Higher-Order Substitutions

Dominic Duggan

Department of Computer Science, Stevens Institute of Technology, Hoboken, New Jersey 07030

The $\lambda\sigma$-calculus is a concrete $\lambda$-calculus of explicit substitutions, designed for reasoning about implementations of $\lambda$-calculi. Higher-order abstract syntax is an approach to metaprogramming that explicitly captures the variable-binding aspects of programming language constructs. A new calculus of explicit substitutions for higher-order abstract syntax is introduced, allowing a high-level description of variable binding in object languages while also providing substitutions as explicit programmer-manipulable data objects. The new calculus is termed the $\lambda\sigma\beta_\omega$-calculus, since it makes essential use of an extension of $\beta_\omega$-unification (described in another paper). Termination and confluence are verified for the $\lambda\sigma\beta_\omega$-calculus similarly to that for the $\lambda\sigma$-calculus, and an efficient implementation is given in terms of first-order renaming substitutions. The verification of confluence makes use of a verified adaptation of Nipkow’s higher-order critical pairs lemma to the forms of rewrite rules required for the statement of the $\lambda\sigma\beta_\omega$-calculus.

1. INTRODUCTION

I don’t really like de Bruijn numbers myself.

—N. G. de Bruijn

Variable binding plays a central rôle both programming languages and mathematical logic, and in their recent combination in type theory and programming logics. It is the essence of the $\lambda$-calculus (Church 1941), the “kernel language” for all functional and Algol-like languages (Landin 1965, Strachey 1967, Reynolds 1981). It is also at the heart of predicate logic, various extensions of which have been proposed as programming logics, including Nuprl (Constable et al. 1986) and the Calculus of Constructions (Coquand and Huet 1988, Paulin-Mohring 1989).

Because of this, some environment support for manipulating constructs with variable-binding operations is desirable, for example, for program transformations and verification. In most cases, the existing support is found somewhat lacking. If we consider ML, the most popular and widely used programming language which was explicitly designed as a metalanguage, object language variables are normally represented as strings, and it is the programmer’s responsibility to manage correctly the manipulation of variable-binding constructs. In compilers written in ML, this is...
done by renaming apart all of the variables in a program as a preliminary stage in its compilation. As demonstrated by recent work on formalizing program analysis schemes (Field 1992, Field et al. 1995), many analyses call for a more sophisticated treatment of program variables. Recent developments in programming language-type systems, for example, the development of explicitly polymorphic $\lambda$-calculi that are becoming the basis for practical programming language designs (Cardelli 1989, Harper and Lillibridge 1994), also highlight the need for more sophisticated metalanguage support for variable binding. This is also true when we consider programming environment tools for reasoning about program correctness and program transformations (Aiello and Prini 1981, Cardelli 1987). For example the Nuprl theorem-proving and program development environment (Constable et al. 1986) has found “notational definition” to be an essential part of its interface. However the original implementation was ad hoc and flawed, particularly because it did not adequately handle variable binding. As a result of this Griffin has formalized an approach to notational definition that explicitly handles its variable-binding aspects (Griffin 1988).

Finally an attractive aspect of providing some support for variable-binding in the metalanguage is the opportunity it offers for implementing and reusing generic environment tools. Programming environments are increasingly focusing on support for the development of multi-language software systems. Modules in such systems are written to communicate in a “location-independent” manner, relying on compile-time analyses to optimize local communication and synchronization (McNamee and Olsson 1990). Although there are several possibilities for doing this, the preferable approach is to provide a preprocessor for each language in which program modules may be implemented. These preprocessors perform source-to-source transformations on object language programs to optimize their communication, and the results of this preprocessing are then fed to the appropriate object language compilers. Metalanguage support for describing variable binding then offers the opportunity to share generic program manipulation tools in such an environment.

Talcott (1993) offers the related example of the code walker for the PCL implementation of the Common Lisp Object System (Curtis 1990): “this tool is used in the production of many further tools in Lisp environments, including code analyzers, macro definitions, and program transformers” (Talcott 1993). Proper support for variable binding in a program environment can play a similar rôle as the basis for various program manipulation tools in a multilanguage programming environment.

Currently the most popular approach to handling object language variable binding in formal environments is the use of de Bruijn numbers (de Bruijn 1972) (reviewed in Section 3). For example de Bruijn numbers are the basis for the implementation of the Calculus of Constructions in the Coq program development and verification environment. The most important operation to be performed on syntax with variable-binding constructs is substitution (it is the central operation in instantiating generic modules, expanding type definitions, inlining procedure calls, etc.). De Bruijn numbers are the basis for the $\lambda\sigma$-calculus (Abadi et al. 1991, Field 1990), a calculus of explicit substitutions (also reviewed in Section 3). The $\lambda\sigma$-calculus in turn is the basis for efficient implementations of $\tilde{\lambda}$-calculi and, for example, was used in the design of the Quest type-checker (Abadi et al. 1991). Crégut also shows how various abstract machines may be derived using the $\lambda\sigma$-calculus (Crégut 1990),
and Leroy shows how the ZINC abstract machine for the CAML Light interpreter can be derived using this approach (Leroy 1990).

Although the \( \lambda \alpha \)-calculus is quite successful as a tool for developing efficient implementations of \( \lambda \)-calculi, it must be considered too low level for the scenarios and programming applications that we consider. (Nevertheless, as we demonstrate in Section 6, it can provide a useful implementation for a higher-level facility.) An alternative has been proposed, originally by Church (1940), and more recently by Miller and Nadathur (1987) and by Pfenning and Elliott (1988). Higher-order abstract syntax (HOAS) provides a single variable-binding operation (\( \lambda \) abstraction) and a built-in substitution operation (\( \beta \)-reduction). HOAS is provided in the LF logical framework (Harper et al. 1993), the Isabelle generic theorem prover (Paulson 1990), and the \( \lambda \)-Prolog and Elf metaprogramming languages (Nadathur and Miller 1988, Pfenning 1990). HOAS is also the basis for the theory of pattern rewrite systems recently developed by Nipkow (Nipkow 1991, 1993b; Mayr and Nipkow 1997). Wand terms the intermediate languages for continuation-passing style (CPS) compilers “higher-order assembly language” (Wand 1992, Wand and Wang 1994), and HOAS plays a crucial rôle in reasoning about the correctness of transformations in such compilers (for example, the choice of closure representations (Wand 1992)). HOAS is also the basis for Field’s PIM rewriting system for analyzing and optimizing imperative programs (Field 1992, Field et al. 1995), providing a framework for extending program analysis algorithms such as conditional constant propagation, alias analysis, and dependence analysis. In fact PIM’s semantics, with some notion of sharing, can be seen as a formalization of program dependence graph semantics; the use of locally introduced “names” plays a crucial rôle here.

A shortcoming of the HOAS approach is precisely its most attractive feature, the fact that variable binding and substitution is “internalized” in the metalanguage. In Isabelle and \( \lambda \)-Prolog, the metalanguage is the simply typed \( \lambda \)-calculus, with operations on object language encodings (including substitution) provided through higher-order unification (This is also how substitution is provided in the Elf meta-language (Pfenning 1990), although only \( \beta \alpha \)-unification is used). In Nipkow’s higher-order rewrite systems, operations on object language encodings are provided through rewrite rules. All of these approaches rely on a “built-in” implementation of substitution. This aspect of the approach has been criticized by, for example, Basin and Constable (1993):

[T]he logic’s presentation cannot easily be the subject of metatheoretic analysis because analysing variables and operations on them requires reasoning about the framework logic itself.

This gap has begun to be addressed by Nipkow, who has extended much of the theory of first-order rewrite systems to higher-order (Nipkow 1991, 1993b; Mayr and Nipkow 1997). There are still some limitations because substitution is treated as a built-in operation (and therefore cannot be reasoned about without reasoning about the metalanguage and its operational semantics as well). Nipkow’s extension of the first-order critical pairs lemma to higher-order (Nipkow 1991) does not lead to a corresponding generalization of the Knuth-Bendix completion procedure (Knuth and Bendix 1970) (as was incorrectly stated by Nipkow (1991), but acknowledged.
by Mayr and Nipkow (1997), due exactly to the fact that substitution is “built-in.”

There are also more practical objections to providing substitution as a primitive operation with HOAS. If our goal is to design a metaprogramming language that provides HOAS and some facility for manipulating it, it is not clear that providing substitution in the language run-time is appropriate. We claim that substitution is a sufficiently important operation, and how to provide it is sufficiently little understood that it should be provided outside of the language run-time and primitive operations. As a very concrete example of this, Huet (1988) considers the use of defined constants essential in his implementation of the Calculus of Constructions, in order to control the explosion in the size of terms due to β-reductions. A similar consideration holds in programming environments: for example, in type-checking with structural type equivalence (such as the Standard ML module system), we would like to expand type definitions as lazily as possible. Huet also considers the sharing of terms essential in any reasonable implementation of a logical framework. Again a similar lesson was learned during the implementation of the Standard ML module system (MacQueen 1988). It is not clear how these manipulations may be performed by the programmer if substitution is provided as a built-in operation.

