S}}(M)$ does not contain any substitutions, otherwise one of the rules could be applied.

4.1.2 Equational Properties

The following induction lemma provides a tool for proving certain equivalences of the form $S_l^L \ \ldots \ S_1^L \ M =_{\mathrm{S}} S_r^R \ \ldots \ S_1^R \ M$, and it has been used in the proofs of the subsequent lemmas which state basic equational properties of the CINNI-calculus.

Lemma 4.4 (Induction Lemma)
Let $S_1^L \ \ldots \ S_l^L$ and $S_1^R \ \ldots \ S_r^R$ be substitutions. Define

$$L(M) = \ S_l^L \ \ldots \ S_1^L \ M, \quad \Uparrow_{\overline{W}}(L)(M) = \Uparrow_{\overline{W}}(S_l^L) \ \ldots \ \Uparrow_{\overline{W}}(S_1^L) \ M,$$
$$R(M) = \ S_r^R \ \ldots \ S_1^R \ M, \quad \Uparrow_{\overline{W}}(R)(M) = \Uparrow_{\overline{W}}(S_r^R) \ \ldots \ \Uparrow_{\overline{W}}(S_1^R) \ M.$$

In order to prove

$$\Uparrow_{\overline{W}}(L)(M) =_{\mathrm{S}} \Uparrow_{\overline{W}}(R)(M) \text{ and in particular } L(M) =_{\mathrm{S}} R(M)$$

for all M, $\overline{W}$, it is sufficient to show that

$$\Uparrow_{\overline{W}}(L)(X_k) =_{\mathrm{S}} \Uparrow_{\overline{W}}(R)(X_k)$$

for all X, k, and $\overline{W}$.

Lemma 4.5 (Simplification I)

(i) $\Uparrow_{\overline{Y}}(S)\ X_m =_{\mathrm{S}} X_m$ *if* $m < |\overline{Y}|_X$,
(ii) $\Uparrow_{\overline{Y}}(S)\ X_m =_{\mathrm{S}} \uparrow_{\overline{Y}} S\ X_{m-i}$ *if* $m \geq |\overline{Y}|_X = i$.

Lemma 4.6 (Simplification II)

(i) $\Uparrow_{\overline{Y}}(\uparrow_Z)\ X_m =_{\mathrm{S}} X_m$ *where* $Z \neq X$,
(ii) $\Uparrow_{\overline{Y}}(\uparrow_Z)\ X_m =_{\mathrm{S}} \uparrow_Z X_m$ *if* $m \geq |\overline{Y}|_X$,
(iii) $\Uparrow_{\overline{Y}}(\uparrow_Z)\ X_m =_{\mathrm{S}} \uparrow_Z X_m$ *if* $Z \notin \overline{Y}$,
(iv) $\Uparrow_{\overline{W}}(\Uparrow_{\overline{Y}}(\uparrow_Z))\ M =_{\mathrm{S}} \Uparrow_{\overline{W}}(\uparrow_Z)\ M$ *if* $Z \notin \overline{Y}$,
(v) $\Uparrow_{\overline{Y}}(\uparrow_Z)\ M =_{\mathrm{S}} \uparrow_Z M$ *if* $Z \notin \overline{Y}$.

Lemma 4.7 (Simplification III)

(i) $\Uparrow_{\overline{Y}}([Z{:=}N])\ X_m =_{\mathrm{S}} X_m$ *where* $Z \neq X$,
(ii) $\Uparrow_{\overline{Y}}([Z{:=}N])\ \Uparrow_{\overline{Y}}(\uparrow_Z)\ X_m =_{\mathrm{S}} X_m$,
(iii) $\Uparrow_{\overline{Y}}([Z{:=}N])\ \Uparrow_{\overline{Y}}(\uparrow_Z)\ M =_{\mathrm{S}} M$.

In each of the following lemmas we explicitly state a strong equivalence, that can be established using the induction lemma, followed by weaker consequences that are typically sufficient for most purposes.

Lemma 4.8 (Shift Shift Reordering)

(i) $\Uparrow_{\overline{W}}(\uparrow_Z)\ \Uparrow_{\overline{W}}(\uparrow_Y)\ M =_{\mathrm{S}} \Uparrow_{\overline{W}}(\uparrow_Y)\ \Uparrow_{\overline{W}}(\uparrow_Z)\ M$,
(ii) $\uparrow_Z \uparrow_Y M =_{\mathrm{S}} \uparrow_Y \uparrow_Z M$.

