2001 0020251

NASA/CR-2000-210621
ICASE Report No. 2000-45

NASA-CR-2000-210621
20010020251

Explicit Substitutions and All That

Mauricio Ayala-Rincón
Universidade de Brasilıa, Brasilia D.F., Brasil

César Muñoz
ICASE, Hampton, Virginia

November 2000

The NASA STI Program Office . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities These results are published by NASA in the NASA STI Report Series, which includes the following report types

- TECHNICAL PUBLICATION Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value NASA's counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e g , quick release reports, working papers, and bibliographies that contain minimal annotation Does not contain extensive analysis

- CONTRACTOR REPORT Scientific and technical findings by NASA-sponsored contractors and grantees

- CONFERENCE PUBLICATIONS. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or cosponsored by NASA

- SPECIAL PUBLICATION Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest

- TECHNICAL TRANSLATION English-language translations of foreign scientific and technical material pertinent to NASA's mission

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized data bases, organizing and publishing research results even providing videos

For more information about the NASA STI Program Office, see the following

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

- Email your question via the Internet to help@sti.nasa gov

- Fax your question to the NASA STI Help Desk at (301) 621-0134

- Telephone the NASA STI Help Desk at (301) 621-0390

- Write to.
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076-1320

NASA/CR-2000-210621
ICASE Report No. 2000-45

Explicit Substitutions and All That

Mauricio Ayala-Rincón
Universidade de Brasilia, Brasilia D.F., Brasil

César Muñoz
ICASE, Hampton, Virginia

ICASE
NASA Langley Research Center
Hampton, Virginia

Operated by Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-97046

November 2000

Available from the following

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 487-4650

EXPLICIT SUBSTITUTIONS AND ALL THAT*

MAURICIO AYALA-RINCON† AND CESAR MUÑOZ‡

Abstract. Explicit substitution calculi are extensions of the λ-calculus where the substitution mechanism is internalized into the theory. This feature makes them suitable for implementation and theoretical study of logic-based tools such as strongly typed programming languages and proof assistant systems. In this paper we explore new developments on two of the most successful styles of explicit substitution calculi: the $\lambda\sigma$- and λs_e-calculi.

Key words. explicit substitution, higher order unification, lambda-calculus, type theory, rewriting

Subject classification. Computer Science

1. Introduction. This paper focuses on the uses of explicit substitutions in the language of the simply-typed λ-calculus. Type theories were used at the beginning of the twentieth century as a formalism to deal with the mathematical paradoxes studied at that time and incorporated in 1940 to the λ-calculus by A. Church [11]. The need of stronger programming languages guided type theory to the interest of computer scientists in the 1970's and 1980's, when new languages based on type theories were developed. Probably the most relevant of these languages is ML [42], developed by R. Milner. In the 1990's, several proof assistant systems based on higher-order logics, such as Coq [5], HOL [27], and PVS [51], were developed. The λ-calculus is the simplest logical framework for reasoning about formal properties of all these systems. Many of the essential techniques and computational procedures involved in these tools have been developed, analyzed, and improved in the context of the simply-typed λ-calculus before being implemented. These techniques include simple mechanisms such as type checking and type inference, and more complex ones such as those used for dealing with the inhabitation problem and the higher order unification problem. The basic operation of the λ-calculus is the β-conversion that was originally defined based on an implicit notion of substitution where renaming of variables was informally assumed to avoid "clashes" and "captures". This implicitness of the notion of substitution was not critical before this theoretical framework was used in other contexts than the ones of computer science, but making the notion of substitution explicit is essential when computational properties such as time and space complexity should be analyzed.

We will focus on two styles of explicit substitutions: $\lambda\sigma$ and λs_e. These calculi use a name-less notation for variables. Therefore, technical nuisances due to the higher order aspect of λ-calculus, such as renaming and capture of variables, are minimized or completely eliminated in $\lambda\sigma$ and λs_e. For these calculi, we will motivate and illustrate different techniques developed for important computational problems and applications such as higher order unification, type inference, and inhabitation problem. These kind of problems arise naturally in many fields of computer science. Some of the current progress in the area of explicit substitution is recorded in the series of "International Workshops on Explicit Substitutions: Theory and Applications to Programs and Proofs" - WESTAPP that runs yearly together with the Conference on Rewriting Techniques

*Work carried out during the visit of the first author at the ULTRA Group, CEE, Heriot-Watt University, Edinburgh, Scotland, and funded by CAPES (BEX0384/99-2) Brazilian Foundation. For the second author, the work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while he was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681-2199, USA.

†Departamento de Matematica, Universidade de Brasilia, 70910-900 Brasília D.F., Brasil, e-mail: ayala@mat.unb.br

‡ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-2199, USA, e-mail: munoz@icase.edu

and Applications - RTA. For other surveys and tutorials on explicit substitution calculi see [38, 56].

Firstly, in section 2 we present basic notions of the λ-calculus, its representation in de Bruijn index notation, its simply-typed version, and the Curry-Howard isomorphism. Afterwards, in section 3, we motivate explicit substitutions and present the two before mentioned calculi of explicit substitutions along with their simply-typed versions. In section 4, we explain briefly the applications of explicit substitutions before concluding in section 5.

2. The λ-calculus. The λ-calculus was developed by Church around 1930 [12] as a formal language for the foundations of mathematics and logic. Although that foundation was later revealed to be inconsistent, indeed Russell paradox [58] can be encoded in it, the λ-calculus still provides a formal model of computability. Church and Kleene [37, 10] proved that the class of λ-expressions and the class of partial-recursive functions are the same. This result, along with Turing's own work, shows that the λ-calculus is as expressive as Turing machines.

The notation consists of a set Λ of terms and rules to manipulate them. The set Λ is built on a countable set of variables $\mathcal{V} = \{x, y, \ldots\}$ and it is inductively defined as follows: $\mathcal{V} \subset \Lambda$, if $M, N \in \Lambda$ then $(M\ N) \in \Lambda$, and if $x \in \mathcal{V}$ and $M \in \Lambda$ then $\lambda x.M \in \Lambda$. Terms of the form $(M\ N)$ are called *applications* and terms of the form $\lambda x.M$ are called *abstractions*. Abstractions are binding structures. As usual for these kind of structures, a notion of *free* and *bound* variables is necessary. The set of *free variables* of M, denoted $\mathcal{FV}(M)$, is defined by $\mathcal{FV}(x) = \{x\}$, $\mathcal{FV}((M\ N)) = \mathcal{FV}(M) \cup \mathcal{FV}(N)$, and $\mathcal{FV}(\lambda x.M) = \mathcal{FV}(M) \setminus \{x\}$. The variable x in a term $\lambda x.M$ is said to be *bound*. Names of bound variables are irrelevant. For instance, $\lambda x.x$ and $\lambda y.y$ represent the same λ-term. This implicit equivalence is called α-conversion. Formally, if $z \notin \mathcal{FV}(M)$, then $\lambda x.M =_\alpha \lambda z.M\{z/x\}$, where for an arbitrary term N, $M\{N/x\}$ denotes the *atomic substitution* of the free occurrences of the variable x in M by N.

Substitution plays a very important role in the λ-calculus. In fact, the main computational rule in this formalism, the β-rule, is expressed as follows: $(\lambda x.M\ N) \xrightarrow{\beta} M\{N/x\}$. Informally, it states that the application of a function $\lambda x.M$ to an argument N, results in a term $M\{N/x\}$ where the formal parameter x has been replaced by the argument N in M (the body of the function). An additional rule, called η, states that abstractions computing the same value for the same argument are convertible. Formally, $\lambda x.(M\ x) \xrightarrow{\eta} M$, if $x \notin \mathcal{FV}(M)$.

The formal definition of substitution is not as simple as it seems. The following one, commonly used in implementations, is wrong: $x\{M/x\} = M$, $y\{M/x\} = y$, if $y \neq x$, $(M_1\ M_2)\{M/x\} = (M_1\{M/x\}\ M_2\{M/x\})$, $(\lambda x.N)\{M/x\} = \lambda x.N$, and $(\lambda y.N)\{M/x\} = \lambda y.N\{M/x\}$, if $y \neq x$. The problem arises in the last case: the term M may contain a free variable y which becomes a bound variable when the substitution is applied. A correct definition should avoid this capture, for instance, by modifying the last case with $(\lambda y.N)\{M/x\} = \lambda z.N\{z/y\}\{M/x\}$, where $z \notin \mathcal{FV}(M)$.

The λ-calculus is not terminating. Indeed, a term like $(\lambda x.(x\ x)\ \lambda x.(x\ x))$ β-reduces to itself and then it can be always reduced. However, the λ-calculus satisfies, the *Church-Rosser property* i.e., if $M_1 =_{\beta\eta} M_2$, then there exists N such that $M_1 \xrightarrow{\beta\eta^*} N$ and $M_2 \xrightarrow{\beta\eta^*} N$.[1] In consequence: (1) the λ-calculus is also *confluent* and (2) normal forms, it they exist, are unique. We refer to [3] for a complete description of the λ-calculus and its properties.

---

[1] As usual, if R is a term rewrite system, we denote by $\xrightarrow{R}$ the relation induced by R and by $\xrightarrow{R^*}$ the reflexive, symmetric, and transitive closure of $\xrightarrow{R}$. Furthermore, the equational theory associated to R defines a congruence denoted by $=_R$.

2.1. de Bruijn indices. At the beginning of the seventies, de Bruijn developed a nameless notation for the λ-calculus [19] In that notation, names of bound variables are replaced with *indices*

DEFINITION 2 1 *The set* Λ_{dB} *of* λ-terms in de Bruijn index notation *is defined inductively as*

$$M, N \quad = \underline{n} \mid (M \ \ N) \mid \lambda M$$

where $n \in \mathbb{N}^{>0}$

An index counts the number of λ-symbols in the binding scope of the bound variable that it represents For instance, in de Bruijn index notation, the term $\lambda x\, x$ is written $\lambda \underline{1}$ since the bound variable x is in the binding scope of one λ-symbol Similarly, the term $\lambda x\,(\lambda y\,(x \ \ y) \ \ x)$ is written $\lambda(\lambda(\underline{2} \ \ \underline{1}) \ \ \underline{1})$ Note that the same index appearing in different binding scopes represents different variables Vice-versa, occurrences of the same variable appearing in different binding scopes are denoted by different indices

Free variables can also be represented by de Bruijn indices In that case, it is necessary to fix an enumeration, namely a *referential*, $x_1, x_2, \quad, x_n$, of free variable names If the occurrence of a variable is denoted by an index $\underline{n}$ and the number of λ-symbols in the binding scope of that occurrence is less than n, say m, then that occurrence of $\underline{n}$ represents the free-variable x_{n-m} of the referential For instance, the term $(\lambda x\,(y \ \ x) \ \ z)$ can be encoded as $(\lambda(\underline{2} \ \ \underline{1}) \ \ \underline{2})$ under the referential y, z and as $(\lambda(3 \ \ \underline{1}) \ \ \underline{1})$ under the referential z, y

The formulation of the rules β and η for Λ_{dB}-terms requires the following functions for updating and substitution of indices

DEFINITION 2 2 *Let* $M \in \Lambda_{dB}$ *The* i-lift *of* M, *denoted* M^{+i} *is defined inductively as follows*

1 $(M_1 \ \ M_2)^{+i} = (M_1^{+i} \ \ M_2^{+i})$,
2 $(\lambda N)^{+i} = \lambda N^{+(i+1)}$,
3 $\underline{n}^{+i} = \begin{cases} \underline{n+1}, \text{ if } n > i \\ \underline{n}, \text{ if } n \leq i \end{cases}$

The lift *of a term* M *is its* 0*-lift and is denoted briefly as* M^+

DEFINITION 2 3 *The* application of the substitution *with* N *at the depth* $n-1$ *on a term* M, *denoted* $M\{N/\underline{n}\}$, *is defined inductively as follows*

1 $(M_1 \ \ M_2)\{N/\underline{n}\} = (M_1\{N/\underline{n}\} \ \ M_2\{N/\underline{n}\})$,
2 $(\lambda M)\{N/\underline{n}\} = \lambda M\{N^+/n+1\}$,
3 $\underline{m}\{N/\underline{n}\} = \begin{cases} \underline{m-1}, \text{ if } m > n \\ N, \text{ if } m = n \\ \underline{m}, \text{ if } m < n \end{cases}$

DEFINITION 2 4 *The rules* β *and* η *are defined for the set of* Λ_{dB}*-terms as follows*

$$\begin{array}{lcl} (\lambda M \ \ N) & \xrightarrow{\beta} & M\{N/\underline{1}\} \\ \lambda(M \ \ \underline{1}) & \xrightarrow{\eta} & N, \text{ if } N^+ = M \end{array}$$