What we seek is a facility for providing substitution with HOAS, but also allowing the programmer to reify substitution to the program level and reflect operations on this facility into the language run-time. Such a facility was a stated objective of Pfenning and Lee’s LEAP metalanguage (Pfenning and Lee 1990); however this project did not achieve its objective, for exactly the reason that substitutions could not be controlled (Pierce et al. 1989). More recently Miller has proposed the \( \mathbf{L}_0 \) metalanguage, based on \( \beta_0 \)-unification (Miller 1991a), and has demonstrated how substitution and indeed higher-order unification are implementable as metaprograms in \( \mathbf{L}_0 \) (Miller 1991b). Unfortunately the implementation of substitution in \( \mathbf{L}_0 \) is inherently less efficient than “built-in” implementations. Because the \( \mathbf{L}_0 \) implementation does not allow substitutions to be composed, this gives rise to a considerable amount of repeated copying of terms. Consider, for example, the implementation of a tool (such as a type-checker or on-line partial evaluator) that reduces terms to head normal form and examines the head of the result before proceeding further. Under the \( \mathbf{L}_0 \) approach the term must be reduced completely to normal form, possibly resulting in much wasted computation and potentially building very large intermediate data structures. This criticism is echoed by Michaylov and Pfenning (1992):

Object-level variables [in HOAS] are typically represented by meta-level variables, which means that object-level capture-avoiding substitution can be implemented via meta-level \( \beta \)-reduction. The syntactic restriction to \( \mathbf{L}_0 \) prohibits this implementation technique, and hence a new substitution predicate must be programmed for each object language. Not only does this make programs harder to read and reason about, but a substitution predicate will be less efficient than meta-language substitution.

Thus for example the Elf language, although basing its operational semantics on \( \beta_0 \)-unification, implements substitutions in the run-time, treating equality constraints
that do not satisfy the $\beta_0$-restriction as “hard goals” that are suspended until they are sufficiently simplified by the resolution of other goals (Pfenning 1990, Dowek et al. 1996).

We propose a new approach to handling substitution in metaprogramming languages based on HOAS. Our approach involves a novel combination of HOAS and explicit substitutions, providing a higher-order version of the $\lambda\sigma$-calculus where object language variables are represented as $\lambda$-bound variables in the metalanguage rather than as de Bruijn numbers. As such our approach combines the high-level facility of HOAS with the efficiency and theoretical properties of the $\lambda\sigma$-calculus, as well as a facility for reifying substitutions in HOAS to the program level. A key issue is how to compose higher-order substitutions; to do this, we use an extension of $\beta_0$-unification introduced by Duggan (1997). We refer to the calculus presented here as the $\lambda\sigma\beta_0$-calculus. In Section 6 we consider an efficient implementation for the matching implied by this extended $\beta_0$-unification.

We provide two examples of higher-order $\lambda\sigma$-caluli: the untyped $\lambda$-calculus and a polymorphic-typed $\lambda$-calculus. To reason about the correctness of these representations, we extend the theory of pattern rewrite systems developed by Nipkow, to rewrite rules involving the extended pattern restriction introduced by Duggan (1997). In particular we verify the critical pairs lemma for rewrite rules based on the extended pattern restriction.

In the next section we introduce our adaptation of Nipkow’s framework of pattern rewrite systems (Nipkow 1991, 1993b; Mayr and Nipkow 1997). In Section 3 we review the concrete $\lambda$-calculus (based on de Bruijn numbers) and the $\lambda\sigma$-calculus. In Section 4 we introduce higher-order $\lambda\sigma$-caluli, using a representation of the untyped $\lambda$-calculus as an example. Confluence for this representation is verified using the results of Section 2. In Section 5 we provide as another example a representation of a polymorphic-typed $\lambda$-calculus. In Section 6 we consider an efficient implementation of the matching required for higher-order substitutions, in terms of a translation of rewrite rules into a first-order calculus of renaming substitutions. Finally in Section 7 we consider our conclusions and related work.

2. PATTERN REWRITE SYSTEMS

We use the theory of pattern rewrite systems (PRS) introduced by Nipkow (Nipkow 1991, 1993b; Mayr and Nipkow 1997), with some modifications. The modifications are:

1. We use the $\omega$-order $\lambda$-calculus with products as a metalanguage, rather than the simple $\lambda$-calculus used by Nipkow.

2. Rather than using the pattern restriction introduced by Miller (1991a), we use the extended pattern restriction introduced by Duggan (1997).

3. We enforce the extended pattern restriction for the right-hand sides of rewrite rules as well as the left-hand side. Whereas Nipkow relies on a “built-in” substitution operation, we use the extended pattern restriction to implement substitutions as higher-order rewrite rules.
The metalanguage we use is the \( \omega \)-order \( \lambda \)-calculus with products. The detailed specification of the \( \omega \)-order \( \lambda \)-calculus, without products, is given in Appendix A. This calculus includes abstraction over types at both the term level \((\mathcal{A}t: K \cdot M)\) and the type level \((\lambda t: K \cdot A)\). The latter is necessary because the definition of critical pairs below requires the “lifting” of free variables, including free type variables, over a surrounding context of \( \lambda \)-bound term and type variables.

The addition of products to this calculus is done in a slightly nontraditional manner, following the approach described by Duggan (1997). Rather than adding elimination rules based on left and right projectors for products, Duggan (1997) introduces elimination rules based on *locators*. The latter have the property that they can be composed to provide a simple uniform normal form for terms. Duggan (1997) also identifies a restriction of this calculus for which unification is decidable and for which most general unifiers exist. This restriction strictly generalizes Miller’s pattern restriction for \( \beta \gamma \)-unification, and plays a crucial rôle in what follows.

For our purposes we make use of a somewhat simpler version of the locator calculus and extended pattern restriction introduced by Duggan (1997). This restriction is warranted by the very limited way in which we use products in this paper.

**Definition 2.1 (Locator Calculus).** A locator is defined as

\[
L ::= \text{id} \mid \pi_1 \mid \pi_2 \mid L_1 L_2.
\]

Locators are equal under the equality relation

\[
L_1(L_2 L_3) = (L_1 L_2) L_3
\]

\[
\text{id }L = L
\]

\[
L \text{id }L = L.
\]

The *local calculus* is defined by adding the following syntax rules to the calculus in Appendix A:

\[
A, B ::= \cdots \mid A \times B
\]

\[
M, N ::= \cdots \mid (M, N) \mid LM
\]

with the added type and equality rules

\[
\begin{align*}
\text{\times F} & \quad \Gamma \models A \in \text{Type} \quad \Gamma \models B \in \text{Type} \\
& \quad \Gamma \models A \times B \in \text{Type} \\
\text{\times I} & \quad \Gamma \models M \in A \quad \Gamma \models N \in B \\
& \quad \Gamma \models (M, N) \in A \times B \\
\text{\times E} & \quad \Gamma \models LM \in A_1 \times A_2 \\
& \quad \Gamma \models (\pi_i L) M \in A_i
\end{align*}
\]
Finally define $L_1 \leq L_2$ to mean $L_2 = L_1 L'_1$ for some $L'_1$.

This version of the locator calculus is somewhat simpler than that described by Duggan (1997). This simplicity comes at a loss of generality; however the description is sufficient for the purposes of the current paper. In fact we do not even need all of the locator calculus presented here. We place the following restrictions on the terms we deal with in higher-order rewrite systems. Denote the free and bound (type and term) variables in a term $M$ of the locator calculus by $FV(M)$ and $BV(M)$ respectively. We assume without loss of generality that $FV(A) \cap BV(A) = \{\}$ and $FV(M) \cap BV(M) = \{\}$, for any $A$ and $M$.

**Definition 2.2 (Extended Pattern Restriction).** A term $M$ in $\beta$-normal form satisfies the extended pattern restriction if:

1. Subterms of $M$ of the form $LN$ are restricted to the case where $N$ is $\eta$-reducible to $L_1 \ldots L_n y$, for some $L_1, \ldots, L_n$ and $y \in BV(M)$.

2. For all $t \in FV(M)$, $t$ is in a subexpression that is $\eta$-reducible to the form $t_{t_1} \cdots t_{t_k}$, where $\{t_{t_1}, \ldots, t_{t_k}\} \subseteq BV(M)$ are distinct.

3. For all $x \in FV(M)$, $x$ is in a subterm that is $\eta \delta$-reducible to the form $x \left[ t_{t_1} \ldots t_{t_k} \right] (L_1 y_1) \cdots (L_m y_m)$, where

   a) $\{t_{t_1}, \ldots, t_{t_k}, y_1, \ldots, y_m\} \subseteq BV(M)$,

   b) if $t_{i} = t_{j}$, then $i = j$, and

   c) if $y_{i} = y_{j}$, $i \neq j$, then $L_{i} \leq L_{j}$ and $L_{j} \leq L_{i}$.