Lemma 4.9 (Lift Lift Reordering)

(i) $\Uparrow_{\overline{W}}(\Uparrow_Z(\Uparrow_Y(S)))\ M =_{\mathrm{S}} \Uparrow_{\overline{W}}(\Uparrow_Y(\Uparrow_Z(S)))\ M$,
(ii) $\Uparrow_Z(\Uparrow_Y(S))\ M =_{\mathrm{S}} \Uparrow_Y(\Uparrow_Z(S))\ M$.

Lemma 4.10 (General Shift Reordering)

(i) $\Uparrow_{\overline{W},Y}(S)\ \Uparrow_{\overline{W}}(\uparrow_Y)\ M =_{\mathrm{S}} \Uparrow_{\overline{W}}(\uparrow_Y)\ \Uparrow_{\overline{W}}(S)\ M$,
(ii) $\Uparrow_Y(S)\ \uparrow_Y M =_{\mathrm{S}} \uparrow_Y S\ M$,
(iii) $\Uparrow_{\overline{Y}}(S)\ \uparrow_{\overline{Y}} M =_{\mathrm{S}} \uparrow_{\overline{Y}} S\ M$.

Lemma 4.11 (Simple Substitution Reordering)

(i) $\Uparrow_{\overline{W}}(S)\ \Uparrow_{\overline{W}}([Z{:=}N])\ M =_{\mathrm{S}} \Uparrow_{\overline{W}}([Z{:=}S\ N])\ \Uparrow_{\overline{W},Z}(S)\ M$,

(ii) $S\ [Z{:=}N]\ M =_{\mathrm{S}} [Z{:=}S\ N]\ \Uparrow_Z(S)\ M$,
(iii) $[Y{:=}L]\ [Z{:=}N]\ M =_{\mathrm{S}} [Z{:=}[Y{:=}L]\ N]\ \Uparrow_Z([Y{:=}L])\ M$.

4.2 Preservation of Confluence

In a typical application context CINNI_L is extended by extra equations such as β-reduction in the case of CINNI_λ and a natural question is whether the resulting system remains confluent if confluence of the system without explicit substitutions has already been established. So let R be the set of (possibly conditional) equations of the form $M = N$ `if` C. Here M and N are terms possibly containing equational logic variables. Assume furthermore that $\Rightarrow_{\mathrm{R}}$ is the characteristic relation defined by these extra equations on terms. We denote the compatible closures of $\Rightarrow_{\mathrm{R}}$ and $\Rightarrow_{\mathrm{S}} \cup \Rightarrow_{\mathrm{R}}$ on terms and substitutions by $\rightarrow_{\mathrm{R}}$ and $\rightarrow_{\mathrm{SR}}$, respectively. We also define a relation $\Rrightarrow_{\mathrm{R}}$ on pure, i.e. substitution-free, terms by $M \Rrightarrow_{\mathrm{R}} M''$ iff there is an M' such that $M \rightarrow_{\mathrm{R}} M'$ and $M'' = \mathrm{NF}_{\mathrm{S}}(M')$. We use $\Rrightarrow_{\mathrm{R}}$ as a reference system that operates on a level of abstraction without (observable) explicit substitutions.

Definition 4.12 (Well-Formedness)
We say that the set R of equations is well-formed *iff for each equation in R both sides of are terms of the (object language) term sort, there are no substitution applications occurring in the left hand side, and for all terms M,N, and substitutions S,*

$$M \Rightarrow_{\mathrm{R}} M' \text{ implies } \exists N' : \mathrm{NF}_{\mathrm{S}}(S\ M) \Rightarrow_{\mathrm{R}} N' =_{\mathrm{S}} S\ M'$$

Intuitively, well-formedness expresses compatibility between application of rules and application of substitutions modulo the equations of the substitution calculus. It is interesting to note that well-formedness excludes quite different kinds of ill-formed rules. For a concrete object language well-formedness can typically be verified using the lemmas given before.

Lemma 4.13 (Well-Formedness Lemma) *Assume that R is well-formed. Then for all terms M,N, $n \in \mathrm{N}$, and substitutions $S_1, \ldots, S_n$,*

$$M \Rightarrow_{\mathrm{R}} M' \text{ implies } \exists N' : \mathrm{NF}_{\mathrm{S}}(S_n\ \ldots S_1\ M) \Rightarrow_{\mathrm{R}} N' =_{\mathrm{S}} S_n\ \ldots S_1\ M'$$

It is remarkable that the proof of the confluence theorem and in particular the proof of the subsequent projection lemma, both given in [26,4] for $\lambda\upsilon$, generalize to our setting without any difficulties (see [37]).