EXAMPLE 2 5 The λ-term $(\lambda x\,(\lambda y\,(x \ \ z) \ \ x) \ \ (z \ \ \lambda z\,(x \ \ z)))$ can be translated under the referential x, y, z into the Λ_{dB}-term $(\lambda(\lambda(\underline{2} \ \ \underline{5}) \ \ \underline{1}) \ \ (\underline{3} \ \ \lambda(\underline{2} \ \ \underline{1})))$ Furthermore, we have

$$(\lambda x\,(\lambda y\,(x \ \ z) \ \ x) \ \ (z \ \ \lambda z\,(x \ \ z))) \xrightarrow{\beta} (\lambda y\,((z \ \ \lambda z\,(x \ \ z)) \ \ z) \ \ (z \ \ \lambda z\,(x \ \ z)))$$

We examine in detail the steps of that reduction for Λ_{dB}-terms

$$\begin{aligned}
(\lambda(\lambda(\underline{2}\ \underline{5})\ \underline{1})\ (\underline{3}\ \lambda(\underline{2}\ \underline{1}))) &\xrightarrow{\beta} (\lambda(\underline{2}\ \underline{5})\ \underline{1})\{(\underline{3}\ \lambda(\underline{2}\ \underline{1}))/\underline{1}\}\\
&= ((\lambda(\underline{2}\ \underline{5}))\{(\underline{3}\ \lambda(\underline{2}\ \underline{1}))/\underline{1}\}\ \underline{1}\{(\underline{3}\ \lambda(\underline{2}\ \underline{1}))/\underline{1}\})\\
&= (\lambda(\underline{2}\ \underline{5})\{(\underline{3}\ \lambda(\underline{2}\ \underline{1}))^{+}/\underline{2}\}\ (\underline{3}\ \lambda(\underline{2}\ \underline{1})))\\
&= (\lambda(\underline{2}\ \underline{5})\{(\underline{3}^{+}\ \lambda(\underline{2}^{+1}\ \underline{1}^{+1}))/\underline{2}\}\ (\underline{3}\ \lambda(\underline{2}\ \underline{1})))\\
&= (\lambda(\underline{2}\ \underline{5})\{(\underline{4}\ \lambda(\underline{3}\ \underline{1}))/\underline{2}\}\ (\underline{3}\ \lambda(\underline{2}\ \underline{1})))\\
&= (\lambda(\underline{2}\{(\underline{4}\ \lambda(\underline{3}\ \underline{1}))/\underline{2}\}\ \underline{5}\{(\underline{4}\ \lambda(\underline{3}\ \underline{1}))/\underline{2}\})\ (\underline{3}\ \lambda(\underline{2}\ \underline{1})))\\
&= (\lambda((\underline{4}\ \lambda(\underline{3}\ \underline{1}))\ \underline{4})\ (\underline{3}\ \lambda(\underline{2}\ \underline{1})))
\end{aligned}$$

The Λ_{dB}-term $(\lambda((\underline{4}\ \lambda(\underline{3}\ \underline{1}))\ \underline{4})\ (\underline{3}\ \lambda(\underline{2}\ \underline{1})))$ represents the term $(\lambda y\ ((z\ \lambda z\ (x\ z))\ z)\ (z\ \lambda z\ (x\ z)))$ under the given referential •

EXAMPLE 2 6 Notice that

$$\lambda((\lambda\lambda(\underline{5}\ (\underline{1}\ \underline{2}))\ \underline{4})\ \underline{1}) \xrightarrow{\eta} (\lambda\lambda(\underline{4}\ (\underline{1}\ \underline{2}))\ \underline{3})$$

since

$$\begin{aligned}
(\lambda\lambda(\underline{4}\ (\underline{1}\ \underline{2}))\ \underline{3})^{+} &= ((\lambda\lambda(\underline{4}\ (\underline{1}\ \underline{2})))^{+}\ \underline{3}^{+})\\
&= (\lambda(\lambda(\underline{4}\ (\underline{1}\ \underline{2})))^{+1}\ \underline{3}^{+})\\
&= (\lambda\lambda(\underline{4}\ (\underline{1}\ \underline{2}))^{+2}\ \underline{3}^{+})\\
&= (\lambda\lambda(\underline{4}^{+2}\ (\underline{1}\ \underline{2})^{+2})\ \underline{3}^{+})\\
&= (\lambda\lambda(\underline{4}^{+2}\ (\underline{1}^{+2}\ \underline{2}^{+2}))\ \underline{3}^{+})\\
&= (\lambda\lambda(\underline{5}\ (\underline{1}\ \underline{2}))\ \underline{4})
\end{aligned}$$

•

2.2. Simply-typed λ-calculus. The λ-calculus is a simple, but yet powerful formalism As we said before, when used as a logical framework, the λ-calculus allows the encoding of paradoxes To solve that problem, Church developed a *typed* version of the λ-calculus [11] which happens to be a simplification of the Type Theory of Whitehead-Russell [58]

The effect of typed λ-calculus can be seen on a term such as $\lambda x\ (x\ x)$ which is a well formed term in the untyped λ-calculus that represents the abstract concept of "self-application" The meaningfulness of this concept may be questioned and was involved in many of the logical paradoxes from the beginning of the twentieth century Thinking about x as a functional variable from A to A or of "*type* $A \rightarrow A$", the application $(x\ x)$ is forbidden, since it's impossible to apply a function of type $A \rightarrow A$ to an argument of type $A \rightarrow A$ This coincides with the conception of functional objects assumed by most mathematicians Of course, if z is a variable of type A, the typed expression $\lambda x\ (x\ (x\ z))$ makes sense For a formal introduction to the theory of the simply-typed λ-calculus and interesting historical remarks see [30]

In a typed λ-calculus, λ-terms are stratified in several categories, namely *types* A *type*, in the *simple type theory*, can be a basic type $a, b,$ or a functional type $A \rightarrow B$, where A and B are types We use upper-case letters A, B to range over types Only terms that follow a type discipline are considered to be valid The type discipline is enforced by a set of *typing rules* Thanks to the typing rules, Russell's paradox cannot be expressed in the simple type theory

$$\frac{x \notin \Gamma}{x\ A, \Gamma \vdash x\quad A}\ \text{(Start)} \qquad\qquad \frac{x \notin \Gamma \qquad \Gamma \vdash M\quad B}{x\ A, \Gamma \vdash M\quad B}\ \text{(Weak)}$$

$$\frac{x\ A, \Gamma \vdash M\quad B}{\Gamma \vdash \lambda x\ A\ M\quad A \to B}\ \text{(Abs)} \qquad\qquad \frac{\Gamma \vdash M\quad A \to B \qquad \Gamma \vdash N\quad A}{\Gamma \vdash (M\quad N)\quad B}\ \text{(Appl)}$$

FIG 2 1 *The simply-typed λ-calculus*

$$\frac{1 \le i \le n}{A_1\ A_2\quad A_n \vdash \underline{i}\quad A_i}\ \text{(Var)} \qquad \frac{A\ \Gamma \vdash M\quad B}{\Gamma \vdash \lambda_A\ M\quad A \to B}\ \text{(Abs)} \qquad \frac{\Gamma \vdash M\quad A \to B \qquad \Gamma \vdash N\quad A}{\Gamma \vdash (M\quad N)\quad B}\ \text{(Appl)}$$

FIG 2 2 *The simply-typed λ-calculus for Λ_{dB}-terms*

Typed λ-terms are elements of the set of Λ-terms except that bound variables in abstractions have type annotations, i e , they have the form $\lambda x\ A\ M$ Rules β and η are modified accordingly

$$(\lambda x\quad A\ M\quad N) \xrightarrow{\beta} M\{N/x\} \qquad \text{and} \qquad \lambda x\quad A\ (M\quad x) \xrightarrow{\eta} M, \text{ if } x \notin \mathcal{FV}(M)$$

A typing judgment $\Gamma \vdash M\quad A$ denotes that the term M has type A in Γ, where Γ is a *context*, i e , a list $x_1\ A_1, \quad , x_n\ A_n$ of variable declarations Henceforth, we use Greek letters $\Gamma, \Delta,$ to range over contexts Figure 2 1 shows the typing rules of the simply-typed λ-calculus We say that a λ-term M is *well typed* in Γ if and only if there exists a type A such that $\Gamma \vdash M\quad A$, and we say that a type A is *inhabited* in Γ if and only if there exists a λ-term M such that $\Gamma \vdash M\quad A$

The presentation of the typed λ-calculus used in this paper corresponds to the *Church-style* In this presentation, typed λ-terms are elements of the set of Λ-terms except for abstractions, which have type annotations An alternative presentation, called *Curry-style*, considers typed λ-terms as standard Λ-terms without type annotations In that case, *type variables* should be added to the formalism Indeed, in a typed λ-calculus *à la* Curry, the type of $\lambda x\ x$ is $\alpha \to \alpha$ where α denotes any type (See [4])

Type checking is decidable for the simply typed λ-calculus That is, there is a method to decide whether or not a term has a type in a given context according to the typing rules As the untyped version of the λ-calculus, the simply-typed λ-calculus enjoys the Church-Rosser property and therefore it is also confluent Furthermore, it also satisfies the following properties

- *Subject reduction*, if $\Gamma \vdash M\quad A$ and $M \xrightarrow{\beta\eta} N$, then $\Gamma \vdash N\quad A$,
- *Type uniqueness*, if $\Gamma \vdash M\quad A$ and $\Gamma \vdash M\quad B$, then $A = B$,
- *Strong normalization*, if M is a well typed term, then M has no reductions of infinite length Therefore, due to the confluence property, normal-forms of well typed terms always exists and they are unique

In the de Bruijn setting of the simply typed λ-calculus, a context Γ is a list of types $A_1 \quad A_n$ where A_i is the type of the free-variable represented by the index $\underline{i}$ The empty context is denoted by ϵ Simply-typed Λ_{dB}-terms are defined by the typing rules of Fig 2 2

2.3. Curry-Howard isomorphism. There is a strong relation between type theory and intuitionistic logic If we identify types with propositions, where an arrow type is an implication, typing rules of the simply-typed λ-calculus correspond one to one to deduction rules of a minimal intuitionistic logic In other words, typing rules are logical rules decorated with typed λ-terms This principle is known as the *Curry-Howard* isomorphism

Consider an intuitionistic minimal logic where propositional formulas are built from atomic propositions $a, b,$ and the implication, i e , if A and B are formulas then $A \to B$ is a formula We use uppercase Greek letters Ω to range over *set* of formulas We write Ω, A as a shorthand for $\Omega \cup \{A\}$ A judgment $\Omega \vdash_I A$ denotes that A is a logical consequence of Ω A judgment is said *provable (in the minimal intuitionistic logic)* if and only if it is derived by top-down application of the following rules

$$\frac{}{\Omega, A \vdash_I A}\,(\text{Axiom}) \qquad \frac{\Omega, A \vdash_I B}{\Omega \vdash_I A \to B}\,(\text{Intro}) \qquad \frac{\Omega \vdash_I A \to B \qquad \Omega \vdash_I A}{\Omega \vdash_I B}\,(\text{Elim})$$

A formula A is a *tautology* if and only if the judgment $\vdash_I A$ is provable For example, the formula $A \to ((A \to B) \to B)$ is a tautology since it can be derived as follows

$$\cfrac{\cfrac{\cfrac{\cfrac{}{A, A \to B \vdash_I A \to B}\,(\text{Axiom}) \qquad \cfrac{}{A, A \to B \vdash_I A}\,(\text{Axiom})}{A, A \to B \vdash_I B}\,(\text{Elim})}{A \vdash_I (A \to B) \to B}\,(\text{Intro})}{\vdash_I A \to ((A \to B) \to A)}\,(\text{Intro})$$

Formally, the Curry-Howard isomorphism says that $\Omega \vdash_I A$ is provable in the minimal intuitionistic logic if and only if $\Gamma \vdash M \quad A$ is a valid typing judgment in the simply-typed λ-calculus, where Γ is a list of variable declaration of propositions, seen as types, in Ω The term M is a λ-term that represents the proof derivation For instance, the term decoration of the tree derivation above results in the valid typing judgment $\vdash \lambda x \; A \; \lambda y \; A \to B \; (y \quad x) \quad A \to ((A \to B) \to A)$

The Curry-Howard isomorphism is extended to intuitionistic first order and higher order logics and it is widely studied in proof theory It is at the base of mathematic formalizations where proofs are just mathematical objects Such languages are the base of automatic systems for proof construction, program verification and program synthesis

3. Explicit Substitutions. Implicitness of substitution is the *Achilles heel* of the λ-calculus Namely, the λ-calculus is a convenient and compact model of the computable functions but it does not provide any mechanism for observing essential operational properties of these functions as time and space complexity The reason for this is that the substitution involved in β-reductions does not belong in the calculus, but rather in an informal meta-level In practice, β-reduction is not a primitive operation and is implemented based on a substitution generally elaborated by renaming variables and/or maintaining some variable convention That makes it impossible to determine or bound in time and space the β-reduction

The $\lambda\sigma$-calculus was the first one presented formally as a mechanism for making explicit substitution in the λ-calculus [1] But before this, today widely considered seminal work, many empiric and theoretic efforts were realized in order to solve the problem of implicitness of the substitution operation From the theoretical point of view, the *Combinatory Logic* of Curry and Feys [18] proposed the first solution to this problem However, this setting does not remain close to the λ-calculus and the number of primitive steps can be extensively larger than required by explicit substitution calculi From the empirical point of view, perhaps the person who provided the foundations to take care of this problem was de Bruijn himself, when developing his system AUTOMATH from the middle of the 1960's Part of his primary conceptions was the previously mentioned nice nameless notation for the λ-calculus [19] His legacy is collected in [50]

Since the $\lambda\sigma$-calculus was introduced in [1], several other variants of explicit substitution calculi have been proposed (see, for example, [54, 38, 32, 7, 39, 17, 35, 43, 24, 44]). These calculi implement several styles of explicit substitutions.