A reduction relation for the locator calculus is obtained by orienting the equality rules from left to right, while identifying a term $M$ with the located term $id M$ (as justified by the first $\times \beta$-rule). The following is verified by Duggan (1997):

**Theorem 1.1.** The locator calculus is confluent.

2. Unification in the locator calculus is decidable, for terms satisfying the extended pattern restriction. Any pair of unifiable terms in the locator calculus have a most general unifier.

It will be noted that, with this restriction, pairs do not occur in our metalanguage. All we actually need are product types and located $\delta$-bound variables, and this simplified version of the extended pattern restriction. In terms of Definition 2.1, we can omit the syntax rule $M := (M, N)$ and the $\times I$, $\times \beta$ and second $\times \eta$ type and equality rules.
Any term or type can also be considered as a function defined over \{1, 2\}, with

\[
(\lambda x : A \cdot M)/1 = M \quad (MN)/1 = M, (MN)/2 = N \\
(M, N)/1 = M, (M, N)/2 = N \quad (LM)/1 = M \\
(At : K \cdot M)/1 = M \quad (M[A])/1 = M, (M[A])/2 = A \\
(\lambda t : K \cdot A)/A = A \quad (AB)/1 = A, (AB)/2 = B.
\]

We also extend this homomorphically to functions over finite sequences in \{1, 2\} *

\[
M/\langle \rangle = M \\
M/\langle a_1, a_2, ..., a_n \rangle = (M/a_1)/\langle a_2, ..., a_n \rangle, \quad n \geq 1,
\]

and similarly for type expressions. We denote such paths by \(p, q\), and (associative) path composition by \(p.q\). We also define the operation of replacing a subterm of a tree (where the last two clauses represent the base cases in the recursive definition):

\[
(M[p \mapsto N])/q = \begin{cases} ((M/a)[p' \mapsto N])/q' & \text{if } p = a.p', q = a.q' \\
(M/b)/q' & \text{if } p = a.p', q = b.q', a \neq b \\
N/q & \text{if } p = \varepsilon \\
M[p \mapsto N] & \text{if } q = \varepsilon.
\end{cases}
\]

Let \(BV(M, p)\) denote the set of variables that are bound by \(\lambda\)-binders encountered on the path \(p\) from the root of \(M\).

A substitution \(\theta\) is a mapping from term variables to terms, and from type variables to types, which is the identity on all but a finite number of variables. We overload our notation slightly and denote a substitution by \(\{M_1/x_1, ..., M_n/x_n\}\) in the traditional manner. We denote substitution application and composition by juxtaposition (\(\theta x\) and \(\theta_1\theta_2\) respectively). \(\text{dom}(\theta) = \{v \mid \theta(v) \neq v\}\) and \(\text{cod}(\theta) = \{\theta(v) \mid \theta(v) \neq v\}\). The application of a substitution to a term, \(\theta M\), is defined to be the application of the homomorphic extension of \(\theta\) to \(M\), followed by the reduction of the result to \(\beta\eta\)-normal form; similarly for \(\theta A\). A substitution \(\theta\) is a well-typed substitution from \(\Gamma\) to \(\Gamma'\) provided \(\text{dom}(\theta) \subseteq \text{dom}(\Gamma)\), \(\text{FV}(\text{cod}(\theta)) \subseteq \text{dom}(\Gamma')\), \(\Gamma\triangleright_{\Sigma} x \in A\) implies \(\Gamma'\triangleright_{\Sigma} \theta x \in A\) for \(x \in \text{dom}(\theta)\), and \(\Gamma\triangleright_{\Sigma} t \in K\) implies \(\Gamma'\triangleright_{\Sigma} \theta t \in K\) for \(t \in \text{dom}(\theta)\).

In the following definition, \(\text{tc}\) represents an arbitrary type constructor constant (e.g., \(\text{int}\) or \(\text{list}\)), and \(\text{tc} A_1 \cdots A_k\) denotes a type formed by the application of this type constructor constant to type arguments \(A_1, ..., A_k\) (e.g., \(\text{list int}\)). This notation is introduced in Appendix A.

**Definition 2.3 (Pattern Rewrite Systems (PRS)).** A rewrite rule is a triple \((\Gamma, l, r)\) such that \(l\) and \(r\) are terms (in \(\beta\eta\)-normal form) such \(\Gamma\triangleright_{\Sigma} l \in (\text{tc} A_1 \cdots A_k)\) and \(\Gamma\triangleright_{\Sigma} r \in (\text{tc} A_1 \cdots A_k)\) (where \(\text{dom}(\Gamma) = \text{FV}(l) \supseteq \text{FV}(r)\) and \(\text{tc} \in \text{dom}(\Sigma)\)), \(l\) and \(r\) satisfy the extended pattern restriction, and \(l\) is not \(\eta\)-reducible to a free variable.

A pattern rewrite system (PRS) is a set of rewrite rules. A PRS \(R\) induces a relation \(R^*\) on terms:
\( M/p = \emptyset \) and \( N = M[p \mapsto \emptyset] \), for some \( p \in \text{dom}(M) \), \((\Gamma, l, r) \in R\),
\( \theta \) a well-typed substitution from \( \Gamma \) to \( BV(M, p) \),
\( M \overset{\theta}{\rightarrow} N \iff \Gamma' = \{ (t, K) \mid t \in BV(M, p) \text{ and } t \text{ is bound to kind } K \text{ in } M \}
\cup \{ (x, A) \mid x \in BV(M, p) \text{ and } x \text{ is bound to type } A \text{ in } M \}
\text{ and } \Gamma' \supseteq \chi(M/p) \in \theta(\text{tc } A_1 \cdots A_k) \).

From now on we omit the \( R \) superscript on the rewrite relation, and we assume that all substitutions are well typed. We denote the rewrite rules in a PRS \( R \) by \( l \rightarrow r \), leaving the environment component \( \Gamma \) implicit. \( \rightarrow \) denotes the reflexive closure of the rewrite relation, while \( \rightarrow^* \) denotes the transitive closure. Define \( M_1 \downarrow M_2 \) to mean that \( M_1 \rightarrow^* M \iff M_2 \) for some \( N \). A rewrite relation is \textit{locally confluent} if \( M_1 \rightarrow^* M \rightarrow^* M_2 \) implies that \( M_1 \downarrow M_2 \). A rewrite relation is \textit{confluent} if \( M_1 \downarrow^* M \rightarrow^* M_2 \) implies that \( M_1 \downarrow M_2 \). A rewrite relation is \textit{terminating} (or Noetherian) if there does not exist an infinite sequence \( M_i \rightarrow M_{i+1} \) for \( i \in \mathbb{N} \). Define \( \emptyset \rightarrow \emptyset' \) to mean that \( \emptyset \rightarrow \emptyset' \forall v \forall \text{ dom}(\emptyset) \leq \text{dom}(\emptyset') \).

**Lemma 2.1.** Given a PRS \( R \), if \( M \rightarrow M' \) and \( \emptyset \rightarrow \emptyset' \), then \( \emptyset M \rightarrow \emptyset' M' \).

**Proof.** We only describe how to make the verification of Mayr and Nipkow (1997) go through in our type of system. Define the measure

\[
\text{ord}(M N) = 1
\]

\[
\text{ord}(A \rightarrow B) = \max(\text{ord}(A) + 1, \text{ord}(B))
\]

\[
\text{ord}(A \times B) = \max(\text{ord}(A), \text{ord}(B))
\]

\[
\text{ord}(\forall t : K \cdot A) = \max(\text{ord}(K) + 1, \text{ord}(A))
\]

\[
\text{ord}(K_1 \rightarrow K_2) = \max(\text{ord}(K_1) + 1, \text{ord}(K_2))
\]

\[
\text{ord}(\text{Type}) = 1.
\]

Given \( \emptyset \) a well-typed substitution from \( \Gamma \) to \( \Gamma' \), define \( \text{ord}_{\Gamma \rightarrow}(\emptyset) \) to be

\[
\max\{\text{ord}(\emptyset A) \mid x \in \text{dom}(\emptyset) \text{ and } \Gamma' \supseteq \chi(\emptyset A)\}.
\]

Given type environments \( \Gamma \) and \( \Gamma' \), \( \emptyset \) a well-typed substitution from \( \Gamma \) to \( \Gamma' \). The verification is by induction on \( \text{ord}_{\Gamma \rightarrow}(\emptyset) \), with a nested induction on the length of the reduction sequence for \( M \rightarrow M' \). The challenge is to show that the first induction measure is useful in a polymorphic setting. We show that this induction measure is useful provided function variable applications satisfy the extended pattern restriction.