Lemma 4.14 (Projection) *Assume that R is well-formed. Then*

(i) $M \rightarrow_{\mathrm{R}} M'$ *implies* $\mathrm{NF}_{\mathrm{S}}(M) \Rrightarrow^*_{\mathrm{R}} \mathrm{NF}_{\mathrm{S}}(M')$ *and*
(ii) $S \rightarrow_{\mathrm{R}} S'$ *implies* $\mathrm{NF}_{\mathrm{S}}(S) \Rrightarrow^*_{\mathrm{R}} \mathrm{NF}_{\mathrm{S}}(S')$.

Lemma 4.15 (Hardin's Interpretation Technique [2,20])
Let S and R be relations on some set T. Assume S is confluent and terminating and $\mathrm{NF}_S(x)$ is the normal form of x w.r.t. S. Assume R_S is a relation on

$\mathrm{NF}_S(T)$ *with* $R_S \subseteq (S \cup R)^*$ *and* $x\ R\ y$ *implies* $\mathrm{NF}_S(x)\ {R_S}^*\ \mathrm{NF}_S(y)$. *Then confluence of* R_S *implies confluence of* $(S \cup R)^*$.

As a direct consequence of the previous two lemmas we obtain:

Theorem 4.16 (Preservation of Confluence)
If R is well-formed and $\Rightarrow_{\mathrm{R}}$ is confluent then $\rightarrow_{\mathrm{SR}}$ is confluent.

A noteworthy point is that confluence is *not* reduced to local confluence via Newman's Lemma. Therefore the previous theorem can also be applied in cases where $\Rightarrow_{\mathrm{R}}$ is not strongly normalizing.

5 Applications

In this section we illustrate the use of CINNI to obtain membership equational logic [28,10] specifications of λ-calculus, Abadi and Cardelli's ς-calculus [1] as well as a rewriting logic [27] specification of Milner's π-calculus [29]. These calculi are interesting, since they have different binding constructs and quite different equational theories. In addition, the π-calculus does not only have an equational theory that specifies process congruence, but there are also rewrite rules to specify the operational semantics in terms of a transition system equipped with an algebraic structure. Both process congruence equations and transition rules make use of substitutions. We show how appropriate instantiations of CINNI can be used in all three cases to obtain formal and executable first-order representations of these languages in Maude [12,11]. Furthermore, in each of these examples we give application-specific confluence results that can be obtained using the general results stated earlier. In each example the well-formedness condition can be verified using the lemmas in Section 4.1.2. Since we aim at a unique model in each case, the specifications we give in the following should all be interpreted under the initial semantics.

5.1 Higher-Order Functions: Lambda-Calculus

For the representation of untyped λ-calculus we use the predefined sort `Qid` to represent names. The following signature defines representations of variables and λ-terms as elements of the sorts `Var` and `Trm`, respectively:

```
sorts Var Trm .
op _{_} : Qid Nat -> Var .
subsort Var < Trm .
op __ : Trm Trm -> Trm .
op [_]_ : Qid Trm -> Trm .

vars n m : Nat .  vars X Y Z : Qid .  vars M N : Trm .
```

Here `X{m}` is the representation of a variable, i.e. an indexed name, while `(M N)` and `[X] M` represent application and abstraction, respectively.

The instantiation of CINNI to the syntax of λ-terms, that is CINNI$_\lambda$, is given below. `[X := M]`, `[shift X]`, and `[lift X S]` represent simple substitutions, shift substitutions, and lifted substitutions, respectively, and `__` is substitution application.

```
sort Subst .  var S : Subst .
op [_:=_] : Qid Trm -> Subst .
op [shift_] : Qid -> Subst .
op [lift__] : Qid Subst -> Subst .
op __ : Subst Trm -> Trm .

eq  ([X := M] (X{0})) = M .
eq  ([X := M] (X{suc(m)})) = (X{m}) .
ceq ([X := M] (Y{n})) = (Y{n}) if X =/= Y .
eq  ([shift X] (X{m})) = (X{suc(m)}) .
ceq ([shift X] (Y{n})) = (Y{n}) if X =/= Y .
eq  ([lift X S] (X{0})) = (X{0}) .
eq  ([lift X S] (X{suc(m)})) = [shift X] (S (X{m})) .
ceq ([lift X S] (Y{m})) = [shift X] (S (Y{m})) if X =/= Y .

eq  S (M N) = (S M) (S N) .
eq  S ([X] M) = [X] ([lift X S] M) .
```