We will focus our attention on two of these styles: the $\lambda\sigma$- and the λs_e-styles. Both of them use a nameless notation based on the de Bruijn index notation, which is completely insensitive to α-conversion. That allows a clean and elegant meta-theoretical study of the calculi which make them suitable for implementation of declarative programming languages, higher order proof assistants, and automated deductive systems. Both styles were shown incomparable in [34].

The $\lambda\sigma$-calculus and its variants have been proposed as a general framework for higher order unification and term synthesis [21, 22, 9, 36, 45, 47, 46, 6]. Furthermore, calculi of the $\lambda\sigma$-family have been incorporated with success into programming languages and proof assistants. For example, an algorithm for pattern unification for dependent types, based on $\lambda\sigma$, has been implemented in the Twelf system [52]. It has also been relevant in the improvement of the explicit substitution for the *rewrite calculus* (ρ-calculus [14]) of the ELAN system, which provides a language based on rewrite rules for specifying and prototyping deductive systems [13].

The λs_e-calculus [32, 33] was developed more recently than the $\lambda\sigma$-calculus and its main claimed advantage over the $\lambda\sigma$-calculus is that it remains as close as possible to the λ-calculus having only one sort of objects. There is a close relation, until now only subjectively purposed, between the λs_e-calculus and the rewrite rules developed by Nadathur and Wilson in the early 1990's and used in the implementation of the higher order logic programming language λProlog [41]. For instance the laziness in the substitution needed in implementations of β-reduction, that arises naturally in the λs_e-calculus, is provided as the informal but empirical concept of *suspension* of substitutions by Nadathur and Wilson rewrite rules, with their notion of substitution being more general than the λs_e one. More recently their rewrite rules were published in the context of explicit substitution as the *suspension calculus* [49, 48]. Establishing formally the relations and differences between the λs_e-calculus and the suspension calculus remains as important work to be done.

3.1. The $\boldsymbol{\lambda\sigma}$-calculus. The $\lambda\sigma$-calculus is a first order rewrite system with two sorts of expressions: *terms* and *substitutions*. In fact, substitutions inherent to the β-rule in de Bruijn index notation, $(\lambda M\ N) \xrightarrow{\beta} M\{N/\underline{1}\}$, are delayed and recorded in the $\lambda\sigma$-calculus as $(\lambda M\ N) \longrightarrow M[N \cdot id]$. Here, $M[N \cdot id]$ is a $\lambda\sigma$-expression representing M with a recorded substitution $N \cdot id$. Additional rules are necessary for applying the recorded substitution to the term M, i.e., replacing all the free occurrences of the de Bruijn index $\underline{1}$ at M with N and decrementing by one remaining free de Bruijn indices over M. Delaying application of substitution is widely used in implementations of functional and logical programming languages, because immediate substitution may give rise to a size explosion of the expressions.

DEFINITION 3.1 (**$\boldsymbol{\lambda\sigma}$**-calculus). *The $\lambda\sigma$-calculus is defined by the rewrite system depicted in Fig. 3.1 where*

$$\begin{array}{llcl} \text{TERMS} & M,N & = & \underline{1} \mid \lambda M \mid (M\ N) \mid M[S] \\ \text{SUBSTITUTIONS} & S,T & = & id \mid \uparrow \mid M \cdot S \mid S \circ T \end{array}$$

The rewrite system obtained by dropping rules (Beta) and (Eta) of $\lambda\sigma$ is called σ.

In $\lambda\sigma$, de Bruijn indices are encoded by means of the constant $\underline{1}$ and the substitution $\uparrow$. We write $\uparrow^n$ as a shorthand for $\overbrace{\uparrow \circ \cdots \circ \uparrow}^{n\text{-times}}$. We overload the notation $\underline{i}$ to represent the $\lambda\sigma$-term corresponding to the index

$$
\begin{array}{llll}
(\lambda M\ N) & \longrightarrow & M[N \cdot id] & \text{(Beta)} \\
(M\ N)[S] & \longrightarrow & (M[S]\ N[S]) & \text{(App)} \\
(\lambda M)[S] & \longrightarrow & \lambda M[\underline{1} \cdot (S \circ \uparrow)] & \text{(Abs)} \\
M[S][T] & \longrightarrow & M[S \circ T] & \text{(Clos)} \\
\underline{1}[M \cdot S] & \longrightarrow & M & \text{(VarCons)} \\
M[id] & \longrightarrow & M & \text{(Id)} \\
(S_1 \circ S_2) \circ T & \longrightarrow & S_1 \circ (S_2 \circ T) & \text{(Assoc)} \\
(M \cdot S) \circ T & \longrightarrow & M[T] \cdot (S \circ T) & \text{(Map)} \\
id \circ S & \longrightarrow & S & \text{(IdL)} \\
S \circ id & \longrightarrow & S & \text{(IdR)} \\
\uparrow \circ (M \cdot S) & \longrightarrow & S & \text{(ShiftCons)} \\
\underline{1} \cdot \uparrow & \longrightarrow & id & \text{(VarShift)} \\
\underline{1}[S] \cdot (\uparrow \circ S) & \longrightarrow & S & \text{(SCons)} \\
\lambda(M\ \underline{1}) & \longrightarrow & N \quad \text{if } M =_\sigma N[\uparrow] & \text{(Eta)}
\end{array}
$$

FIG. 3.1. *The $\lambda\sigma$-calculus [1]*

i, i.e.,

$$
\underline{i} = \begin{cases} \underline{1} & \text{if } i = 1 \\ \underline{1}[\uparrow^n] & \text{if } i = n + 1 \end{cases}
$$

This one-shift encoding is interesting because involving a built-in deduction mechanism for arithmetic in implementations of systems based on the $\lambda\sigma$-calculus makes it difficult the analysis of time and space quantitative performance. But in any conceivable implementation one should use full indices at the meta-level instead of the one-shift encoding.

An explicit substitution denotes a mapping from indices to terms. Thus, id maps each index i to the term $\underline{i}$, $\uparrow$ maps each index i to the term $\underline{i+1}$, $S \circ T$ is the composition of the mapping denoted by T with the mapping denoted by S (notice that the composition of substitution follows a reverse order with respect to the usual notation of function composition), and finally, $M \cdot S$ maps the index 1 to the term M, and recursively, the index $i + 1$ to the term mapped by the substitution S on the index i.

The $\lambda\sigma$-calculus is not a confluent rewrite system [17], however it is confluent on ground expressions [1] and confluent on substitution-closed expressions (i.e., expressions without substitution variables) [54]. On the other hand, the σ-calculus, i.e., $\lambda\sigma$ without (Beta), is confluent and terminating [1].

A term is called *pure* if it does not contain substitutions. Notice that the set of pure terms in $\lambda\sigma$ and the set of Λ_{dB}-terms are identifiable. Furthermore, the $\lambda\sigma$-calculus simulates the λ-calculus [17], i.e., the relations induced by $\xrightarrow{\beta}$ and $\xrightarrow{(Beta)}\ \xrightarrow{\sigma^*}$ (one step of (Beta) followed by a σ-normalization) coincide *on pure terms*. However, the $\lambda\sigma$-calculus does not preserve strong-normalization of the λ-calculus [40], i.e., strongly normalizing λ-terms can be reduced forever in $\lambda\sigma$.

3.2. The $\lambda_{\mathcal{L}}$-calculus. As pointed out before, the one-shift encoding of indices in $\lambda\sigma$ is a theoretically convenient feature, but impractical for implementations. Nadathur also remarked in [48] that the non-left-linear rule of $\lambda\sigma$, namely (SCons), is difficult to handle in real implementations. Instead of rule (SCons), he suggested the meta-rule $\underline{1}[\uparrow^n] \cdot \uparrow^{n+1} \longrightarrow \uparrow^n$. Since $\uparrow^n$ is a shorthand in $\lambda\sigma$, an infinite set of rules is

$$
\begin{array}{llll}
(\lambda M\ N) & \longrightarrow & M[N \cdot \uparrow^0] & \text{(Beta)} \\
(\lambda M)[S] & \longrightarrow & \lambda M[\underline{1} \cdot (S \circ \uparrow^1)] & \text{(Abs)} \\
(M\ N)[S] & \longrightarrow & (M[S]\ N[S]) & \text{(App)} \\
M[S][T] & \longrightarrow & M[S \circ T] & \text{(Clos)} \\
\underline{1}[M \cdot S] & \longrightarrow & M & \text{(VarCons)} \\
M[\uparrow^0] & \longrightarrow & M & \text{(Id)} \\
(M \cdot S) \circ T & \longrightarrow & M[T] \cdot (S \circ T) & \text{(Map)} \\
\uparrow^0 \circ S & \longrightarrow & S & \text{(IdS)} \\
\uparrow^{n+1} \circ (M \cdot S) & \longrightarrow & \uparrow^n \circ S & \text{(ShiftCons)} \\
\uparrow^{n+1} \circ \uparrow^m & \longrightarrow & \uparrow^n \circ \uparrow^{m+1} & \text{(ShiftShift)} \\
\underline{1} \cdot \uparrow^1 & \longrightarrow & \uparrow^0 & \text{(Shift0)} \\
\underline{1}[\uparrow^{n+1}] \cdot \uparrow^{n+2} & \longrightarrow & \uparrow^{n+1} & \text{(ShiftS)} \\
\lambda(M\ \underline{1}) & \longrightarrow & N \quad \text{if } M =_{\mathcal{L}} N[\uparrow^1] & \text{(Eta)}
\end{array}
$$

FIG 3 2 *The rewrite system* $\lambda_{\mathcal{L}}$

represented by this scheme

Non-left-linear rules are not only annoying to implement, but they are usually responsible for non-confluence and typing problems Indeed, $\lambda\sigma$ is not confluent [17] and it does not preserve typing in a dependent-type system [45], both problems because of the non-left-linearity of the calculus

The $\lambda_{\mathcal{L}}$-calculus [44] is a left-linear variant of $\lambda\sigma$ where $\uparrow^n$ is a first-class substitution This allows the formulation of the rule suggested by Nadathur as a regular first order rule In fact, instead of (SCons), the the $\lambda_{\mathcal{L}}$-calculus has the following rule $\underline{1}[\uparrow^{n+1}] \cdot \uparrow^{n+2} \longrightarrow \uparrow^{n+1}$

DEFINITION 3 2 (**$\lambda_{\mathcal{L}}$**-calculus) *The $\lambda_{\mathcal{L}}$-calculus is defined by the rewrite system depicted in Fig 3 2 where*

$$
\begin{array}{llll}
\text{NATURAL NUMBERS} & n & = & 0 \mid n+1 \\
\text{TERMS} & M, N & = & \underline{1} \mid \lambda M \mid (M\ N) \mid M[S] \\
\text{SUBSTITUTIONS} & S, T & = & \uparrow^n \mid M \cdot S \mid S \circ T
\end{array}
$$

The $\mathcal{L}$-rewrite system is obtained by dropping rule (Beta) from $\lambda_{\mathcal{L}}$

We adopt the notation $\underline{i}$ as a shorthand for $\underline{1}[\uparrow^n]$ when $i = n+1$ Substitutions id and $\uparrow$ are written in $\lambda_{\mathcal{L}}$ as $\uparrow^0$ and $\uparrow^1$, respectively In general, $\uparrow^n$ denotes the mapping of each index i to the term $\underline{i+n}$ Using $\uparrow^n$, the scheme of rule proposed by Nadathur can be encoded in a first order rewrite system Natural numbers are constructed with 0 and $n+1$ Arithmetic calculations on indices are embedded in the rewrite system