We consider the case when \( M = N = \alpha t : \bar{K} : \bar{x} : \bar{A} \cdot F[M] \bar{M} \), where \( F \in \text{dom}(\emptyset) \). Then \( \emptyset F = \alpha F : \bar{K} : \bar{x} : \bar{A} : \bar{M} \rightarrow \alpha F : \bar{K} : \bar{x} : \bar{A} : \bar{M} = \emptyset F \) for some \( M' \) and \( M'' \). Define \( \Gamma_0 = (\Gamma, \bar{F} : \bar{K} : \bar{x} : \bar{A}) \), \( \emptyset_0 = \{ \emptyset F / \bar{F}, \emptyset M / \bar{M} \} \), and \( \emptyset_0 = \{ \emptyset F / \bar{F}, \emptyset M / \bar{M} \} \).
Then \( \theta_0 \rightarrow \theta_0 \). Furthermore \( \text{ord}_{F_\mathcal{R}}(\theta_0) < \text{ord}_{F_\mathcal{R}}(\theta) \), since the extended pattern restriction ensures that \( \theta(t) \) is only a located variable for any \( t \in \text{dom}(\theta_0) \). So instantiating type variables with \( \theta_0 \) does not change the order of any variables in \( \text{dom}(\theta_0) \).

Therefore we can apply the induction hypothesis to get \( \theta_0 M' \rightarrow \theta_0 M' \), so we have

\[
\theta M = \alpha\ell : \overline{\mathcal{A}} \cdot \theta \mathcal{A} (\theta F) [\mathcal{A} \theta \mathcal{M}]
\]

\[
= \alpha\ell : \overline{\mathcal{A}} \cdot \theta \mathcal{A} (\theta N)
\]

\[
= \theta N.
\]

We now consider critical pairs in this framework of pattern rewrite systems. \( \rho = \{v'_i/v_i\} \) is a renaming away from \( V \) if each \( v'_i \) is distinct and \( V \cap \{v_1, \ldots, v_k\} = \{\} \).

**Definition 2.4 (Critical Pairs).** Given a PRS with rewrite rules \( l_1 \rightarrow r_1 \) and \( l_2 \rightarrow r_2 \), and a position \( p \in \text{dom}(l_1) \) such that:

1. \( FV(l_1) \cap BV(l_1) = \{\} \).
2. \( l_1/p \) is not an application of a free variable,
3. \( \{t_1, \ldots, t_k, x_1, \ldots, x_h\} = BV(l_1, p) \) and \( \sigma = \{(\rho(t) t_1) / t, (\rho(x) t_1 x_1) / x \mid t, x \in FV(l_2)\} \), where \( \rho \) is a renaming away from \( FV(l_1) \). Then the two patterns \( \alpha\ell_{t_1} : \overline{\mathcal{A}_{x_1}} : \overline{\mathcal{A}_{x_2}} : \overline{\mathcal{A}_{x_3}} : (l_1/p) \) and \( \alpha\ell_{t_2} : \overline{\mathcal{A}_{x_4}} : (\sigma l_2) \) have a most general unifier \( \theta \).

Then the pattern \( l_1 \) overlaps the pattern \( l_2 \) at position \( p \). The rewrite rules determine the critical pair \( (\theta r_1, \theta (l_1[p \mapsto \sigma r_2])) \).

To motivate this definition (taken from Mayr and Nipkow (1997)), consider the following example of overlapping patterns: \( l_1 = c_1(\overline{\mathcal{A}} \cdot \overline{\mathcal{A}} (c_2(F \cdot x \cdot y))) \) and \( l_2 = c_2 G \).

We now extend the critical pair lemma of Knuth and Bendix (1970) to our notion of pattern rewrite systems. We adapt the proof of Mayr and Nipkow (1997) for Nipkow’s PRSs. We first introduce the following concept:

**Definition 2.5 (Locator Renaming).** \( \theta \) is a locator renaming if \( \theta = \{(L, v_i, v'_i)\} \), where \( v_i = v_j \), \( i \neq j \), implies \( L_i \subseteq L_j \) and \( L_i \subseteq L_j \); \( v'_i = v'_j \) implies \( i = j \); and \( \{v_i\} \cap \{v'_i\} = \{\} \). The application of \( \theta \) is defined by \( \theta(L v) = L' v' \), where \( L = L' \) \( L'' \) and \( (L', v, v') \in \theta \).

An example of locator renamings is provided below. Note that the well definedness of a locator renaming is guaranteed by the extended pattern restriction on its domain. Furthermore we have:

**Lemma 2.2.** If \( \theta \) is a locator renaming, then \( \theta^{-1} = \{(v, L v') \mid (L v', v) \in \theta\} \) is a substitution. Furthermore \( \theta \theta^{-1} = \theta^{-1} \theta = \{\} \).
Locator renamings are the critical addition in our adaptation of the proof of the critical pairs lemma of Mayr and Nipkow (1997). To motivate this, consider the following example. We have the PRS with two rules:

\[ l_1 = c_1(\lambda x \cdot F(p_1 x)) \rightarrow c_2(F) = r_1 \]
\[ l_2 = c_3(G) \rightarrow c_4(G) = r_2. \]

Then the term \( M = c_1(\lambda x \cdot c_4(p_1 x)) \) can be rewritten using either of these (non-overlapping) rewrite rules, to:

\[ \bullet \text{ c}_2(\lambda y \cdot \text{c}_3(\pi_2 y)) \text{ under the matching substitution } \theta_1 = \{ \lambda y \cdot \text{c}_3(\pi_2 y)/F \}, \]
\[ \bullet \text{ c}_4(\lambda x \cdot \text{c}_3(\pi_2 x)) \text{ under the matching substitution } \theta_2 = \{ \pi_2 \pi_1 x/G \}. \]

We observe that \( \{ \pi_1 x/y \}(c_3(\pi_2 y)) = \theta_4(F(\pi_2 x)) = \theta_2(c_4 G) \), where \( \{ \pi_1 x/y \} \) is the inverse of the locator renaming \( \{ y/\pi_1 x \} \). So applying the locator renaming to both sides, we have \( c_3(\pi_2 y) = \{ y/\pi_1 x \} \theta_2(c_4 G) \). Define

\[ \theta'_1 = \{ (\lambda y \cdot \{ y/\pi_1 x \} \theta_2(c_4 G))/F \} = \{ (\lambda y \cdot \text{c}_4(\pi_2 y))/F \}. \]

Then we have

\[ \text{c}_2(\lambda y \cdot \text{c}_3(\pi_2 y)) \rightarrow \theta'_1(\text{c}_2 F) \leftarrow \text{c}_4(\lambda x \cdot \text{c}_3(\pi_2 x)). \]

This example illustrates the reasoning in the verification of the following critical pair lemma. The proof is adapted from that of Mayr and Nipkow (1997); we concentrate on those aspects of the verification that are changed by the consideration of extended patterns (using locator renamings).

**Lemma 2.3 (Critical Pair Lemma).** A PRS \( R \) is locally confluent if and only if \( M_1 \downarrow M_2 \) for every critical pair \( (M_1, M_2) \) of \( R \).

**Proof.** We concentrate on the “if” direction. Suppose that

\[ M \rightarrow M_1 \text{ by } M/p_1 = \theta_1 l_1 \text{ and } M_1 = M[p_1 \mapsto \theta_1 r_1] \]
\[ M \rightarrow M_2 \text{ by } M/p_2 = \theta_2 l_2 \text{ and } M_2 = M[p_2 \mapsto \theta_2 r_2] \]

for some \( p_1, p_2 \in \text{dom}(M) \), \( (l_1 \rightarrow r_1), (l_2 \rightarrow r_2) \in R \). We concentrate on the case where \( p_2 = p_1, q \) for some \( q \) so \( M_2/p_2 = (\theta_1 l_1)[q \mapsto \theta_2 r_2] \) and \( (\theta_1 l_1)/q = \theta_2 l_2 \).

**Case 1.** The rules do not overlap, so \( q = q_1, q_2 \) with \( q_1 \in \text{dom}(l_1) \) and \( l_1/q_1 = F[B_{k_1}]/N_{k_2} \). Let \( \theta_1 \) be \( \eta \)-convertible to \( A[k_1] \cdot K_{k_1} \cdot \lambda x_{k_2} : A[k_2] \cdot N \) for some \( N \). Then

\[ (B_{k_1}/\theta_1 k_1, N_{k_1}/\theta_1 k_2) N/q_2 = (\theta_4(F[B_{k_1}]/N_{k_2}))/q_2 = (\theta_1(l_1/q_1))/q_2 = \theta_2 l_2. \]

Since the inverse of \( B_{k_1}/\theta_1 k_1, N_{k_1}/\theta_1 k_2 \) is a locator renaming, we have \( N/q_2 = \{ \theta_1 k_1/B_{k_1}, \theta_1 k_2/N_{k_2} \} \theta_2 l_2 \). Define

\[ \theta'_1 = \theta_1 \{ A[k_1] \cdot K_{k_1} \cdot \lambda x_{k_2} : A[k_2] \cdot N[q_2 \mapsto \{ \theta_1 k_1/B_{k_1}, \theta_1 k_2/N_{k_2} \} \theta_2 l_2]/F \}. \]
Then $\theta_1 \rightsquigarrow \theta_1'$, so by stability $\theta_1 r_1 \rightsquigarrow \theta_1' r_1$.