Now the B-rule is given by

```
eq  (([X] M) N) = [X := N] M . *** (B)
```

As an example we define the standard combinators:

```
op I K S : -> Trm .
eq I = ['z] 'z{0} .
eq K = ['u] ['v] 'u{0} .
eq S = ['x] ['y] ['z] (('x{0} 'z{0})('y{0} 'z{0})) .
```

The fact that `(S K K) == I` can be verified by reduction:

```
red (S K K) .
--- rewrites: 47
--- result Trm: ['z]'z{0}
```

Theorem 5.1 *The rewrite relation induced by the above specification CINNI$_\lambda$+ B is confluent.*

In λ-calculus with the standard named notation, the best confluence result that is possible is confluence modulo α-conversion, since weird renaming has to be compensated by a notion of equivalence weaker than identity. The fact that the previous theorem states confluence literally, i.e. in its strongest conceivable form, is noteworthy and is made possible by the canonical treatment of names in the CINNI calculus.

5.2 Object-Orientation: Sigma-Calculus

As another application of CINNI we give a first-order representation of the ς-calculus [1]. Just as the λ-calculus can be seen as the most basic model for higher-order functional programming, the object-calculus provides a basic model for object-oriented programming. In contrast to [23] which presents a version of the ς-calculus with explicit substitutions that is based on the standard named representation with α-equality, the following specification is first-order and immediately executable using Maude.

In the ς-calculus an object is seen as a set of attributes, where each attribute has a label and a method. The labels are required to be unique inside an object, since they are used to invoke and update the object's attributes. Below labels, attributes, sets of attributes, methods, objects, and object variables are represented as elements of sorts `Lab`, `Attr`, `Attrs`, `Meth`, `Obj`, and `Var`, respectively.

```
sorts Obj Attr Attrs Meth Lab Var .
var L : Lab .  vars O B O' : Obj .  vars M M' : Meth .
var A : Attr .  var AA : Attrs .  vars X Y Z : Qid .

op _=_ : Lab Meth -> Attr .

op emptyAttrs : -> Attrs .
subsort Attr < Attrs .
op _,_ : Attrs Attrs -> Attrs [assoc comm id: emptyAttrs].

op _{_} : Qid Nat -> Var .
subsorts Var < Obj .
op {_} : Attrs -> Obj .
op method[_]_ : Qid Obj -> Meth .

op _._ : Obj Lab -> Obj .
op _._:=_ : Obj Lab Meth -> Obj .
```

Notice that in contrast to the λ-calculus we make use of structural equations (expressed by operator attributes in Maude) in the definition of the syntax of the ς-calculus.

The last two operators are invokation and update, written as `O . L` and `O . L := M`, respectively. In order to define these operations we need a notion of substitution. So we instantiate the CINNI calculus to obtain the following specification of CINNI$_\varsigma$:

```
sort Subst .  var S : Subst .  vars n m : Nat .
op [_:=_] : Qid Obj -> Subst .
op [shift_] : Qid -> Subst .
op [lift__] : Qid Subst -> Subst .
```

```
op __ : Subst Obj   -> Obj .   op __ : Subst Meth  -> Meth .
op __ : Subst Attr  -> Attr .  op __ : Subst Attrs -> Attrs  .

eq  ([X := O] (X{O})) = O .
eq  ([X := O] (X{suc(m)})) = (X{m}) .
ceq ([X := O] (Y{n})) = (Y{n}) if X =/= Y .
eq  ([shift X] (X{m})) = (X{suc(m)}) .
ceq ([shift X] (Y{n})) = (Y{n}) if X =/= Y .
eq  ([lift X S] (X{O})) = (X{O}) .
eq  ([lift X S] (X{suc(m)})) = [shift X] (S (X{m})) .
ceq ([lift X S] (Y{m})) = [shift X] (S (Y{m})) if X =/= Y .

eq  S (L = M) = (L = S M) .
eq  S emptyAttrs = emptyAttrs .
ceq S (AA, AA') = (S AA), (S AA') if
    AA =/= emptyAttrs and AA' =/= emptyAttrs .
eq  S ({AA}) = {S AA} .
eq  S (method [X] B) = method [X] ([lift X S] B) .
eq  S (O . L) = (S O) . L .
eq  S (O . L := M) = (S O) . L := S M .
```

Notice that __ is overloaded, i.e. we have a substitution application operator for each syntactic kind. As a slight optimization we eliminated the application of substitutions to labels. Another noteworthy point is that we added a condition to the syntax-specific equation for application of substitutions to attribute sets in order to avoid nontermination in the operational semantics.