The $\lambda_{\mathcal{L}}$-calculus is confluent on substitution-closed expressions and it simulates the λ-calculus [45] Just as $\lambda\sigma$, it does not preserve strong normalization

Another left-linear variant of $\lambda\sigma$ is the $\lambda\sigma_{\Uparrow}$-calculus [17] The $\lambda\sigma_{\Uparrow}$-calculus is a confluent first order rewrite system, i e , it is confluent on presence of both term and substitution variables However, $\lambda\sigma_{\Uparrow}$ raises some technical problem with η-conversions due to the fact that substitutions id and $\underline{1} \cdot \uparrow$ are not $\lambda\sigma_{\Uparrow}$-convertible

3.3. The λs_e-calculus. The λs_e-calculus avoids introducing two different sets of entities as the $\lambda\sigma$-calculus does, insisting in this way on remaining close to the syntax of the λ-calculus Next to abstraction and application, the λs_e-calculus introduces substitution (σ) and updating (φ) operators

$$
\begin{array}{llll}
(\lambda M\ \ N) & \longrightarrow & M\,\sigma^1 N & (\sigma\text{-generation}) \\
(\lambda M)\,\sigma^i N & \longrightarrow & \lambda(M\,\sigma^{i+1} N) & (\sigma\text{-}\lambda\text{-transition}) \\
(M_1\ \ M_2)\,\sigma^i N & \longrightarrow & ((M_1\,\sigma^i N)\ \ (M_2\,\sigma^i N)) & (\sigma\text{-app-transition}) \\
\underline{n}\,\sigma^i N & \longrightarrow & \begin{cases} \underline{n-1} & \text{if}\ \ n > i \\ \varphi^i_0\,N & \text{if}\ \ n = i \\ \underline{n} & \text{if}\ \ n < i \end{cases} & (\sigma\text{-destruction}) \\
\varphi^i_k(\lambda M) & \longrightarrow & \lambda(\varphi^i_{k+1}\,M) & (\varphi\text{-}\lambda\text{-transition}) \\
\varphi^i_k(M_1\ \ M_2) & \longrightarrow & ((\varphi^i_k\,M_1)\ \ (\varphi^i_k\,M_2)) & (\varphi\text{-app-transition}) \\
\varphi^i_k\,\underline{n} & \longrightarrow & \begin{cases} \underline{n+i-1} & \text{if}\ \ n > k \\ \underline{n} & \text{if}\ \ n \le k \end{cases} & (\varphi\text{-destruction}) \\
(M_1\,\sigma^i M_2)\,\sigma^j N & \longrightarrow & (M_1\,\sigma^{j+1} N)\,\sigma^i\,(M_2\,\sigma^{j-i+1} N) \quad \text{if}\ \ i \le j & (\sigma\text{-}\sigma\text{-transition}) \\
(\varphi^i_k\,M)\,\sigma^j N & \longrightarrow & \varphi^{i-1}_k M \quad \text{if}\ \ k < j < k+i & (\sigma\text{-}\varphi\text{-transition 1}) \\
(\varphi^i_k\,M)\,\sigma^j N & \longrightarrow & \varphi^i_k(M\,\sigma^{j-i+1} N) \quad \text{if}\ \ k+i \le j & (\sigma\text{-}\varphi\text{-transition 2}) \\
\varphi^i_k(M\,\sigma^j N) & \longrightarrow & (\varphi^i_{k+1}\,M)\,\sigma^j\,(\varphi^i_{k+1-j}\,N) \quad \text{if}\ \ j \le k+1 & (\varphi\text{-}\sigma\text{-transition}) \\
\varphi^i_k\,(\varphi^j_l\,M) & \longrightarrow & \varphi^j_l\,(\varphi^i_{k+1-j}\,M) \quad \text{if}\ \ l+j \le k & (\varphi\text{-}\varphi\text{-transition 1}) \\
\varphi^i_k\,(\varphi^j_l\,M) & \longrightarrow & \varphi^{j+i-1}_l M \quad \text{if}\ \ l \le k < l+j & (\varphi\text{-}\varphi\text{-transition 2}) \\
\lambda(M\ \ \underline{1}) & \longrightarrow & N \quad \text{if}\ \ M =_{s_e} \varphi^2_0 N & (\text{Eta})
\end{array}
$$

FIG 3 3 *Rewriting system of the λs_e-calculus*

DEFINITION 3 3 (**λs_e**-calculus) *The λs_e-calculus is given by the rewrite system in Fig 3 3 and the grammar*

$$M, N \ = \underline{n} \mid (M\ \ N) \mid \lambda M \mid M\sigma^j N \mid \varphi^i_k M \quad \textit{for}\ \ n, j, i \ge 1\ \ \textit{and}\ \ k \ge 0$$

The calculus of substitutions associated with the λs_e-calculus, namely s_e, is the rewriting system generated by the set of rules $s_e = \lambda s_e - \{\sigma\text{-generation}, Eta\}$

Intuitively, the substitution operator, σ, initiates (rule (σ-generation)) one-step of β-reduction, from $(\lambda M\ \ N)$, propagating the associated substitution innermost (rules (σ-λ) and (σ-app-transition)) Once this propagation is finished, when necessary, the updating operator, φ, is introduced to make the appropriate lift over N (rule (σ-destruction)) Otherwise either free de Bruijn indices are decremented by one or bounded maintained

The λs_e-calculus simulates β-reduction and is confluent [33] It does not preserve strong normalization [28]

3.4. Simply-typed calculi of explicit substitutions. In this section, we only include the essential notation of the simply-typed $\lambda_{\mathcal{L}}$- and λs_e-calculi Properties can be found in detail in [44] and [32], respectively Typing rules in both calculi follow the scheme as those of the simply-typed $\lambda\sigma$-calculus [21]

The rewrite rules of the typed $\lambda_{\mathcal{L}}$- and λs_e-calculi are defined by adding to their respective set of rules the necessary typing information Thus, for the simply-typed $\lambda_{\mathcal{L}}$-calculus we have the typed rules

$$
\begin{array}{llll}
(\lambda_A\,M\ \ N) & \longrightarrow & M[N\ \ \uparrow^0] & (\text{Beta}) \\
(\lambda_A\,M)[S] & \longrightarrow & \lambda_A\,M[\underline{1}\ \ (S \circ \uparrow^1)] & (\text{Abs}) \\
\lambda_A\,(M\ \ \underline{1}) & \longrightarrow & N \quad \text{if}\ \ M =_{\mathcal{L}} N[\uparrow^1] & (\text{Eta})
\end{array}
$$

$$\frac{}{A\,\Gamma \vdash \underline{1}\;\; A}\,(\text{Var}) \qquad \frac{A\,\Gamma \vdash N\;\; B}{\Gamma \vdash \lambda_A\, N\;\; A \rightarrow B}\,(\text{Lambda})$$

$$\frac{\Gamma \vdash M\;\; A \rightarrow B \qquad \Gamma \vdash N\;\; A}{\Gamma \vdash (M\;\; N)\;\; B}\,(\text{App}) \qquad \frac{\Gamma \vdash S \triangleright \Delta \qquad \Delta \vdash M\;\; A}{\Gamma \vdash M[S]\;\; A}\,(\text{Clos})$$

$$\frac{}{\Gamma \vdash \uparrow^0 \triangleright \Gamma}\,(\text{Id}) \qquad \frac{\Gamma \vdash \uparrow^n \triangleright \Delta}{A\,\Gamma \vdash \uparrow^{n+1} \triangleright \Delta}\,(\text{Shift})$$

$$\frac{\Gamma \vdash M\;\; A \qquad \Gamma \vdash S \triangleright \Delta}{\Gamma \vdash M\;\; S \triangleright A\,\Delta}\,(\text{Cons}) \qquad \frac{\Gamma \vdash T \triangleright \Delta_2 \qquad \Delta_2 \vdash S \triangleright \Delta_1}{\Gamma \vdash S \circ T \triangleright \Delta_1}\,(\text{Comp})$$

FIG 3 4 *Typing rules for the $\lambda_{\mathcal{L}}$-calculus*

$$\frac{}{A\,\Gamma \vdash \underline{1}\;\; A}\,(\text{Var}) \qquad \frac{\Gamma \vdash \underline{n}\;\; B}{A\,\Gamma \vdash \underline{n+1}\;\; B}\,(\text{Varn})$$

$$\frac{A\,\Gamma \vdash N\;\; B}{\Gamma \vdash \lambda_A\, N\;\; A \rightarrow B}\,(\text{Lambda}) \qquad \frac{\Gamma \vdash N\;\; A \rightarrow B \qquad \Gamma \vdash M\;\; A}{\Gamma \vdash (N\;\; M)\;\; B}\,(\text{App})$$

$$\frac{\Gamma_{\geq i} \vdash N\;\; B \qquad \Gamma_{<i}\, B\, \Gamma_{\geq i} \vdash M\;\; A}{\Gamma \vdash M\, \sigma^i N\;\; A}\,(\text{Sigma}) \qquad \frac{\Gamma_{\leq k}\, \Gamma_{>k+i} \vdash M\;\; A}{\Gamma \vdash \varphi^i_k M\;\; A}\,(\text{Phi})$$

FIG 3 5 *Typing rules for the λs_e-calculus*

and for the typed λs_e-calculus

$$\begin{array}{llll}
(\lambda_A\, M\;\; N) & \longrightarrow & M\, \sigma^1 N & (\sigma\text{-generation}) \\
(\lambda_A\, M)\, \sigma^i N & \longrightarrow & \lambda_A\, (M\, \sigma^{i+1} N) & (\sigma\text{-}\lambda\text{-transition}) \\
\varphi^i_k (\lambda_A\, M) & \longrightarrow & \lambda_A\, (\varphi^i_{k+1}\, M) & (\varphi\text{-}\lambda\text{-transition}) \\
\lambda_A\, (M\;\; \underline{1}) & \longrightarrow & N \quad \text{if } M =_{s_e} \varphi^2_0 N & (\text{Eta})
\end{array}$$

Typing rules for the $\lambda_{\mathcal{L}}$-calculus and the λs_e-calculus are presented in the Figures 3 4 and 3 5, respectively Notice that in the case of the $\lambda_{\mathcal{L}}$-calculus, substitutions receive contexts as types This is denoted as $\Gamma \vdash S \triangleright \Delta$ Let Γ be a context of the form $A_1\; A_2\;\; A_n\; \Delta$ We use the notation $\Gamma_{\leq k}$ and $\Gamma_{\geq k}$ for denoting the contexts $A_1\;\; A_k$ and $A_k\;\; A_n\; \Delta$, respectively This notation is extended for "<" and ">" in the obvious manner

EXAMPLE 3 4 In order to illustrate the use of the typing rules, we show how to infer the type of the term $\lambda_{A\rightarrow B}\; \lambda_{B\rightarrow C}\; \lambda_A\; (\underline{2}\;\; (\underline{3}\;\; \underline{1}))$ in λs_e

For short, let $\Gamma = A\; B{\rightarrow}C\; A{\rightarrow}B$ Firstly, observe that

$$\frac{}{(1)\;\Gamma \vdash \underline{1}\;\; A}\,(\text{Var}) \qquad \frac{\dfrac{}{B{\rightarrow}C\; A{\rightarrow}B \vdash \underline{1}\;\; B{\rightarrow}C}\,(\text{Var})}{(2)\;\Gamma \vdash \underline{2}\;\; B{\rightarrow}C}\,(\text{Varn}) \qquad \frac{\dfrac{\dfrac{}{A{\rightarrow}B \vdash \underline{1}\;\; A{\rightarrow}B}\,(\text{Var})}{B{\rightarrow}C\; A{\rightarrow}B \vdash \underline{2}\;\; A{\rightarrow}B}\,(\text{Varn})}{(3)\;\Gamma \vdash \underline{3}\;\; A{\rightarrow}B}\,(\text{Varn})$$

Then, we have

$$\frac{(2) \qquad \dfrac{(3) \qquad (1)}{\Gamma \vdash (\underline{3}\;\; \underline{1})\;\; B}\,(\text{App})}{\Gamma \vdash (\underline{2}\;\; (\underline{3}\;\; \underline{1}))\;\; C}\,(\text{App})$$

Finally, notice that

$$\cfrac{\cfrac{\cfrac{\Gamma \vdash (\underline{2}\ (\underline{3}\ \underline{1}))\ \ C}{B{\to}C\ A{\to}B \vdash \lambda_A\ (\underline{2}\ (\underline{3}\ \underline{1}))\ \ A{\to}C}\ (\text{Lambda})}{A{\to}B \vdash \lambda_{B\to C}\ \lambda_A\ (\underline{2}\ (\underline{3}\ \underline{1}))\ \ (B{\to}C){\to}(A{\to}C)}\ (\text{Lambda})}{\vdash \lambda_{A\to B}\ \lambda_{B\to C}\ \lambda_A\ (\underline{2}\ (\underline{3}\ \underline{1}))\ \ (A{\to}B){\to}((B{\to}C){\to}(A{\to}C))}\ (\text{Lambda})$$