Furthermore we have

$$\theta_1 l_1[q \mapsto \theta_2 r_2] = (\theta_1 l_1)[q \mapsto (\theta_1(F[B_{k_1}\overline{N_{k_1}}]))[q \mapsto \theta_2 r_2]$$

$$= (\theta_1 l_1)[q \mapsto (\theta_1(F[B_{k_1}\overline{N_{k_1}}]))[q_2 \mapsto \theta_2 r_2]]$$

$$= (\theta_1 l_1)[q \mapsto (\theta_1(F[B_{k_1}\overline{N_{k_1}}]))[q_2 \mapsto \theta_2 r_2] t][q_2 \mapsto \theta_2 r_2]]$$

$$= (\theta_1 l_1)[q \mapsto (\theta_1(F[B_{k_1}\overline{N_{k_1}}]))[t][q_2 \mapsto \theta_2 r_2)]]$$

$$= (\theta_1 l_1)[q \mapsto (\theta_1(F[B_{k_1}\overline{N_{k_1}})] q_2 \mapsto \theta_2 r_2))]$$

$$\rightarrow \theta_1 r_1.$$  

**Case 2.** The rules do overlap, i.e., $q \in \text{dom}(l_1)$ and $l_1/q$ is not the application of a free variable, and $\theta_1(l_1/q) = \theta_2 l_2$. Let $\{t_1, \ldots, t_{k_2}, x_1, \ldots, x_{k_2}\} = \text{FV}(l_1, q)$. Let $\rho$ be a renaming of $\text{FV}(l_2)$ away from $\text{FV}(l_1)$, and let $\sigma = \{t \mapsto \rho(t) : t \in \text{FV}(l_1)\}$.

Therefore there exists a most general unifier $\theta$, and $(\theta r_1 \rightarrow \theta_1 l_1[q \mapsto \sigma] r_2))$ is a critical pair. So by assumption $\theta r_1 \rightarrow \theta_1 l_1[q \mapsto \sigma] r_2))$ for some $N$.

Since $\theta$ is a most general unifier, $\theta_0 = \theta \theta'$ for some $\theta'$. So

$$M_1 = M[p_1 \mapsto \theta r_1] = M[p_1 \mapsto \theta r_1'] = M[p_1 \mapsto \theta' r_1] \rightarrow M[p_1 \mapsto \theta' N],$$

and

$$M_2 = M[p_2 \mapsto \theta r_2]$$

$$= M[p_1 \mapsto (\theta_1 l_1)[q \mapsto \theta_2 r_2]]$$

$$= M[p_1 \mapsto (\theta_1 l_1)[q \mapsto \theta_2 r_2)]$$

$$= M[p_1 \mapsto (\theta_1 l_1)[q \mapsto \theta_2 r_2)]$$

$$= M[p_1 \mapsto (\theta_1 l_1)[q \mapsto \theta_2 r_2)]$$

$$\rightarrow M[p_1 \mapsto \theta' N],$$

where the equality marked by $(\ast)$ follows from the fact that $\theta_2' x = \theta_2 \rho(x) t_{k_2} \overline{N_{k_2}} \theta_2 x$ for $x \in \text{FV}(l_2)$; similarly for $t \in \text{FV}(l_2)$.}

The following result comes from an application of Newman’s Lemma (Newman 1942):

**Corollary 2.1.** A terminating PRS $R$ is confluent if and only if, for all critical pairs $(M, N), \ M \rightarrow N$. 
We conclude this section with a discussion of how our framework relates to that of Nipkow (Nipkow 1991, 1993b). The introduction of polymorphism introduces some interesting extended functionality. Beyond the obvious example of generic rewrite rules (for lists, etc), consider the following example of how the typed λ-calculus may be represented in our framework:

\[
\begin{align*}
\text{term : Type} & \rightarrow \text{Type} \\
\text{abs : } & (\text{Type} \to \text{Type}) \to (\text{Type} \to \text{Type}) \\
\text{app : } & (\text{Type} \to \text{Type}) \to (\text{Type} \to \text{Term}) \\
\text{con : int} & \rightarrow \text{Term}
\end{align*}
\]

with the rewrite rules

\[
\begin{align*}
(\beta) & \quad \text{app}[t_1][t_2](\text{abs}[t_1][t_2] F) S = F(S) \\
(\eta) & \quad \text{abs}[t_1][t_2] (\lambda x : \text{term} t_1 \cdot \text{app}[t_1][t_2] F x) = F.
\end{align*}
\]

Unfortunately our critical pair lemma does not extend to this example, because of the restriction that both sides of a rewrite rule must satisfy the extended pattern restriction. This is in contrast to Nipkow, who only enforces the pattern restriction for the left-hand sides of rewrite rules (Nipkow 1991, 1993b; Mayr and Nipkow 1997). On the other hand, Nipkow’s weakened restriction means that higher-order critical pairs do not lead to a higher-order form of Knuth–Bendix completion:

“One of the main selling points of critical pairs has been the fact that they come with a so-called completion algorithm: a non-confluent rewrite system can be transformed into an equivalent (w.r.t. the equational theory) but confluent system by adding critical pairs as new reduction rules... However, higher-order critical pairs may no longer be pattern rewrite rules in case neither of the two components is a pattern. It is easy to see that this unfortunate state cannot arise if the original PRS we start with contains only rewrite rules where both the left and the right-hand sides are patterns, a rare situation in practice” (Mayr and Nipkow 1997).

As discussed in Subsection 5.4, it is possible to overcome this technical obstacle by programming substitution directly in the rewrite system, in a similar way to how substitutions are programmed in \(L_2\). However this approach has the same problems associated with the implementation of substitution in \(L_2\) (Miller 1991b), due to the inability to compose substitutions. In the next section we show that, using product types and polymorphism, it is possible to provide explicit substitutions as higher-order rewrite rules.

3. REVIEW OF THE \(\lambda\alpha\)-CALCULUS

In this section we review the concrete \(\lambda\)-calculus (based on de Bruijn numbers) and the \(\lambda\alpha\)-calculus. We use the particular formulation of the \(\lambda\alpha\)-calculus described by Abadi et al. (1991).
The untyped $\lambda$-calculus is described by the grammar:

$$E ::= x \mid \lambda x \cdot E \mid E_1 E_2.$$  

"Computation" is modelled by substitution, as represented by the rule for $\beta$-reduction:

$$(\lambda x \cdot E_1) E_2 \rightarrow [E_2/x] E_1.$$  

Since $\beta$-reduction may occur within a $\lambda$-abstraction, substitution must be defined to perform renaming of bound variables to avoid accidental capture of free variables. Thus the case in the definition of (capture-avoiding) substitution for $\lambda$-abstraction is given by

$$\{E_2/y\} \lambda x \cdot E_1 = \begin{cases} 
\lambda x \cdot \{E_2/y\} E_1 & \text{if } x \notin \text{FV}(E_2) \\
\lambda z \cdot \{E_2/y\} \{z/x\} E_1 & \text{otherwise, where } z \notin \text{FV}(E_1) \cup \text{FV}(E_2).
\end{cases}$$  

For example $(\lambda x \cdot ((\lambda y \cdot \lambda z \cdot y) x))$ reduces to $(\lambda x \cdot \lambda z \cdot z)$. Although such renaming is unnecessary in program evaluators, which do not evaluate inside $\lambda$-abstractions, it may become necessary in tasks such as type-checking, partial evaluation, and program transformation.

The de Bruijn $\lambda$-calculus avoids some of the problems associated with renaming by replacing variables by numerical indices, counting the number of levels of nesting of $\lambda$-abstractions between a variable occurrence and the corresponding $\lambda$-binder. The syntax for the de Bruijn $\lambda$-calculus is given by the grammar

$$E ::= n \mid \lambda E \mid E_1 E_2.$$  

For example $(\lambda x \cdot ((\lambda y \cdot \lambda z \cdot y) x))$ is represented in this concrete representation by $\lambda 1((\lambda 2) 1)$. $\beta$-reduction now corresponds to replacing any occurrence of the de Bruijn index 1 with the substitute term. In addition, when reduction occurs inside a $\lambda$-abstraction, the remaining indices must be shifted to reflect the removal of a $\lambda$-binder. The rule for $\beta$-reduction in the concrete $\lambda$-calculus is given by

$$(\lambda E_1) E_2 \rightarrow [E_2/1, 1/2, 2/3, \ldots] E_1.$$  

The complication in the definition of substitution is again the case for $\lambda$-abstractions. The de Bruijn indices in the substitute term must be shifted to reflect the fact that the term is being moved inside of a $\lambda$-binder:

$$\{E_2/1, 1/2, 2/3, \ldots\}(\lambda E_1) = \lambda (\{1/2, (\{2/1, 3/2, \ldots\} E_2)/2, 2/3, \ldots\} E_1).$$  