Now method update (MU) and method invokation (MI) can be defined as follows:

```
eq {L = M, AA} . L := M' = {L = M', AA} . *** (MU)
eq {L = method [X] B, AA} . L =
   [X := {L = method [X] B, AA}] B .      *** (MI)
```

Theorem 5.2 *The rewrite relation induced by the above specification CINNI$_\varsigma$+ MI+ MU is confluent modulo the structural equations.*

5.3 Mobile Processes: Pi-Calculus

Quite different from the λ-calculus and the ς-calculus is the π-calculus [29], a calculus of communicating mobile processes. Here the mobility refers to the fact that processes can exchange names of channels and use them for subsequent communications so that the logical communication topology can evolve dynamically. In the λ-calculus and the ς-calculus the user is mainly interested in the result of evaluation. Since both calculi are confluent, such a result is unique if it exists and can be found by reduction. In the π-calculus a term is a collection of possibly interconnected processes, and as a particular case of

a reactive system the overall dynamic behaviour is relevant. Typically such systems are nonterminating and nondeterministic, and the states that such a system can reach should be clearly distinguished from each other rather than being identified by equations. So instead of using just membership equational logic as in the λ-calculus and the ς-calculus, the capabilities of rewriting logic to specify dynamic systems are exploited in the present example. Nevertheless, the equational part, which includes in particular the equationally defined notion of substitution and the process congruence of the π-calculus, will still play a major role.

The π-calculus distinguishes between channels and process terms which we represent by elements of the sorts `Chan` and `Trm`, respectively. There is an associative, commutative parallel composition `P|Q` defined on process terms with the empty process `nil` as identity element. Given a process `P`, the term `out CX < CY > . P` represents a process that sends the channel (name) `CY` via the channel `CX` and then continues like `P`. Notice that this is not a binding construct, whereas the construct `in CX [Y] P` binds the name `Y` in `P`. It represents a process that receives a channel name via channel `CX` and then behaves like `P` with the channel variable `Y` (that is `Y{0}`) substituted by the received channel name. Another binding construct is `new [X] P` which declares `X` to be a local channel w.r.t. `P` and is also called the hiding construct. The full π-calculus has additional constructs for choice and replication, but the fragment introduced here will be sufficient to explain the application of CINNI in this context. The syntax of this fragment is given by the following specification:

```
sorts Chan Trm .
op _{_} : Qid Nat -> Chan .
op nil : -> Trm .
op _|_ : Trm Trm -> Trm [assoc comm id: nil] .
op new[_]_ : Qid Trm -> Trm .
op out_<_>._ : Chan Chan Trm -> Trm .
op in_[_]._ : Chan Qid Trm -> Trm .

vars X Y Z : Qid .  vars CX CY CZ : Chan .  vars M N P Q : Trm .
```

The instantiation of CINNI for the π-calculus syntax, that is CINNI_π, is given next. Since the π-calculus variables can only range over channels, it is sufficient to have substitutions of variables by channels. As in the specification of the ς-calculus, `__` is overloaded, so that substitutions can operate on channels and on process terms.

```
sort Subst .  var S : Subst .  vars n m : Nat .
op [_:=_] : Qid Chan -> Subst .
op [shift_] : Qid -> Subst .
op [lift__] : Qid Subst -> Subst .
op __ : Subst Chan -> Chan .  op __ : Subst Trm -> Trm .
```

```
eq  ([X := CZ] (X{0})) = CZ .
eq  ([X := CZ] (X{suc(m)})) = (X{m}) .
ceq ([X := CZ] (Y{n})) = (Y{n}) if X =/= Y .
eq  ([shift X] (X{m})) = (X{suc(m)}) .
ceq ([shift X] (Y{n})) = (Y{n}) if X =/= Y .
eq  ([lift X S] (X{0})) = (X{0}) .
eq  ([lift X S] (X{suc(m)})) = [shift X] (S (X{m})) .
ceq ([lift X S] (Y{m})) = [shift X] (S (Y{m})) if X =/= Y .

eq  S nil = nil .
ceq S (M | N) = (S M) | (S N) if N =/= nil and M =/= nil .
eq  S (out CX < CZ > . M) = out (S CX) < S CZ > . (S M) .
eq  S (in CX [ Y ] . M) = in (S CX) [ Y ] . ([lift Y S] M) .
eq  S (new [X] M) = new [X] ([lift X S] M) .
```

The process congruence is generated by the structural equations for `nil` and `_|_` given above and the following equations NEW1, NEW2 and NEW3 involving the hiding construct. NEW2 and NEW3 are constrained by conditions to avoid nontermination. Here we presuppose a total order `_<_` on names.

```
eq  new [X] nil = nil .                                  *** (NEW1)
ceq (new [X] P) | Q = new [X] (P | [shift X] Q)
    if P =/= nil and Q =/= nil .                         *** (NEW2)
ceq new [X] new [Y] P = new [Y] new [X] P if Y < X .  *** (NEW3)
```

This completes the part of the specification that defines the process congruence. As stated by the following theorem the specification is confluent, and it is therefore not only logically but also operationally appropriate to use equations.