For the $\lambda_{\mathcal{L}}$-calculus the inference is identical except for the first steps, for instance, notice that

$$\cfrac{\cfrac{\cfrac{}{B{\to}C\ A{\to}B \vdash {\uparrow}^0 \rhd B{\to}C\ A{\to}B}\ (\text{Id})}{\Gamma \vdash {\uparrow}^1 \rhd B{\to}C\ A{\to}B}\ (\text{Shift}) \quad \cfrac{\cfrac{}{A{\to}B \vdash {\uparrow}^0 \rhd A{\to}B}\ (\text{Id})}{B{\to}C\ A{\to}B \vdash {\uparrow}^1 \rhd A{\to}B}\ (\text{Shift})}{\Gamma \vdash {\uparrow}^2 \rhd A{\to}B}\ (\text{Comp})$$

Then,

$$\cfrac{\Gamma \vdash {\uparrow}^2 \rhd A{\to}B \quad \cfrac{}{A{\to}B \vdash \underline{1}\ \ A{\to}B}\ (\text{Var})}{\Gamma \vdash \underline{3}\ \ A{\to}B}\ (\text{Clos})$$

Remember that the language of the $\lambda_{\mathcal{L}}$-calculus only includes the de Bruijn index $\underline{1}$ and the others are simulated using the ${\uparrow}^n$ •

The simply-typed versions of the $\lambda_{\mathcal{L}}$- and λs_e-calculus satisfy, among others, the properties of subject reduction and type uniqueness Additionally, they are Weakly Normalizing (WN) and Church-Rosser (CR)

4. Applications. Although in an intuitionistic logic, the concepts of *propositions* and *types* are identified, proof construction and term synthesis do not necessarily go in the same direction For instance, to prove the proposition $A{\to}(B{\to}A)$, one may assume A as an hypothesis and then, recursively, try to prove $(B{\to}A)$ Eventually, one gets the axiom $A, B \vdash A$ and the proof derivation is completed On the other hand, the proof synthesis procedure decorates with λ-terms the proof-tree derivation from the axioms, i e , $x\ A, y\ B \vdash x\ \ A$, down to the conclusion, i e , $\vdash \lambda x\ A\ \lambda y\ B\ x\ \ A{\to}(B{\to}A)$

In order to synthesize a λ-term at the same time as a proof is being developed, it is necessary to represent *incomplete-proofs* Assume, for example, the proposition $A{\to}(B{\to}A)$ The bottom-up application of the rule (Abs) results in a term $\lambda x\ A\ X$ where X is a term to be constructed of type $(B{\to}A)$ A term as $\lambda x\ A\ X$ is called an *open term* and the place-holder X denotes a hole to be filled with a term of the right type, in this case of type $(B{\to}A)$ Place-holders are also called *meta-variables* to distinguish them from the variables of the λ-calculus Meta-variables are written as uppercase last letters of the Latin alphabet $X, Y,$ At some moment during the proof derivation, we get the typing judgment $x\ A, \Gamma \vdash \lambda y\ B\ x\ \ (B{\to}A)$ Hence, to obtain a close term, i e , a term without meta-variables, we can *instantiate* the meta-variable X with the term $\lambda y\ B\ x$ This results in $\lambda x\ A\ \lambda y\ B\ x$ In contrast to substitution of variables, instantiation of meta-variables is a first order replacement that does not take care of renaming of bound variables or capture of free-variables

Notice, however, that open terms are not λ-terms In fact, (1) instantiation and β-reduction do not commute, and (2) instantiation and typing do not commute To illustrate the first point, take the open term $(\lambda x\ X\ \ y)$ and the instantiation of X with x The instantiation results in $(\lambda x\ x\ \ y)$, which β-reduces to y However, the original term β-reduces to X, which gets instantiated as x To see why instantiation and typing do not commute, consider the context $\Gamma = x\ A, z\ (B{\to}A){\to}C$ and the open term $(z\ \ \lambda x\ B\ X)$ of type C, where X is a meta-variable of type A If we instantiate X with the variable x of Γ, then we obtain the ill-typed term $(z\ \ \lambda x\ B\ x)$

Meta-variables can be encoded in classical λ-calculus by using a technique taken from the higher order unification tradition [31] This technique uses a functional handle of scope For instance, the *open* term $\lambda x\ A\ Y$, where Y is a meta-variable of type B, is encoded as the λ-term $\lambda x\ A\ (y\ \ x)$, where y is a fresh variable of type $A{\to}B$ In this case, the information that the variable x can indeed occur in a subsequent substitution of y is taking into account by the application $(y\ \ x)$ Thus, an instantiation of Y with M in the original problem is translated as a substitution of y by $\lambda x\ A{\to}B\ M$ in the λ-calculus Notice, however that the meta-variable Y has the type B while the corresponding variable y has the type $A{\to}B$

Explicit substitutions and de Bruijn indices allow a simple and natural notation for open terms First, in a de Bruijn setting, meta-variables are just variables of the free algebra of terms Notice that bound and free variables of the λ-calculus are represented as indices And second, explicit substitution calculi as $\lambda\sigma$, $\lambda_{\mathcal{L}}$, and λs_e, are confluent on open terms (in the case of $\lambda\sigma$ and $\lambda_{\mathcal{L}}$, on substitution-closed terms) Thus, in these calculi, commutation of instantiation and the β-reduction is for free

We will consider meta-variables over a set $\mathcal{X}$

DEFINITION 4 1 *The set* $\Lambda_{dB}(\mathcal{X})$ *of* λ-terms in de Bruijn index notation *with meta-variables over the set* $\mathcal{X}$ *is defined inductively as*

$$M, N \ = \underline{n} \mid X \mid (M\ \ N) \mid \lambda M$$

where $n \in \mathbb{N}^{>0}, X \in \mathcal{X}$

DEFINITION 4 2 *A* valuation *is a mapping from* $\mathcal{X}$ *to* $\Lambda_{dB}(\mathcal{X})$ *The homeomorphic extension of a valuation,* θ*, from its domain* $\mathcal{X}$ *to the domain* $\Lambda_{dB}(\mathcal{X})$ *is called the* grafting *of* θ

As usual valuations and their corresponding graftings are denoted by the same Greek letters Application of a grafting θ to a term M will be written in postfix notation $M\theta$ For explicit representation of a valuation and its corresponding grafting θ, we use the notation $\theta = \{X \mapsto X\theta \mid X \in Dom(\theta)\}$ A grafting is the formal concept for meta-variable instantiation

The set of $\lambda\sigma$-, $\lambda_{\mathcal{L}}$-, and λs_e-terms with meta-variables, and their respective *grafting* notion, can be defined in a similar way The typing rule for meta-variables in these systems is [21]

$$\frac{}{\Gamma_X \vdash X\ \ A_X}\ (\mathrm{Meta}_{\mathcal{X}})$$

where A_X and Γ_X are, respectively, a *unique* type and a *unique* context associate to each meta-variable By using this rule, typing and instantiation of meta-variables commute [21]

4.1. Higher order unification. Higher order unification (HOU) is essential in automated reasoning, where it has formed the basis for generalizations of the Resolution Principle in higher order logics, being a *sine qua non* mechanism in the implementation of higher order proof assistants and higher order logic programming languages as the ones previously referenced For a very simple presentation of HOU see [57] and for a detailed introduction in the context of declarative programming see [53] As for the first order case, substitution is the key operation for HOU and its implicitness makes difficult the analysis of important computational properties Therefore, use of calculi of explicit substitution in the formal implementation of HOU procedures is relevant

HOU problems are expressed in the language of the simply-typed λ-calculus in de Bruijn indices over a set of meta-variables $\mathcal{X}$, denoted $\Lambda_{dB}(\mathcal{X})$ Meta-variables play the role of unification variables A simple example of a HOU problem is to search for function solutions F of the equality $F(f(a)) =^? f(F(a))$ That can be written in $\Lambda_{dB}(\mathcal{X})$ as $(X\ \ (\underline{2}\ \ \underline{1})) =^?_{\beta\eta} (\underline{2}\ \ (X\ \ \underline{1}))$, where both X and $\underline{2}$ are of functional type, say $A{\to}A$

and $\underline{1}$ of atomic type A A solution for X is the function identity, $\lambda_A\ \underline{1}$ but $\{\lambda_A\ (\underline{3}\ \underline{1}), \lambda_A\ (\underline{3}\ (\underline{3}\ \underline{1})), \quad \}$ (correspondingly, $\{F = f, F = f^2, \quad \}$) are solutions too

The first person to present a HOU algorithm of practical interest was Huet [31] Huet's work was relevant because he realized that to generalize Robinson first order Resolution Principle [55] to higher order theories it is useful to verify the existence of unifiers without computing them explicitly Huet's algorithm is a semi-decision one that may never stop when the input unification problem has no unifiers, but when the problem has a solution it always presents an explicit unifier Unification for second-order logic was proved undecidable in general by Goldfarb [26] Goldfarb's proof is based on a reduction from Hilbert's Tenth Problem This result shows that there are arbitrary higher order theories where unification is undecidable, but there exist particular higher order languages of practical interest that have a decidable unification problem In particular, for the second-order case, unification is decidable, when the language is restricted to monadic functions [23] Another problem of HOU is that the notion of most general unifier does not apply and that a notion more complex than the one of complete set of unifiers is necessary Huet has showed that equations of the form $(\lambda x\ F\quad a) =^? (\lambda x\ G\quad b)$ (called *flex-flex*) of third-order may not have minimal complete sets of unifiers and that there may exist an infinite chain of unifiers, one more general than the other, without having a most general one (for references see section 4 1 in [53])

The general method of HOU via calculi of explicit substitutions was introduced in [21] (for the $\lambda\sigma$-calculus) and consists mainly in firstly, a translation or "pre-cooking" from HOU problems in $\Lambda_{dB}(\mathcal{X})$ into the language of a calculus of explicit substitutions Secondly, an application of (first order) unification in the selected calculus of explicit substitutions to solve the translated problems Finally, translation back of the given grafting solutions into substitution solutions of the original HOU problem In this way HOU problems are solved via first order unification in the language of calculi of explicit substitution We will explain with examples how reduction relations from the simply-typed $\lambda\sigma$-calculus and λs_e-calculus of explicit substitutions are used to solve HOU problems in $\Lambda_{dB}(\mathcal{X})$ For a formal presentation of the methods consult [21] and [2]

DEFINITION 4 3 *Let* $\theta = \{X_1 \mapsto a_1, \quad , X_n \mapsto a_n\}$ *be a valuation from the set of meta-variables* $\mathcal{X}$ *to* $\Lambda_{dB}(\mathcal{X})$ *The corresponding* substitution, $\{a_1/X_1, \quad , a_n/X_n\}$, *also denoted by* θ *but written in a prefix notation, is defined inductively as follows*

1 $\theta(\underline{m}) = \underline{m}$, *for* $m \in \mathbb{N}$,

2 $\theta(X) = X\{X_1 \mapsto a_1, \quad , X_n \mapsto a_n\}$, *for* $X \in \mathcal{X}$,

3 $\theta(a_1\ \ a_2) = (\theta(a_1)\ \ \theta(a_2))$,

4 $\theta(\lambda a_1) = \lambda\theta^+(a_1)$,

where θ^+ *denotes the substitution corresponding to the valuation* $\theta^+ = \{X_1 \mapsto a_1^+, \quad , X_n \mapsto a_n^+\}$

Unifying two terms M and N in $\Lambda_{dB}(\mathcal{X})$ consists in finding a *grafting* θ such that its corresponding substitution satisfies $\theta(M) =_{\beta\eta} \theta(N)$ Notice that application of a grafting has a different effect to the application of its corresponding substitution For instance, although $(\lambda X)\{X \mapsto M\} = \lambda M$, a unifier of the problem $\lambda X =^?_{\beta\eta} \lambda M$ is not $\{M/X\}$, since $(\lambda X)\{M/X\} = \lambda(X\{M^+/X\}) = \lambda M^+$ However, by translating appropriately the $\Lambda_{dB}(\mathcal{X})$-terms M, N, the HOU problem $M =^?_{\beta\eta} N$ can be reduced to first order unification either in the $\lambda\sigma$- or in the λs_e-calculus Essentially, the pre-cooking translation from terms in $\Lambda_{dB}(\mathcal{X})$ into the language of the $\lambda\sigma$-calculus replaces each occurrence of a meta-variable X with $X[\uparrow^k]$, where k is the number of abstractors above the occurrence of X For the case of the λs_e-calculus the pre-cooking translates each occurrence of a meta-variable X into $\varphi_0^{k+1}X$, where k is as before