The $\lambda\sigma$-calculus makes substitutions explicit in the concrete $\lambda$-calculus. In the process the repeated shifting of indices in the concrete calculus is replaced by a shift.
operator that performs this shifting operation lazily. De Bruijn indices are now replaced by repeated applications of the shift operator to the initial index 1. The syntax for the $\lambda\sigma$-calculus is given by

$$
E ::= 1 \mid \lambda E \mid E_1 E_2 \mid E[S] \\
S ::= \text{id} \mid E.S \mid S_1 \circ S_2 \uparrow.
$$

Here id is the identity substitution. $E.S$ is the extension of the substitution $S$ with the binding of $E$ for 1 (defined informally as $\{E/1, S(1)/2, S(2)/3, \ldots\}$). $S_1 \circ S_2$ is the composition of $S_1$ with $S_2$, defined as the mapping of $S_2$ over $S_1$ and the appending of $S_1$ to the result. $E[S]$ is the “closure” of $E$ with the “environment” $S$. The computation rules for the $\lambda\sigma$-calculus include

$$
(\lambda E_1) E_2 \rightarrow E_1[E_2, \text{id}] \\
(E[S_1])[S_2] \rightarrow E[S_1 \circ S_2].
$$

The first rule performs a $\beta$-reduction by forming a closure. The second rule composes two nested closures. The calculus contains other rules for mapping substitutions over terms and over other substitutions. $\uparrow$ is the lazy shift operator; for example moving a substitution inside of a $\lambda$-binder is given by

$$
(\lambda E)[S] \rightarrow \lambda(E[1.(S \circ \uparrow)]),
$$

where the composition of $S$ with $\uparrow$ corresponds to the shifting of the indices in the substitute terms in $S$. De Bruijn numbers are now obtained by shifting 1: the index $n$ is an abbreviation for $1[\uparrow \circ \uparrow \cdots \uparrow]$. So we have

$$
\uparrow \circ (E.S) \rightarrow S \\
1[E.S] \rightarrow E \\
n[E_1 \cdots E_n S] \rightarrow E_n.
$$

4. A HIGHER-ORDER $\lambda\sigma$-CALCULUS

In this section we develop the $\lambda\sigma\beta\delta$-calculus, a higher-order version of the $\lambda\sigma$-calculus presented in the previous section. We use the untyped $\lambda$-calculus as an initial example, to explain how higher-order substitutions are represented. In Section 5 we present a typed version of the $\lambda\sigma\beta$-calculus, which is more representative of the forms of applications this calculus is intended for. The latter example also illustrates the use of local constant introduction, and the separation in the calculus between “free” variables and variables that are bound by substitutions.
The untyped $\lambda$-calculus can be represented as a PRS by introducing the signature

$$
\text{Term} : \text{Type}
$$

$$
\text{abs} : (\text{Term} \to \text{Term}) \to \text{Term}
$$

$$
\text{app} : \text{Term} \to (\text{Term} \to \text{Term}),
$$

and the single higher order rewrite rule

$$
\text{app}(\text{abs} \, M) \, N \quad \longrightarrow \quad M \, N.
$$

A PRS does not impose the pattern restriction on right-hand sides and relies on “built-in” substitution for evaluating the result of the application of a rule. Since we do not wish to rely on built-in substitution, we also impose the (extended) pattern restriction on the right-hand sides of rewrite rules. Following the approach of the $\lambda\sigma$-calculus, we may replace substitution with higher-order closures (Hannan and Miller 1990):

$$
\text{clos} : (\text{Term} \to \text{Term}) \to \text{Term} \to \text{Term}
$$

$$
\text{app}(\text{abs} \, M) \, N \quad \longrightarrow \quad \text{clos} \, M \, N.
$$

The difficulty with this approach is composing substitutions. For example what is the outermost term constructor of $\text{clos}(\lambda x. \text{clos} \, M(N_1, x)) \, N_2$? This question is relevant in a scenario where we only wish to reduce to head normal form before proceeding further (for example, in on-line partial evaluation or type-checking). In the framework of languages such as $\lambda$-Prolog and Elf, examining the term in a closure involves extending the context with local constants to as many levels as there is of nesting of substitutions in the closure. We would like to find an alternative that does not involve resorting to de Bruijn numbers (as is done for example by Hannan and Miller (1990)).

Our approach is inspired to some extent by the categorical explanation of de Bruijn numbers (see for example Hindley and Seldin (1986)). Consider the categorical semantics of the simple typed $\lambda$-calculus. Here types are modeled by objects in a Cartesian closed category. Semantics are defined over type judgements (or equivalently over type derivations), with a type judgement interpreted as a morphism

$$
\left[ x_1 : A_1, \ldots, x_n : A_n \vdash E : B \right] : (\cdots((1 \times A_1) \times A_2) \cdots \times A_n) \to B,
$$

where 1 is “the” initial object. A de Bruijn index $n$ can be interpreted as the iterated projection $\pi_2 \pi_1^n \equiv \pi_3 \pi_1 \cdots \pi_1$. “Shifting” can be seen as specializing a projection further:

$$
n[\uparrow^m] = (\pi_2 \pi_1^n) \circ (\pi_1^m) = \pi_2 \pi_1^{n+m}.
$$

This intuition suggests an approach to composing substitutions in higher-order closures, based on the use of products. In our system substitutions will be trees of
values (rather than lists as $\lambda\sigma$-calculus and ACCL (Abadi et al. 1991, Field 1990)). We introduce a map constructor for mapping a substitution over another substitution and model composition by pairing

$$clos(\lambda x \cdot clos(Mx)(N_1x))N_2 = clos(\lambda x \cdot clos(\lambda y \cdot Mx y)(N_1x))N_2$$
$$\mapsto clos(\lambda z \cdot M(\pi_1z)(\pi_2z))(N_2, map N_1 N_2).$$

This almost works but for the typing. The following extension of the signature for Term fixes the typing

$$\begin{align*}
Subst &: Type \rightarrow Type \\
clos &: AS : (S \rightarrow Term) \rightarrow (Subst S) \rightarrow Term \\
[\_] &: Term \rightarrow Subst Term,
\end{align*}$$

and we have the rewrite rule

$$\text{app}(\text{abs} M) N \mapsto clos[Term] M[N].$$

Thus the (metalanguage) type of a substitution is parameterized by a product type reflecting the structure of the substitution. Note that we are making nontrivial use of both product types and polymorphism in the definition of substitutions. We still need to consider the composition of substitutions. In the $\lambda\sigma$-calculus there is a single composition operation $\circ$ with equations including

$$\begin{align*}
(E[S_1])[S_2] &= E[S_1 \cdot S_2] \\
(E . S_1) \cdot S_2 &= (E[S_2]) . (S_1 \cdot S_2).
\end{align*}$$

These equations characterize $\circ$ as two operations, mapping a substitution over another substitution and appending two substitutions. In our higher-order calculus we have two operations:

$$\begin{align*}
map &: AS_1 \cdot AS_2 : (S_1 \rightarrow Subst S_2) \rightarrow Subst S_1 \rightarrow Subst S_2 \\
\circ &: AS_1 \cdot AS_2 : Subst S_1 \rightarrow Subst S_2 \rightarrow Subst(S_1 \times S_2).
\end{align*}$$

Then composition of higher-order substitutions is given by

$$\begin{align*}
clos[S_1](\lambda x : S_1 \cdot clos[S_2](Mx)(s_1, x)) s_2 \\
= clos[S_1](\lambda x : S_1 \cdot clos[S_2](\lambda y : S_2 \cdot Mx y)(s_1, x)) s_2 \\
\mapsto clos[S_1 \times S_2](\lambda z : S_1 \times S_2 \cdot M(\pi_1z)(\pi_2z))(s_2 \cdot (map[S_1][S_2] s_1, s_2)),
\end{align*}$$

where $s_1$ has type $S_1 \rightarrow Subst S_2$ and $s_2$ has type $Subst S_1$. "Looking up" in such a closure is based on matching on the possible projections in a closure

$$\begin{align*}
clos[S_1 \times S_2](\lambda x : S_1 \times S_2 \cdot M(\pi_1x))(s_1, s_2) \mapsto clos[S_s] M s_1,
\end{align*}$$
where \( s_1 \) has type \( \text{Subst } S_1 \) and \( s_2 \) has type \( \text{Subst } S_2 \). We also need a rule for composing the mapping of substitutions

\[
\begin{align*}
\text{map}[S][S_1](\lambda x : S \cdot \text{map}[S_2][S_1](s_1 x)(s_2 x)) s &= \text{map}[S][S_1](\lambda y : S \cdot \text{map}[S_2][S_1](s_1 x y)(s_2 x)) s \\
&\quad\rightarrow \text{map}[S \times S_2][S_1](\lambda z : S \times S_2 \cdot s_1(z_1)(z_2))((s \circ (\text{map}[S_2][S_1] s_2)) s),
\end{align*}
\]

where \( s \) has type \( \text{Subst } S \), \( s_1 \) has type \( S \rightarrow S \rightarrow \text{Subst } S_1 \), and \( s_2 \) has type \( S \rightarrow \text{Subst } S_2 \). Figure 1 gives the complete set of rules for defining substitutions for the representation of the untyped \( \lambda \)-calculus. These rules define a PRS as defined in Section 2, with the restriction that right-hand sides are also patterns.