Theorem 5.3 *The rewrite relation induced by the above specification CINNI$_\pi$+ NEW1+ NEW2+ NEW3 is confluent modulo the structural equations.*

Now communication of two parallel processes via a channel `CX` can be expressed by the following rule that models an atomic interaction in which `(out CX < CZ > . P)` sends the channel name `CZ` to `(in CX [ Y ] . Q)`.

```
rl  [communicate] : (out CX < CZ > . P) | (in CX [ Y ] . Q) =>
                    P | [Y := CZ] Q  .
```

By restricting the application of the compatibility rule of rewriting logic to the process constructors `nil`, `_|_` and `new[_]_` we make sure that, in conformance with the π-calculus, communication never takes place inside `(out CX < CZ > . P)` or `(in CX [ Y ] . Q)`. In Maude such nonstandard congruence properties are reflected operationally by using a suitable execution strategy.

Related approaches to representing the π-calculus in rewriting logic, that make use of de Bruijn index based substitution calculi, are given in [39] and [21]. In

fact, the former is more closely related to the presentation given before, since it uses a π-calculus version of $\lambda\upsilon$ so that the representation of syntax and the substitution subcalculus arise as a special case of CINNI in the sense explained earlier. On the other hand [39] covers the full π-calculus with choice and replication, and uses a representation of the operational semantics exploiting rules and strategies. A CINNI version of the full π-calculus can be obtained by a straightforward adaptation of these rules.

5.4 Further Applications

Another application of CINNI that should be placed in the context of higher-order logic and type theory is presented in [36], where CINNI is instantiated to the family of *pure type systems* [5,38]. Pure type systems generalize the λ-cube [3] and are considered to be of key importance, since their generality and simplicity makes them an ideal basis for representing and implementing higher-order logics.

Last but not least we would like to point out that the CINNI calculus is currently being applied in the design and implementation of a proof assistant for OCC, the *open calculus of constructions*, an extension of the calculus of constructions [13] that incorporates equational logic as a computational sublanguage. OCC supports conditional equations and conditional assertions together with an operational semantics based on conditional rewriting modulo equations. In fact, we have developed an experimental Maude specification of OCC that makes use of Maude's reflective capabilities to evaluate higher-order equational specifications and programs with reasonable efficiency.

6 Three Orthogonal Research Directions

To place our work into the context of research conducted by other authors we describe in the following what could be visualized as a (partial) *cube of explicit substitution calculi*, namely an informal classification of three orthogonal research directions concerned with explicit substitution calculi and their (potential) combinations. We restrict our attention here to first-order calculi. We consider the minimal and well-investigated substitution calculus $\lambda\upsilon$ as a reference point and we distinguish three orthogonal directions of research:

(i) Generalizing the object language by metavariables to represent not only closed but also open terms (enrichment).

(ii) Generalizing the fixed syntax and computation rules of the object language from λ-calculus to languages with arbitrary syntax and computation rules (parameterization).

(iii) Generalizing the underlying representation based on de Bruijn's indices to Berkling's representation allowing for an explicit representation of names (enrichment).

Although we are not concerned with the direction (1) in this paper, this was historically the first direction investigated. The idea to deal with open terms, i.e. terms with metavariables, is already present in $\lambda\sigma$ [2] and $\lambda_{\Uparrow}$ [15] from which $\lambda\upsilon$ can be derived as a subcalculus [25]. Conversely, $\lambda\upsilon$ can be extended by a composition operator and corresponding rules to obtain $\lambda_{\Uparrow}$. Our main motivation for using $\lambda\upsilon$ as a starting point is its minimality and its logical completeness in the sense that standard properties of composition are inductive consequences. By definition the terms of $\lambda\upsilon$ (and CINNI) do not contain metavariables, hence confluence means ground-confluence in this context. If terms with metavariables are considered, confluence does not hold anymore in the core calculus. The extension $\lambda_{\Uparrow}$ is confluent on open terms, but does not preserve strong normalization (even on closed terms), a property that has, however, been established for $\lambda\upsilon$ [26,4]. On the other hand, confluence on open terms is not needed for many important applications such as execution (see Section 5) or type checking (see [36]). This motivates our choice to use the minimal calulus $\lambda\upsilon$, which enjoys good metatheoretic properties on terms without metavariables, as a reference point in this paper.