EXAMPLE 4 4 Consider the problem $\underline{2} =^?_{\beta\eta} (X\ \ \underline{2})$ being $\underline{2}$ of type A and X of type $A{\rightarrow}A$ Introducing a fresh meta-variable Y of type A the problem is translated into $\underline{2} =^?_{\beta\eta} (\lambda Y\ \ \underline{2}) \wedge X =^?_{\beta\eta} \lambda Y$

In the λs_e-calculus the problem is normalized into $\underline{2} =^?_{\lambda s_e} Y\sigma^1\underline{2} \wedge X =^?_{\lambda s_e} \lambda Y$, whose solutions are $\{\underline{1}/Y\}$ and $\{\underline{3}/Y\}$ giving as result the solutions $\{\lambda\underline{1}/X\}$ and $\{\lambda\underline{3}/X\}$

In the $\lambda\sigma$-calculus the problem is normalized into $\underline{2} =^?_{\lambda\sigma} Y[\underline{2}\ id] \wedge X =^?_{\lambda\sigma} \lambda Y$, from which we infer the solutions above •

EXAMPLE 4 5

Now consider the HOU problem $\underline{2} =^?_{\beta\eta} (\lambda Z\ \underline{2})$, where $\underline{2}$ and Z are of type A

In the λs_e-calculus the problem is pre-cooked into $\underline{2} =^?_{\lambda s_e} (\lambda\varphi^2_0 Z\ \underline{2})$ and then transformed into $\underline{2} =^?_{\lambda s_e} (\varphi^2_0 Z)\sigma^1\underline{2}$ and subsequently into $\underline{2} =^?_{\lambda s_e} \varphi^1_0 Z$ by normalization The sole possible solution given is $\{Z \mapsto \underline{2}\}$ Observe, on the one side, that $(\lambda\varphi^2_0 Z\ \underline{2})\{Z \mapsto \underline{2}\} = (\lambda\varphi^2_0\underline{2}\ \underline{2}) =_{\lambda s_e} (\lambda\underline{3}\ \underline{2}) =_{\lambda s_e} \underline{3}\sigma^1\underline{2} =_{\lambda s_e} \underline{2}$ On the other side, turning back the pre-cooking transformation, this corresponds to the substitution solution $\{\underline{2}/Z\}$ for the original problem In fact, $(\lambda Z\ \underline{2})\{\underline{2}/Z\} = ((\lambda Z)\{\underline{2}/Z\}\ \underline{2}\{\underline{2}/Z\}) = (\lambda(Z\{\underline{2}^+/Z\})\ \underline{2}) = (\lambda\underline{3}\ \underline{2})$ The previous term β-reduces into $\underline{2}$

In the $\lambda\sigma$-calculus the problem is pre-cooked into $\underline{1}[\uparrow] =^?_{\lambda\sigma} (\lambda Z[\uparrow]\ \underline{1}[\uparrow])$ which $\lambda\sigma$-reduces into $\underline{1}[\uparrow] =^?_{\lambda\sigma} (Z[\uparrow])[\underline{1}[\uparrow]\ id]$ and subsequently into $\underline{1}[\uparrow] =^?_{\lambda\sigma} Z[\uparrow \circ (\underline{1}[\uparrow]\ id)]$ and into $\underline{1}[\uparrow] =^?_{\lambda\sigma} Z[id]$ and finally into $\underline{1}[\uparrow] =^?_{\lambda\sigma} Z$ giving the corresponding sole solution $\{Z \mapsto \underline{1}[\uparrow]\}$ This corresponds to the above grafting solution in λs_e On the one side, $(\lambda Z[\uparrow]\ \underline{1}[\uparrow])\{Z \mapsto \underline{1}[\uparrow]\} = (\lambda((\underline{1}[\uparrow])[\uparrow])\ \underline{1}[\uparrow]) =_{\lambda\sigma} (\lambda\underline{1}[\uparrow^2]\ \underline{1}[\uparrow]) =_{\lambda\sigma} \underline{1}[\uparrow^2][\underline{1}[\uparrow]\ id] =_{\lambda\sigma} \underline{1}[\uparrow^2 \circ (\underline{1}[\uparrow]\ id)] =_{\lambda\sigma} 1[\uparrow]$ On the other side, turning back the pre-cooking transformation, this corresponds to the substitution solution $\{\underline{2}/Z\}$ for the original problem in $\Lambda_{dB}(\mathcal{X})$ as above

Notice that $\{\underline{1}/Z\}$ is not a substitution solution of the previous problem, since for any de Bruijn index $\underline{n}$ we have $(\lambda Z)\{\underline{n}/Z\} = \lambda(Z\{\underline{n}^+/Z\}) = \lambda(\underline{n+1})$ •

The following example illustrates why pre-cooking of λ-terms before applying unification rules is essential

EXAMPLE 4 6 (Continuing example 4 5) In the λs_e-calculus, when normalizing the HOU problem $\underline{2} =^?_{\beta\eta} (\lambda Z\ \underline{2})$ before pre-cooking we obtain $\underline{2} =^?_{\lambda s_e} Z\sigma^1\underline{2}$, whose solutions are the graftings $\{Z \mapsto \underline{1}\}$ and $\{Z \mapsto \underline{3}\}$ As previously mentioned $\{\underline{1}/Z\}$ is not a substitution solution of the original HOU problem Analogously, in the $\lambda\sigma$-calculus, when normalizing the corresponding problem $\underline{1}[\uparrow] =^?_{\lambda\sigma} (\lambda Z\ \underline{1}[\uparrow])$ we obtain $\underline{1}[\uparrow] =^?_{\lambda\sigma} \lambda Z[\underline{1}[\uparrow]\ id]$, whose solutions are $\{Z \mapsto \underline{1}\}$ and $\{Z \mapsto \underline{1}[\uparrow^2]\}$ given rise to the same problem •

4.2. Type inference. In order to infer types of λ-terms (or $\lambda\sigma$-terms or λs_e-terms) we deal with new sets of type variables τ_i and context variables γ_i, $i \in \mathbb{N}$ Essentially, we will take as input of a type inference problem a term without knowing its type and context and as output we will formulate a first order unification problem on type and context variables Well-typedness of the input term will then correspond to solvability of the generated first order unification problem Here we illustrate the general method mentioned above using the language of the λs_e-calculus Simple modifications according to the typing rules of the selected language will adapt this method to other settings

Let M be a λs_e-term Initially, we introduce new variables for the type and for the context of each subterm of M Then M can be seen as a new term M' with all its subterms *decorated* with one different type variable as subscript and one different context variable as superscript

EXAMPLE 4 7 $(\lambda_A\,(\lambda_B\,(\lambda_C\,(\underline{2}^{\gamma_1}_{\tau_1}\ (\underline{3}^{\gamma_2}_{\tau_2}\ \underline{1}^{\gamma_3}_{\tau_3})^{\gamma_4}_{\tau_4})^{\gamma_5}_{\tau_5})^{\gamma_6}_{\tau_6})^{\gamma_7}_{\tau_7})^{\gamma_8}_{\tau_8}$, where τ_i and γ_i, $i = 1, \ ,8$ are new mutually different type and context variables, is the decorated version of the λ-term $\lambda_A\ \lambda_B\ \lambda_C\ (\underline{2}\ (\underline{3}\ \underline{1}))$ •

Afterwards, we apply the set of transformation rules in Table 4 1 for pairs of the form $\langle R, E\rangle$, where R is a set of decorated terms and E a set of equations on type and context variables The application of these transformation rules begin from the pair $\langle R_0, \emptyset\rangle$, where R_0 is the set of all decorated subterms of M'

Table 4 1

Transformation rules for type inference in the λs_e-calculus

(Var)	$\langle R \cup \{\underline{1}^{\gamma}_{\tau}\}, E\rangle$	$\to$	$\langle R, E \cup \{\gamma = \tau\,\gamma'\}\rangle$, where γ' is a fresh context variable,
(Varn)	$\langle R \cup \{\underline{n}^{\gamma}_{\tau}\}, E\rangle$	$\to$	$\langle R, E \cup \{\gamma = \tau'_1 \ \ \tau'_{n-1}\,\tau\,\gamma'\}\rangle$, where γ' and $\tau'_1, \ \ , \tau'_{n-1}$ are fresh context and type variables,
(Lambda)	$\langle R \cup \{(\lambda_A\ M^{\gamma_1}_{\tau_1})^{\gamma_2}_{\tau_2}\}, E\rangle$	$\to$	$\langle R, E \cup \{\tau_2 = A \to \tau_1, \gamma_1 = A\,\gamma_2\}\rangle$,
(App)	$\langle R \cup \{(M^{\gamma_1}_{\tau_1}\ \ N^{\gamma_2}_{\tau_2})^{\gamma_3}_{\tau_3}\}, E\rangle$	$\to$	$\langle R, E \cup \{\gamma_1 = \gamma_2, \gamma_2 = \gamma_3, \tau_1 = \tau_2 \to \tau_3\}\rangle$,
(Sigma)	$\langle R \cup \{(M^{\gamma_1}_{\tau_1}\sigma^{i} N^{\gamma_2}_{\tau_2})^{\gamma_3}_{\tau_3}\}, E\rangle$	$\to$	$\langle R, E \cup \{\tau_1 = \tau_3, \gamma_1 = \tau'_1 \ \ \tau'_{i-1}\,\tau_2\,\gamma_2, \gamma_3 = \tau'_1 \ \ \tau'_{i-1}\,\gamma_2\}\rangle$, where $\tau'_1, \ \ , \tau'_{i-1}$ are fresh type variables and in the case that $i = 1$ the sequence $\tau'_1 \ \ \tau'_{i-1}$ is empty,
(Phi)	$\langle R \cup \{(\varphi^{i}_{k} M^{\gamma_1}_{\tau_1})^{\gamma_2}_{\tau_2}\}, E\rangle$	$\to$	$\langle R, E \cup \{\tau_1 = \tau_2, \gamma_2 = \tau'_1 \ \ \tau'_{k+i-1}\,\gamma', \gamma_1 = \tau'_1 \ \ \tau'_{k-1}\,\gamma'\}\rangle$, where γ' and $\tau'_1, \ \ , \tau'_{k+i-1}$ are fresh context and type variables and in the case that $k \leq 1$ respectively $k = 0$ and $i = 1$ the sequences $\tau'_1 \ \ \tau'_{k-1}$ respectively $\tau'_1 \ \ \tau'_{k+i-1}$ are empty,
(Meta)	$\langle R \cup \{X^{\gamma}_{\tau}\}, E\rangle$	$\to$	$\langle R, E \cup \{\gamma = \Gamma_X, \tau = A_X\}\rangle$, where $\Gamma_X \vdash X \ \ A_X$,

Notice that the transformation rules in the Table 4 1 are built according to the typing rules of the λs_e-calculus After the application of each of the transformation rules the size of the current set of decorated subterms R decreases by one Consequently, the application of these rules beginning from the pair $\langle R_0, \emptyset\rangle$ finishes after a finite number of steps (exactly as many steps as subterms in M) giving as result an empty set of decorated terms and a set E_f of equation on type and context variables E_f is a first order unification problem on type and context variables

Finally, our algorithm terminates by applying any first order unification algorithm to E_f If the unification algorithm fails then our term is ill-typed Otherwise, if the unification algorithm succeeds, the most general unifier resulting as output gives straightforwardly a context Γ and a type A such that $\Gamma \vdash M \ \ A$ Of course, the construction of Γ and A is done from the bindings given in the resulting unifier corresponding to the outermost context and type variables selected in the decoration of M

Correctness and completeness of this method is a direct consequence from the correctness and completeness of the first order unification and of the typing rules of the λs_e-calculus used to construct the transformation rules in Table 4 1

Example 4 8 (Continuing Example 4 7) The initial input for the set of inference rules is $\langle R_0, \emptyset\rangle$, where

$$R_0 = \{\underline{2}^{\gamma_1}_{\tau_1}, \underline{3}^{\gamma_2}_{\tau_2}, \underline{1}^{\gamma_3}_{\tau_3}, (\underline{3}^{\gamma_2}_{\tau_2}\ \ \underline{1}^{\gamma_3}_{\tau_3})^{\gamma_4}_{\tau_4}, (\underline{2}^{\gamma_1}_{\tau_1}\ \ (\underline{3}^{\gamma_2}_{\tau_2}\ \ \underline{1}^{\gamma_3}_{\tau_3})^{\gamma_4}_{\tau_4})^{\gamma_5}_{\tau_5}, (\lambda_C\ (\underline{2}^{\gamma_1}_{\tau_1}\ \ (\underline{3}^{\gamma_2}_{\tau_2}\ \ \underline{1}^{\gamma_3}_{\tau_3})^{\gamma_4}_{\tau_4})^{\gamma_5}_{\tau_5})^{\gamma_6}_{\tau_6},$$
$$(\lambda_B\ (\lambda_C\ (\underline{2}^{\gamma_1}_{\tau_1}\ \ (\underline{3}^{\gamma_2}_{\tau_2}\ \ \underline{1}^{\gamma_3}_{\tau_3})^{\gamma_4}_{\tau_4})^{\gamma_5}_{\tau_5})^{\gamma_6}_{\tau_6})^{\gamma_7}_{\tau_7}, (\lambda_A\ (\lambda_B\ (\lambda_C\ (\underline{2}^{\gamma_1}_{\tau_1}\ \ (\underline{3}^{\gamma_2}_{\tau_2}\ \ \underline{1}^{\gamma_3}_{\tau_3})^{\gamma_4}_{\tau_4})^{\gamma_5}_{\tau_5})^{\gamma_6}_{\tau_6})^{\gamma_7}_{\tau_7})^{\gamma_8}_{\tau_8}\}$$