We now follow a line of reasoning similar to that for the \( \lambda \sigma \)-calculus (Abadi et al. 1991) to verify the confluence of this system. To this purpose we separate the PRS into two subsystems: \( \lambda_n^{\beta} \) (constituting of only the \( \beta \) rule) and \( \lambda_n^U \) (constituting of the remaining rules). \( \lambda_n^{\beta U} \) denotes the union of these rewrite systems. We denote (untyped) reduction under \( \lambda_n^{\beta U} \), \( \lambda_n^{\beta U} \), and \( \lambda_n^U \) by \( \rightarrow_{\beta n} \), \( \rightarrow_{\beta n} \), and \( \rightarrow_{\beta n} \), respectively. Untyped \( \beta \)-reduction is denoted by \( \rightarrow_{\beta} \) and given by the rule

\[
\text{app}(\text{abs } M) N \rightarrow_{\beta} M N.
\]

**Figure 1.** PRS for Untyped \( \lambda \)-calculus
Lemma 4.1 (Termination of $\lambda^U$). The PRS $\lambda^U$ is Noetherian, i.e., terminating.

Proof. We adapt Field’s termination proof for Accl (Field 1991). To reason about termination we will use a lexicographic semantic path ordering. Since both the right-hand sides as well as the left-hand sides of the rewrite rules are required to be patterns, it is sufficient to consider techniques developed for first-order rewrite systems. In particular we use a lexicographic semantic path ordering similar to that considered by Field.

We define a translation of terms in the $\lambda U$-calculus into a first-order calculus without $\mu$-abstraction. This calculus has constructors $\text{abs}$, $\text{apply}$, $\text{clos}$, $\text{map}$, and $\cdot$, of the appropriate arity, as well as a constant $\text{var}$ into which all $\mu$-bound variables are translated. The translation is defined as follows on normal-form terms:

\[
\begin{align*}
(\text{abs } M)^* &= \text{abs}(M^*) \\
(\text{apply } M \, N)^* &= \text{apply}(M^*, N^*) \\
(\text{clos} [S] \, M \, s)^* &= \text{clos}(M^*, s^*) \\
(\text{map} [S_1] [S_2] \, s_1 \, s_2)^* &= \text{map}(s_1^*, s_2^*) \\
([M])^* &= [M]^* \\
(s_1 \cdot s_2)^* &= s_1^* \cdot s_2^* \\
(\lambda x : A \cdot M)^* &= (\{ \text{var}/x \} \, M)^* \\
(M_1 \ldots M_n)^* &= x \\
((L \, \text{var}) \, M_1 \ldots M_n)^* &= \text{var}.
\end{align*}
\]

We define the following precedence on first-order constructors:

\[
clos \gg map \gg \text{app} \gg \text{abs} \gg \cdot \gg [\ldots] \gg \text{var}.
\]

The following measure gives a rough estimate of the eventual size of a term or substitution after normalization of substitutions:

\[
\begin{align*}
|x| &\overset{\text{def}}{=} 1 \\
|\text{var}| &\overset{\text{def}}{=} 1 \\
|\text{app}(M, N)| &\overset{\text{def}}{=} |M| + |N| + 1 \\
|\text{abs}(M)| &\overset{\text{def}}{=} |M| + 1 \\
|\text{clos}(M, s)| &\overset{\text{def}}{=} |M| \cdot |s| \\
|[M]| &\overset{\text{def}}{=} |M| \\
|s_1 \cdot s_2| &\overset{\text{def}}{=} \text{max}(|s_1|, |s_2|) \\
|\text{map}(s_1, s_2)| &\overset{\text{def}}{=} |s_1| \cdot |s_2|.
\end{align*}
\]
For first-order terms $M = t_1(M_1, ..., M_m)$ and $N = t_2(N_1, ..., N_n)$, define the precedence ordering $M \succeq N$ by the lexicographic combination of $\succ$ and eventual size under $\sigma$-normalization:

$$M \succeq N \iff (t_1 \succ t_2 \lor (t_1 \approx t_2 \land |M| > |N|)).$$

Finally $\succ$ is extended to a simplification ordering $\succ_*$:

$$M = t_1(M_1, ..., M_m) \succ_* t_2(N_1, ..., N_n) = N$$

if

1. $M_i \succ_* N$ for some $i = 1, ..., m$, or
2. $M \succ_* N$ and $M_i \succ_{N_j}$ for $j = 1, ..., n$, or
3. $M \succeq N$, $(M_1, ..., M_m) \succeq_* (N_1, ..., N_n)$ and $M > N_j$ for all $j = 1, ..., n$,

where $\succeq_*$ is the lexicographic extension of $\succeq$ to sequences. To verify that $\succ_*$ is a simplification ordering, it suffices to check that the 11 axioms given by Kamin and Levy (1980) (see also Huet (1986)) are satisfied. As noted by Field, the crucial case is the last axiom, stating that if $M \rightarrow N$ then $t(..., M, ...) \succeq_* t(..., N, ...)$. The verification follows from the fact that $|M| \geq |N|$ where $M$ is the left-hand side and $N$ the right-hand side, respectively, of any rule in the PRS (for example, for ClosSubst, $|M| \cdot |s_1| \cdot |s_2| = |M| \cdot \max(|s_1|, |s_2|)$), and the fact that size is defined monotonically on the structure of terms.

Finally it is straightforward to verify that $M \succ N$ for each rule $M \rightarrow N$ in the PRS. For ClosConst and ClosVar this follows from the first part of the definition. For ClosL and ClosR, this follows using the third and then the first part of the definition. For ClosApp, ClosAbs, MapTerm, and MapComp, the result follows using the first and then the second parts of the definition of the ordering (using clos $\succ_*$ abs, clos $\succ_*$ apply, clos $\approx_*$ map $\succ_*$ [...], and clos $\approx_*$ map $\succ_*$ (..., ...), respectively). Finally the verification for ClosSubst and MapSubst proceeds using the third and first parts of the ordering. (Note that the use of a lexicographic ordering of the term arguments is needed here.)

A term $M$ is canonical if it is constructed from an application of one of the constructors abs, apply, or clos, or a located variable $Lx$ where $x \in \text{dom}(Γ)$, and every subexpression of $M$ is canonical. A substitution $s$ is canonical if it is constructed from an application of one of the constructors [...], map or _ _ _ and every subexpression of $s$ is canonical.

**Lemma 4.2.** Given a type environment $Γ$ where all types have the form

$$S ::= \text{Term} \mid S_1 \times S_2$$

1. If $Γ \triangleright_\Sigma M \in \text{Term}$ where $M$ is in $\beta\eta$-normal form, then $M$ is canonical.
2. If $Γ \triangleright_\Sigma s \in \text{Subst}(A)$ where $s$ is in $\beta\eta$-normal form, then $A$ has the form described by $S$, and $s$ is canonical.
Proof. This is proved by induction on a derivation for \( \Gamma \vdash x : M \in \text{Term}(\Gamma \vdash x : s \in \text{Subst}(A)) \). If \( \Gamma \vdash x : s \in \text{Subst}(A) \), then \( s \) must be constructed from an application of one of the constructors (by the restriction on \( \Gamma \)). If \( s \) has the form \( \text{map}[B][A](\lambda x : B \cdot s_d) s_B \), then applying the induction hypothesis we have \( s_B \) is canonical and \( B \) has the form described by \( S \). Then \( \Gamma, x : B \) has the form required for the induction hypothesis, so we have \( s_d \) is canonical and \( A \) has the form described by \( S \).}

**Lemma 4.3** (Confluence of \( \lambda^e \_ \)). \( \lambda^e \_ \) is confluent on closed terms, i.e., if \( M \xrightarrow{\eta} N_1 \) and \( M \xrightarrow{\eta} N_2 \), then there exists an \( N \) such that \( N_1 \xrightarrow{\eta} N \) and \( N_2 \xrightarrow{\eta} N \).