Direction (2) has been pursued first in the context of combinatory reduction systems [9] and later as a generalization of $\lambda_{\Uparrow}$ in [33]. The work [9] uses combinatory reduction systems which are not first-order and belong to a level of abstraction higher than the one we deal with in this paper. The first-order approach presented in [33] is interesting and closely related to our work in the sense that it investigates a condition similar to what we called well-formedness. However, the author does not aim at *preservation* of confluence results in the sense we presented them in this paper. Instead, he investigates a number of sufficient conditions to *ensure* confluence. Also preservation of strong normalization, which we consider as an important future extension of our work, is impossible in [33] since already $\lambda_{\Uparrow}$ does not have this property. Another important point is that [33] does not consider explicit names. Loosely speaking, [33] investigates the directions (1) and (2) whereas we deal with the directions (2) and (3). Of course, an interesting question is whether (1), (2) and (3) can still be combined in a reasonable way if we disregard the problem of strong normalization.

Direction (3) appears to be a natural direction in the context of explicit substitution calculi that has not been explored so far. Instead of de Bruijn's representation it makes use of Berkling's representation [6,7] as a basis of an explicit substitution calculus. In view of the clear advantages, it is surprising that the CINNI calculus seems to be the first explicit substitution calculus based on this representation. It might appear that the generalization is rather straightforward, but this is definitely not true for the metatheory as indicated by our strong normalization result which cannot be proved by just reusing the techniques of [4,26]. This indicates that CINNI is a nontrivial generalization that deserves a careful study. Furthermore, we think that the difference becomes even more challenging if other calculi such as $\lambda\sigma$ and $\lambda_{\Uparrow}$ are used as a

starting point for the direction (3).

7 Conclusions

The main contribution of this paper is the introduction of a generic first-order calculus of explicit substitutions that combines the advantages of both named and indexed notation in a natural way. The fact that our approach contains $\lambda\upsilon$ as a special case is of great help for developing the theory, since most of the statements and proofs of [26,4] can be fruitfully generalized to our setting. In the future we plan to further exploit this connection: As we did for confluence, we will try to generalize and modularize the proof of preservation of strong normalization given in [4] in a way that allows us to deduce preservation of strong normalization for object languages different from λ-calculus. Another interesting challenge is to extend CINNI by a notion of composition, although it appears that composition is not compatible with preservation of strong normalization even for indexed-based calculi.

Acknowledgements. This work has been supported by DARPA through Rome Laboratories Contract F30602-C-0312, by DARPA and NASA through Contract NAS2-98073, by Office of Naval Research Contract N00014-99-C-0198, and by National Science Foundation Grant CCR-9900334. We would like to thank José Meseguer for his advice, including his early suggestion to represent higher-order languages in rewriting logic using explicit substitutions, Cesar Muñoz for many discussions on calculi of explicit substitutions, especially in connection with type theory and proof synthesis, and Narciso Martí-Oliet for many useful suggestions and his constructive criticism. We also appreciate the comments of several referees which helped to improve the paper. In particular, we are indebted to an anonymous referee for pointing out that the representation used by CINNI for pure terms is equivalent to Berkling's representation, and for providing us with the corresponding references.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

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

[3] H. P. Barendregt. Lambda-calculi with types. In S. Abramsky, D. M. Gabbay, and T.S.E. Maibaum, editors, *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*. Oxford: Clarendon Press, 1992.

[4] Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, September 1996.

[5] S. Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and other systems in Barendregt's cube. Technical report, Carnegie Mellon University and Universita di Torino, 1988.

[6] K. J. Berkling. A symmetric complement to the lambda-calulus. Technical Report Interner Bericht ISF-76-7, GMD, St. Augustin, Germany, 1976.

[7] K. J. Berkling and E. Fehr. A consistent extension of the lambda-calculus as a base for functional programming languages. *Information and Control*, 55:89–101, 1982.

[8] R. Bloo. *Preservation of Termination for Explicit Substitution*. PhD thesis, Eindhoven University of Technology, 1997. IPA Dissertation Series 1997-05.

[9] R. Bloo and K. H. Rose. Combinatory reduction systems with explicit substitution that preserve strong normalisation. In H. Ganzinger, editor, *RTA '96–Rewriting Techniques and Applications, New Brunswick, New Jersey, July 1996, Rutgers University*, volume 1103 of *Lecture Notes in Computer Science*, pages 169–183. Springer-Verlag, 1996.