In the sequel, we show the steps of the application of the transformation rules For convenience we apply the rules in an specific order (from smaller to bigger subterms), but the application of the rules is nondeterministic Applying the rules in any order we will obtain different sets of equations that correspond to the same unification problem

$$
\begin{array}{ll}
\langle R_0, \emptyset\rangle & \to Var \\
\langle R_1 = R_0 \setminus \{\underline{1}^{\gamma_3}_{\tau_3}\}, E_1 = \{\gamma_3 = \tau_3\ \gamma'_1\}\rangle & \to Varn \\
\langle R_2 = R_1 \setminus \{\underline{2}^{\gamma_1}_{\tau_1}\}, E_2 = E_1 \cup \{\gamma_1 = \tau'_1\ \tau_1\ \gamma'_2\}\rangle & \to Varn \\
\langle R_3 = R_2 \setminus \{\underline{3}^{\gamma_2}_{\tau_2}\}, E_3 = E_2 \cup \{\gamma_2 = \tau'_2\ \tau'_3\ \tau_2\ \gamma'_3\}\rangle & \to App \\
\langle R_4 = R_3 \setminus \{(\underline{3}^{\gamma_2}_{\tau_2}\ \ \underline{1}^{\gamma_3}_{\tau_3})^{\gamma_4}_{\tau_4}\}, E_4 = E_3 \cup \{\gamma_2 = \gamma_3, \gamma_3 = \gamma_4, \tau_2 = \tau_3 \to \tau_4\}\rangle & \to App \\
\langle R_5 = R_4 \setminus \{(\underline{2}^{\gamma_1}_{\tau_1}\ \ (\underline{3}^{\gamma_2}_{\tau_2}\ \ \underline{1}^{\gamma_3}_{\tau_3})^{\gamma_4}_{\tau_4})^{\gamma_5}_{\tau_5}\}, E_5 = E_4 \cup \{\gamma_1 = \gamma_4, \gamma_4 = \gamma_5, \tau_1 = \tau_4 \to \tau_5\}\rangle & \to Lambda \\
\langle R_6 = R_5 \setminus \{(\lambda_C\ (\underline{2}^{\gamma_1}_{\tau_1}\ \ (\underline{3}^{\gamma_2}_{\tau_2}\ \ \underline{1}^{\gamma_3}_{\tau_3})^{\gamma_4}_{\tau_4})^{\gamma_5}_{\tau_5})^{\gamma_6}_{\tau_6}\}, E_6 = E_5 \cup \{\tau_6 = C \to \tau_5, \gamma_5 = C\ \gamma_6\}\rangle & \to Lambda \\
\langle R_7 = R_6 \setminus \{(\lambda_B\ (\lambda_C\ (\underline{2}^{\gamma_1}_{\tau_1}\ \ (\underline{3}^{\gamma_2}_{\tau_2}\ \ \underline{1}^{\gamma_3}_{\tau_3})^{\gamma_4}_{\tau_4})^{\gamma_5}_{\tau_5})^{\gamma_6}_{\tau_6})^{\gamma_7}_{\tau_7}\}, E_7 = E_6 \cup \{\tau_7 = B \to \tau_6, \gamma_6 = B\ \gamma_7\}\rangle & \to Lambda \\
\langle \emptyset = R_7 \setminus \{(\lambda_A\ (\lambda_B\ (\lambda_C\ (\underline{2}^{\gamma_1}_{\tau_1}\ \ (\underline{3}^{\gamma_2}_{\tau_2}\ \ \underline{1}^{\gamma_3}_{\tau_3})^{\gamma_4}_{\tau_4})^{\gamma_5}_{\tau_5})^{\gamma_6}_{\tau_6})^{\gamma_7}_{\tau_7})^{\gamma_8}_{\tau_8}\}, E_8 = E_7 \cup \{\tau_8 = A \to \tau_7, \gamma_7 = A\ \gamma_8\}\rangle &
\end{array}
$$

Now the reader is invited to apply his/her preferred first order unification algorithm for resolving the unification problem $E_8 = \{\gamma_3 = \tau_3\ \gamma'_1, \gamma_1 = \tau'_1\ \tau_1\ \gamma'_2, \gamma_2 = \tau'_2\ \tau'_3\ \tau_2\ \gamma'_3, \gamma_2 = \gamma_3, \gamma_3 = \gamma_4, \tau_2 = \tau_3 \to \tau_4, \gamma_1 = \gamma_4, \gamma_4 = \gamma_5, \tau_1 = \tau_4 \to \tau_5, \tau_6 = C \to \tau_5, \gamma_5 = C\ \gamma_6, \tau_7 = B \to \tau_6, \gamma_6 = B\ \gamma_7, \tau_8 = A \to \tau_7, \gamma_7 = A\ \gamma_8\}$ and then to resolve the bindings of the resulting unifier (if it exists) for giving appropriate contexts and types for the input λ-term •

4.3. Inhabitation and higher order logics. Given a type A and a context of variable declarations Γ, the inhabitation problem consists of finding a term M such that $\Gamma \vdash M\ \ A$ Using the open term approach, the problem can be formulated as finding a pure instantiation for the meta-variable X satisfying $\Gamma \vdash X\ \ A$ Thus, the term to instantiate X can be constructed at the same time as the proof derivation of A by applying the typing rules in a bottom-up manner and introducing new meta-variables for the unknown terms

For the simply-typed λ-calculus this problem is decidable In fact, since provability in the minimal propositional intuitionistic logic is decidable, the term M can be built directly from the proof-tree derivation of $\Omega \vdash_I A$, where Ω is the set of types in Γ, as explained before However, when we move to a first order or a higher order intuitionistic logic and, in consequence, we extend the type system to handle quantification, the problem becomes much more complicated In [47], a semi-algorithm to solve the inhabitation problem via the $\lambda_{\mathcal{L}}$-calculus has been presented It uses the fact that $\lambda_{\mathcal{L}}$ is confluent on substitution-closed terms and weakly normalizing, even for dependent type settings of the calculus

Although first and higher order logics are out of the scope of this paper, we give some hints of the inhabitation problem for these kind of logics See [20] for a complete description of a term synthesis algorithm in the Cube of Type Systems and [47] for a similar algorithm via explicit substitutions and open terms

The Dependent Type theory, namely $\lambda\Pi$ [29], is a conservative extension of the simply-typed λ-calculus It allows a finer stratification of terms by generalizing the function space type In fact, in $\lambda\Pi$, the type of a function $\lambda x\ A\ M$ is $\Pi x\ A\ B$ where B (the type of M) may depend on x Hence, the type $A \to B$ of the simply-typed λ-calculus is just a notation in $\lambda\Pi$ for the product $\Pi x\ A\ B$ where x does not appear free in B The Calculus of Constructions, namely CC, [15, 16] extends the $\lambda\Pi$-calculus with polymorphism and constructions of types From a logical point of view, $\lambda\Pi$ and CC allow representation of proofs in the first and higher order intuitionistic logic, respectively Via the types-as-proofs principle, a term of type $\Pi x\ A\ B$ is a proof-term of the proposition $\forall x\ A\ B$

Terms in these calculi can be variables, applications, or abstractions, like in classical λ-calculus, or two new kind of terms products ($\Pi x\ A\ B$), and sorts ($Type, Kind$) Term and types belong to the same syntactical category Thus, $\Pi x\ A\ B$ is a term, as well as $\lambda x\ A\ M$ However, terms are stratified in several levels according to a type discipline For instance, given an appropriate context of variable declarations,

$$\frac{x\ A, \Gamma \vdash M\ \ B \qquad x\ A, \Gamma \vdash B\ \ \{Type, Kind\}}{\Gamma \vdash \lambda x\ A\ M\ \ \Pi x\ A\ B}\text{(Abs)} \qquad \frac{\Gamma \vdash M\ \ \Pi x\ A\ B \qquad \Gamma \vdash N\ \ A}{\Gamma \vdash (M\ \ N)\ \ B\{N/x\}}\text{(Appl)}$$

FIG 4 1 *Rules (Abs) and (Apl) for the CC type system*

$\lambda x\ A\ \ M\ \ \ \Pi x\ A\ \ B$, $\Pi x\ A\ \ B\ \ \ Type$, and $Type\ \ \ Kind$ The term $Kind$ cannot be typed in any context, but it is necessary since a circular typing as $Type\ \ \ Type$ leads to the Girard's paradox [25] In Fig 4 1 we give rules (Abs) and (Appl) for the CC type system

The $\lambda_{\mathcal{L}}$-calculus has been extended with products for the $\lambda\Pi$ and CC-type systems in [45] These variants satisfy the same properties as the simply-typed version confluent on substitution-closed terms, weakly-normalizing, and subject reduction For further details we refer to [45]

EXAMPLE 4 9 We can proof the first order predicate $(\forall x\ (P\ x)) \rightarrow (P\ c)$ by finding a term X of type $(\Pi x\ A\ (P\ \ x)) \rightarrow (P\ \ c)$ in a context where the term c has the type A and P has the type $A \rightarrow Type$ The bottom-up application of rule (Abs) results in a term X having the form $\lambda y\ (\Pi x\ A\ (P\ \ x))\ Y$ where Y is a term of type $(P\ \ c)$ in a context where the variable y has the type $\Pi x\ A\ (P\ \ x)$ If we instantiate Y with the term $(y\ \ c)$, which is a well typed term of type $(P\ \ c)$, we obtain the term $\lambda y\ (\Pi x\ A\ (P\ \ x))\ (y\ \ c)$ of type $\Pi x\ (\Pi x\ A\ (P\ \ x))\ (P\ \ c)$ Notice that in this example we have used the meta-variables X and Y and the instantiation mechanism of meta-variables to build incrementally a proof •

Typing of meta-variables is more complicated in dependent-type systems than in the simply-type case Since meta-variables can appear in terms, types, and contexts, the typing rules should take care of possible circular dependences

5. Conclusion. The λ-calculus uses an external and atomic operation to compute the substitutions of variables by terms Calculi of explicit substitutions improve the substitution mechanism by allowing substitutions to be part of the formal language by means of special constructors and reduction rules There are several versions of calculi of explicit substitutions Figure 5 1 summarizes the main characteristics of some of them All these calculi implement the β-reduction by means of a lazy mechanism of reduction of substitutions

In this paper we have explored new developments and applications on two of the most successful styles of explicit substitution $\lambda\sigma$ and λs_e

REFERENCES

[1] M ABADI, L CARDELLI, P -L CURIEN, AND J -J LÉVY, *Explicit Substitutions*, Journal of Functional Programming, 1 (1991), pp 375–416

[2] M AYALA-RINCÓN AND F KAMAREDDINE, *Unification via λs_e-Style of Explicit Substitution*, in 2nd International Conference on Principles and Practice of Declarative Programming, Montreal, Canada, September 2000, ACM Press

[3] H P BARENDREGT, *The Lambda Calculus Its Syntax and Semantics (revised edition)*, North Holland, 1984

[4] ——, *λ-calculi with types*, Handbook of Logic in Computer Science, II (1992)

[5] B BARRAS, S BOUTIN, C CORNES, J COURANT, J FILLIATRE, E GIMÉNEZ, H HERBELIN,

	$\lambda\sigma$	$\lambda_{\mathcal{L}}$	λs_e	$\lambda\sigma_{\Uparrow}$	λ_{ς}	$\lambda\upsilon$	λ_d	$\lambda\mathrm{x}$	$\lambda\chi$
Confluence	Mv	Mv	$\smile$	$\smile$	$\smile$	Gnd	Gnd	Gnd	Gnd
Normalization	Wk	Wk	Wk	Wk	PSN	PSN	PSN	PSN	PSN
Composition	$\smile$	$\smile$	$\stackrel{**}{\frown}$	$\smile$	$\stackrel{**}{\frown}$	$\stackrel{**}{\frown}$	$\sim^{\star}$	$\stackrel{**}{\frown}$	$\stackrel{**}{\frown}$
Finitary 1st-order	$\smile$	$\smile$	$\stackrel{**}{\frown}$	$\smile$	$\smile$	$\smile$	$\smile$	$\stackrel{**}{\frown}$	$\stackrel{**}{\frown}$
Variables	dB	dB	dB	dB	dB	dB	dB	Nm	Lv
Number of rules	13	12	$13^{\dagger}$	19	13	8	19	6	$10^{\dagger}$
β-reduction	$\smile$	$\smile$	$\smile$	$\smile$	$\sim^{\ddagger}$	$\smile$	$\smile$	$\smile$	$\smile$
Reference	[1]	[44]	[32]	[17]	[43]	[38]	[35]	[8]	[39]