[10] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. A tutorial on Maude. SRI International, March 2000, `http://maude.csl.sri.com`.

[12] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. Computer Science Laboratory, SRI International, Menlo Park, 1999. `http://maude.csl.sri.com`.

[13] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[14] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[15] 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.

[16] N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Proc. Kninkl. Nederl. Akademie van Wetenschappen*, volume 75(5), pages 381–392, 1972.

[17] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science, San Diego, California, June 1995*, pages 366–374. IEEE Computer Society Press, 1995.

[18] G. Dowek, T. Hardin, and C. Kirchner. HOL-lambda-sigma: An intentional first-order expression of higher-order logic. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99), Trento, Italy, July 1999*, volume 1631 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1999.

[19] J. Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proceedings of the 17th Anual ACM Symposium on Principles of Programming Languages, Orlando (Florida, USA)*, pages 1–15. ACM, San Francisco, 1990.

[20] T. Hardin. Confluence results for the pure strong categorical combinatory logic. *Theoretical Computer Science*, 1988.

[21] D. Hirschkoff. Handling substitutions explicitely in the pi-calculus. In *Proceedings of WESTAPP'99, Second International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs, July 5, 1999, Trento, Italy.*

[22] F. Kamareddine and A. Ríos. Relating the $\lambda\sigma$- and λs-styles of explicit substitutions. *Journal of Logic and Computation*, 2000.

[23] D. Kesner and P.E.M. López. Explicit substitutions for objects and functions. *The Journal of Functional and Logic Programming*, 1999. Special Issue 2.

[24] P. Lescanne. Termination of rewrite systems by elementary interpretations. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming, Third International Conference, Volterra, Italy, September 1992, Proceedings*, volume 632 of *Lecture Notes in Computer Science*, pages 21 – 36. Springer-Verlag, 1992.

[25] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In Hans Boehm, editor, *Proc. 21st Annual ACM Synposium on Principles of Programming Languages, Portland, USA*, pages 60–69. ACM, 1994.

[26] P. Lescanne and J. Rouyer-Degli. The calculus of explicit substitutions $\lambda\upsilon$. Technical Report RR-2222, INRIA-Lorraine, January 1994.

[27] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[28] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT '97, Tarquinia, Italy, June 3-7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18 – 61. Springer-Verlag, 1998.

[29] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus.* Cambridge University Press, May 1999.

[30] C. Muñoz. A calculus of substitutions for incomplete-proof representation in type theory. Technical Report RR-3309, Unité de recherche INRIA-Rocquencourt, Novembre 1997.

[31] C. Muñoz. Proof synthesis via explicit substitutions on open terms. In *Proc. International Workshop on Explicit Substitutions, Theory and Applications, WESTAPP 98*, Tsukuba (Japan), April 1998.

[32] M.J. O'Donnell. Computing in systems descibed by equations. volume 58 of *Lecture Notes in Computer Science*. Springer-Verlag, 1977.

[33] B. Pagano. X.R.S.: Explicit reduction systems – a first-order calculus for higher-order calculi. In C. Kirchner and H. Kirchner, editors, *Automated Deduction – CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 1998, Proceedings*, volume 1421 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1998.

[34] C. Reinke. *Functions, Frames and Interactions — completing a λ-calculus-based purely functional language with respect to programming-in-the-large and interactions with runtime environments.* PhD thesis, Christian-Albrechts-Universität Kiel, Germany, 1998. Report 9804.

[35] K. H. Rose. Explicit substitution - tutorial & survey. Technical Report Lecture Series LS-96-3, BRICS, Dept. of Computer Science, University of Aarhus, September 1996. `http://www.brics.dk/LS/96/3/BRICS-LS-96-3/BRICS-LS-96-3.html`.

[36] M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic. In *Proc. of LFM'99: Workshop on Logical Frameworks and Meta-languages, Paris, France, September 28, 1999.*

[37] M.O. Stehr. CINNI - A Generic Calculus of Explicit Substitutions and its Application to lambda-, sigma- and pi-calculi. Extended version of this paper available at `http://www.csl.sri/~stehr`, October 2000.

[38] J. Terlouw. Een nadere bewijstheoretische Analyse van GSTTs. Manuscript, University of Nijmegen, The Netherlands, 1989.

[39] P. Viry. Input/output for ELAN. In J. Meseguer, editor, *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, Asilomar, California, September 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.com/locate/entcs/volume4.html`.