$\smile$	The general property holds
$\stackrel{**}{\frown}$	The property does not hold
$\sim$	The property holds with restrictions
Mv	Confluence on semi-open expressions, i e only with meta-variables of terms
Gnd	Confluence on ground expressions
Wk	Weak normalization on typed terms
PSN	Preservation of strong normalization
dB	De Bruijn indices notation of variables
Nm	Variable names
Lv	De Bruijn levels notation with variable names
$\star$	Restricted composition In particular, the λ_d-calculus does not allow simultaneous substitutions
$\dagger$	Number of schemes The λs_e-calculus is not finitary
$\ddagger$	Big-step semantic of β-reduction The λ_{ς}-calculus does not simulate each step of β-reduction

FIG 5 1 *Some calculi of explicit substitutions*

G HUET, C MUÑOZ, C MURTHY, C PARENT, C PAULIN, A SAIBI, AND B WERNER, *The Coq Proof Assistant Reference Manual – Version V6 1*, Tech Report 0203, INRIA, August 1997

[6] N BJØRNER AND C MUÑOZ, *Absolute explicit unification*, in International Conference on Rewriting Techniques and Applications (RTA'2000), vol. 1833 of Lecture Notes in Computer Science, Norwich, U K , July 2000, pp 31–46

[7] R BLOO, *Preservation of Termination for Explicit Substitution*, PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1997

[8] R BLOO AND K H ROSE, *Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection*, in Proc CSN-95 Computer Science in the Netherlands, Nov 1995

[9] D BRIAUD, *Higher order unification as a typed narrowing*, tech report, CRIN 96-R-112, 1996

[10] A CHURCH, *An unsolvable problem of elementary number theory*, American Journal of Mathematics, 58 (1936), pp 345–363

[11] ——, *A formulation of the simple theory of types*, Journal of Symbolic Logic, 5 (1940), pp 56–68

[12] ——, *The Calculi of Lambda-conversion*, Princeton University Press, 1941

[13] H CIRSTEA AND C KIRCHNER, *Combining Higher-order and First-order Computation Using ρ-Calculus Towards a Semantics of ELAN*, in Frontiers of Combining Systems 2, D M Gabbay and M de Rijke, eds , Studies on Logic and Computation, 7, Research Studies Press/Wiley, 1999, ch 6, pp 95–121

[14] ——, *Introduction to the Rewriting Calculus*, *Rapport de Recherche* 3818, INRIA, Dec 1999

[15] T COQUAND, *Une Théorie de Constructions*, thèse de doctorat, U Paris VII, 1985

[16] T COQUAND AND G HUET, *The Calculus of Constructions*, Information and Computation, 76 (1988), pp 96–120

[17] P -L CURIEN, T HARDIN, AND J -J LÉVY, *Confluence Properties of Weak and Strong Calculi of Explicit Substitutions*, Journal of the ACM, 43 (1996), pp 362–397 Also as *Rapport de Recherche* INRIA 1617, 1992

[18] H B CURRY AND R FEYS, *Combinatory Logic*, vol 1, North Holland, 1958

[19] N DE BRUIJN, *Lambda-calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem*, Indag Mat , 34(5) (1972), pp 381–392

[20] G DOWEK, *A complete proof synthesis method for type systems of the cube*, Journal of Logic and Computation, 3 (1993), pp 287–315

[21] G DOWEK, T HARDIN, AND C KIRCHNER, *Higher-order Unification via Explicit Substitutions*, Information and Computation, 157 (2000), pp 183–235

[22] G DOWEK, T HARDIN, C KIRCHNER, AND F PFENNING, *Unification via explicit substitutions The case of higher-order patterns*, in Proceedings of the Joint International Conference and Symposium on Logic Programming, M Maher, ed , Bonn, Germany, Sept 1996, MIT Press

[23] W FARMER, *A Unification Algorithm for Second-order Monadic Terms*, Annals of Pure and Applied Logic, 39 (1988), pp 131–174

[24] M C F FERREIRA, D KESNER, AND L PUEL, *Lambda-calculi with explicit substitutions and composition which preserve beta-strong normalization*, in Algebraic and Logic Programming, 5th International Conference, ALP'96, M Hanus and M Rodríguez-Artalejo, eds , vol 1139 of LNCS, Aachen, Germany, 25–27 Sept 1996, Springer, pp 284–298

[25] J -Y GIRARD, *Interprétation Fonctionelle et Élimination des Compures de l'Arithmétic d'Ordre Supérieur*, thèse de doctorat, Université Paris VII, 1972

[26] W GOLDFARB, *The Undecidability of the Second-order Unification Problem*, Theoretical Computer Science, 13 (1981), pp 225–230

[27] M GORDON AND T MELHAM, *Introduction to HOL A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993

[28] B GUILLAUME, *The λs_e-calculus Does Not Preserve Strong Normalization*, Journal of Functional Programming, 10 (2000), pp 321–325

[29] R HARPER, F HONSELL, AND G PLOTKIN, *A framework for defining logics*, Journal of the Association for Computing Machinery, 40 (1993), pp 143–184

[30] J R HINDLEY, *Basic Simple Type Theory*, no 42 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1997

[31] G HUET, *A Unification Algorithm for Typed λ-Calculus*, Theoretical Computer Science, 1 (1975), pp 27–57

[32] F KAMAREDDINE AND A RÍOS, *A λ-calculus à la de Bruijn with Explicit Substitutions*, in Proc of

PLILP'95, vol 982 of LNCS, Springer, 1995, pp 45–62

[33] ——, *Extending a λ-calculus with Explicit Substitution which Preserves Strong Normalisation into a Confluent Calculus on Open Terms*, Journal of Functional Programming, 7 (1997), pp 395–420

[34] F Kamareddine and A Ríos, *Relating the $\lambda\sigma$- and λs-Styles of Explicit Substitutions*, Journal of Logic and Computation, 10 (2000), pp 349–380

[35] D Kesner, *Confluence properties of extensional and non-extensional λ-calculi with explicit substitutions (extended abstract)*, in Proceedings of the Seventh International Conference on Rewriting Techniques and Applications (RTA-96), H Ganzinger, ed , vol 1103 of LNCS, New Brunswick, New Jersey, 1996, Springer-Verlag, pp 184–199

[36] C Kirchner and C Ringeissen, *Higher-order Equational Unification via Explicit Substitutions*, in Proc Algebraic and Logic Programming, vol 1298 of LNCS, Springer, 1997, pp 61–75

[37] S C Kleene, *λ-definability and recursiveness*, Duke Mathematical Journal, 2 (1936), pp 340-353

[38] P Lescanne, *From $\lambda\sigma$ to $\lambda\upsilon$ a Journey Through Calculi of Explicit Substitutions*, in Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1994, pp 60–69

[39] P Lescanne and J Rouyer-Degli, *Explicit substitutions with de Bruijn's levels*, in Proceedings of the International Conference on Rewriting Techniques and Applications (RTA-95), J Hsiang, ed , vol 914 of LNCS, Chapel Hill, North Carolina, 1995, Springer-Verlag, pp 294–308

[40] P -A Melliès, *Typed λ-calculi with explicit substitutions may not terminate*, LNCS, 902 (1995), pp 328–338

[41] D Miller, *λProlog An Introduction to the Language and Its Logic* Draft, Department of Computer Science and Engineering, The Pennsylvania State University, 1998

[42] R Milner, M Tofte, and R Harper, *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1991

[43] C Muñoz, *Confluence and preservation of strong normalisation in an explicit substitutions calculus (extended abstract)*, in Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, July 1996, IEEE Computer Society Press, pp 440–447

[44] ——, *A left-linear variant of $\lambda\sigma$*, in Proc International Conference PLILP/ALP/HOA'97, vol 1298 of LNCS, Southampton (England), September 1997, Springer, pp 224–234

[45] C Muñoz, *Un calcul de substitutions pour la représentation de preuves partielles en théorie de types*, PhD thesis, Université Paris 7, 1997 English version in *Rapport de recherche* INRIA RR-3309, 1997

[46] C Muñoz, *Dependent types and explicit substitutions* To appear in Mathematical Structures in Computer Science It also appears as report NASA/CR-1999-209722, ICASE No 99-43, 1999

[47] ——, *Proof-term synthesis on dependent-type systems via explicit substitutions* To appear in Theoretical Computer Science It also appears as report NASA/CR-1999-209730, ICASE No 99-47, 1999

[48] G Nadathur, *A fine-grained notation for lambda terms and its use in intensional operations*, Journal of Functional and Logic Programming, (1999)

[49] G Nadathur and D S Wilson, *A Notation for Lambda Terms A Generalization of Environments*, Theoretical Computer Science, 198 (1998), pp 49–98

[50] R P Nederpelt, J H Geuvers, and R C de Vrijer, *Selected papers on Automath*, North-Holland, Amsterdam, 1994

[51] S Owre, J M Rushby, and N Shankar, *PVS A prototype verification system*, in 11th Interna-

tional Conference on Automated Deduction (CADE), D Kapur, ed , vol 607 of Lecture Notes in Artificial Intelligence, Saratoga, NY, June 1992, Springer-Verlag, pp 748–752

[52] F Pfenning and C Schurmann, *Twelf user's guide, 1 2 edition*, Tech Report CMU-CS-1998-173, Carnegie Mellon University, September 1998

[53] C Prehofer, *Progress in Theoretical Computer Science*, in Solving Higher-Order Equations From Logic to Programming, R V Book, ed , Birkhauser, 1997

[54] A Ríos, *Contributions à l'étude de λ-calculs avec des substitutions explicites*, thèse de doctorat, Université Paris VII, 1993

[55] J A Robinson, *A Machine-oriented Logic Based on the Resolution Principle*, Journal of the ACM, 12 (1965), pp 23–41

[56] K H Rose, *Explicit Substitution - Tutorial & Survey*, BRICS, Lecture Series LS-96-3, Department of Computer Science, University of Aarhus, 1996

[57] W Snyder and J Gallier, *Higher-order Unification Revisited Complete Sets of Transformations*, Journal of Symbolic Computation, 8 (1989), pp 101–140

[58] A N Whitehead and B Russell, *Principia Mathematica*, Cambridge University Press, Cambridge, revised ed , 1925–1927 Three volumes The first edition was published 1910–1913

REPORT DOCUMENTATION PAGE			Form Approved OMB No 0704-0188

1 AGENCY USE ONLY(*Leave blank*)	2 REPORT DATE	3 REPORT TYPE AND DATES COVERED
	November 2000	Contractor Report

4 TITLE AND SUBTITLE
Explicit substitutions and all that

5 FUNDING NUMBERS
C NAS1-97046
WU 505-90-52-01

6 AUTHOR(S)
Mauricio Ayala-Rincón and César Muñoz

7 PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
ICASE
Mail Stop 132C
NASA Langley Research Center
Hampton, VA 23681-2199

8 PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No 2000-45

9 SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-2199

10 SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA/CR-2000-210621
ICASE Report No 2000-45

11 SUPPLEMENTARY NOTES
Langley Technical Monitor Dennis M Bushnell
Final Report
To appear in the Colombian Journal of Computation

12a DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited
Subject Category 60, 61
Distribution Nonstandard
Availability NASA-CASI (301) 621-0390

12b DISTRIBUTION CODE

13 ABSTRACT *(Maximum 200 words)*
Explicit substitution calculi are extensions of the λ-calculus where the substitution mechanism is internalized into the theory This feature makes them suitable for implementation and theoretical study of logic-based tools such as strongly typed programming languages and proof assistant systems In this paper we explore new developments on two of the most successful styles of explicit substitution calculi the $\lambda\sigma$- and λs_e-calculi

14 SUBJECT TERMS
explicit substitution, higher order unification, lambda-calculus, type theory, rewriting

15 NUMBER OF PAGES
27

16 PRICE CODE
A03

17 SECURITY CLASSIFICATION OF REPORT	18 SECURITY CLASSIFICATION OF THIS PAGE	19 SECURITY CLASSIFICATION OF ABSTRACT	20 LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298(Rev 2-89)
Prescribed by ANSI Std Z39-18
298-102

End of Document