**Proof.** We verify local confluence by an examination of higher-order critical pairs. Confluence then follows from termination, by an application of Corollary 2.1. For example for the critical pair formed by \text{ClosConst} and \text{ClosSubst}, we have that

\[
\text{clos}[S_1](\lambda x : S_1 \cdot \text{clos}[S_2] M s_2) s_1 \\
\xrightarrow{\eta} \text{clos}[S_1 \times S_2](\lambda z : S_1 \times S_2 : M(\pi_2z))(s_1 \circ (\text{map}[S_1][S_2] s_2 s_1)) \\
\text{by ClosSubst} \\
\xrightarrow{\eta} \text{clos}[S_2](\lambda x : S_2 \cdot M) s_2 \quad \text{by ClosR},
\]

where \( \xrightarrow{\eta} \) denotes equality using the rules for explicit substitutions. The difficult case is for the critical pair formed by \text{ClosL} and \text{ClosSubst} in

\[
\text{clos}[S_1](\lambda x : S_1 \cdot \text{clos}[S_2](M(\pi_1x))(s_\lambda(\pi_1x)))(s_\mu s'_\mu).
\]

(The case for \text{ClosR} and \text{ClosSubst} is similar.) For brevity we will ignore type annotations in the following argument. Define

\[
\mathcal{M}_n \overset{\text{def}}{=} \text{clos}(\lambda z : M(\pi_1^{n+1}z)(\pi_2\pi_1^1z)_{\text{\(\iota\}_{1(n_1)}}) \mathcal{G}_n \\
\mathcal{M}_n' \overset{\text{def}}{=} \text{clos}(\lambda z : M(\pi_1^{n+1}z)(\pi_2\pi_1^1z)_{\text{\(\iota\}_{1(n_1)}}) \mathcal{G}_n' \\
\mathcal{G}_0 \overset{\text{def}}{=} s_\mu \circ (\text{map} s_\mu s_\mu) \\
\mathcal{G}_n \overset{\text{def}}{=} \mathcal{G}_{n-1} \circ (\text{map}(\lambda x : s_d(\pi_1^{n+1}z)(\pi_2\pi_1^1z)_{\text{\(\iota\}_{1(n_1)}}) \mathcal{G}_{n-1}) \\
\mathcal{G}_0' \overset{\text{def}}{=} (s_\mu \circ s'_\mu) \circ (\text{map}(\lambda x : s_d(\pi_1^1x))(s_\mu \circ s'_\mu)) \\
\mathcal{G}_n' \overset{\text{def}}{=} \mathcal{G}_{n-1}' \circ (\text{map}(\lambda x : s_d(\pi_1^{n+1}z)(\pi_2\pi_1^1z)_{\text{\(\iota\}_{1(n_1)}}) \mathcal{G}_{n-1}', \\
\]

where \( M(N_n)_{\text{\(\iota\}_{1(n_1)}} \) denotes the application \( M N_n \ldots N_0 \). Then our claim is that

1. \( \langle \text{map}(s_\mu s_\mu) \downarrow (\text{map}(\lambda x : s_d(\pi_1^1x))(s_\mu \circ s'_\mu)) \rangle \)
2. \( \langle \text{map}(\lambda x : s_d(\pi_1^{n+1}x)(\pi_2\pi_1^1x)_{\text{\(\iota\}_{1(n_1)}}) \mathcal{G}_n) \downarrow \text{map}(\lambda x : s_d(\pi_1^{n+1}x)_{\text{\(\iota\}_{1(n_1)}}) \mathcal{G}_n \rangle \)
3. \( \mathcal{M}_n \downarrow \mathcal{M}_n' \)
using the rewrite rules for substitutions. We prove these simultaneously by induction on (length of reduction sequence, size of term). Consider (1) first of all:

**Case 1.** \( s_y = \lambda y \cdot [M \cdot y] \). Then

\[
\text{map } s_y \cdot s_x \rightarrow [\text{clos } M \cdot s_x]
\]

\[
\downarrow [\text{clos} (\lambda y \cdot M(\pi_1 \cdot y))(s_x \cdot s'_x)] \text{ by (3)}
\]

\[
\leftrightarrow \text{map}(\lambda y \cdot s_x(\pi_1 \cdot y))(s_x \cdot s'_x).
\]

**Case 2.** \( s_y = \lambda y \cdot (s \cdot y) \cdot (s' \cdot y) \). Then

\[
\text{map } s_y \cdot s_x \rightarrow (\text{map } s \cdot s_x) \cdot (\text{map } s' \cdot s_x)
\]

\[
\downarrow (\text{map}(\lambda y \cdot s(\pi_1 \cdot y))(s_x \cdot s'_x)) \cdot (\text{map}(\lambda y \cdot s'(\pi_1 \cdot y))(s_x \cdot s'_x))
\]

\[
\leftrightarrow \text{map}(\lambda y \cdot (s(\pi_1 \cdot y)))(s_x \cdot s'_x).
\]

**Case 3.** \( s_y = \lambda y \cdot \text{map}(\lambda z \cdot s \cdot y \cdot z)(s' \cdot y) \). Then

\[
\text{map } s_y \cdot s_x \rightarrow \text{map}(\lambda z \cdot s(\pi_1 \cdot z)(\pi_2 \cdot z))(s_x \cdot (\text{map } s' \cdot s_x))
\]

\[
\downarrow \text{map}(\lambda z \cdot s(\pi_1 \cdot z)(\pi_2 \cdot z))(s_x \cdot (\text{map } s' \cdot s_x)) \text{ by (2)}
\]

\[
\leftrightarrow \text{map}(\lambda y \cdot \text{map}(\lambda z \cdot s(\pi_1 \cdot z)(\pi_2 \cdot z)))(s_x \cdot s'_x).
\]

The proof for (2) is essentially similar, we only prove the case where \( s_n = \lambda \pi_{n+2} \cdot [M(\pi_{n+2})] \):

\[
\text{map}(\lambda x_n \cdot [M(\pi_{n+2} \cdot x_n)(\pi_2 \cdot x_n)]_{\pi_i \in \{n+1\}}) \mathscr{L}_{n}
\]

\[
\rightarrow [\text{clos}(\lambda x_n \cdot M(\pi_{n+2} \cdot x_n)(\pi_2 \cdot x_n))_{\pi_i \in \{n+1\}}] \mathscr{L}_{n}
\]

\[
= [\mathscr{L}_{n-1}]
\]

\[
\downarrow [\mathscr{L}_{n-1}] \text{ by (3) and induction}
\]

\[
\leftrightarrow \text{map}(\lambda x_n \cdot M(\pi_{n+2} \cdot x_n)(\pi_2 \cdot x_n))_{\pi_i \in \{n+1\}} \mathscr{L}_{n-1}.
\]

We finally verify (3) by induction on the structure of the term \( M \):

**Case 1.** \( M = \lambda \pi_{n+2} \cdot u_i \) for some \( i \in \{2, ..., n+2\} \). Then

\[
\mathscr{L}_{n} = \text{clos}(\lambda z \cdot \pi_{n+2} \cdot x_{n+2}) \mathscr{L}_{n}
\]

\[
\rightarrow \text{clos}(\lambda z \cdot \pi_2 \cdot x_{n+2}) \mathscr{L}_{n-2}.
\]
Subcase a. \( i = 2 \). Then

\[
clos(\lambda z \cdot \pi_2 z) \mathcal{S}_0 = clos(\lambda z \cdot \pi_2 z)((s_s \circ (\text{map } s_y s_s))
\]
\[
\rightarrow clos(\lambda z \cdot z)(map s_y s_s)
\]
\[
\downarrow clos(\lambda z \cdot z)(map(\lambda y \cdot x_j(\pi_1 y))(s_s \circ s'_s)) \quad \text{by (1)}
\]
\[
\leftarrow clos(\lambda z \cdot \pi_2 z)((s_s \circ s'_s) \circ (\text{map } \lambda y \cdot x_j(\pi_1 y))(s_s \circ s'_s))
\]
\[
= clos(\lambda z \cdot \pi_2 z) \mathcal{S}_0.
\]

Subcase b. \( 2 < i \leq n + 2 \). Let \( j = i - 2 \), then

\[
clos(\lambda z \cdot \pi_2 z) \mathcal{S}_j
\]
\[
= clos(\lambda z \cdot \pi_2 z)((\mathcal{S}_{j-1} \circ (\text{map }\lambda x_j \cdot s_j(\pi_1 + 1 x_j)(\pi_2 \pi_1^b x_j s_{(j+1)}) \mathcal{S}_{j-1}))
\]
\[
\rightarrow clos(\lambda z \cdot z)(\text{map }\lambda x_j \cdot s_j(\pi_1 + 1 x_j)(\pi_2 \pi_1^b x_j s_{(j+1)})) \mathcal{S}_{j-1}
\]
\[
\downarrow clos(\lambda z \cdot z)(\text{map }\lambda x_j \cdot s_j(\pi_1 + 2 x_j)(\pi_2 \pi_1^b x_j s_{(j+1)})) \mathcal{S}_{j-1} \quad \text{by (2)}
\]
\[
\leftarrow clos(\lambda z \cdot \pi_2 z)((\mathcal{S}_{j-1} \circ (\text{map }\lambda x_j \cdot s_j(\pi_1 + 2 x_j)(\pi_2 \pi_1^b x_j s_{(j+1)})) \mathcal{S}_{j-1}))
\]

In either case we have

\[
clos(\lambda z \cdot \pi_2 z) \mathcal{S}_{i-2} \downarrow clos(\lambda z \cdot \pi_2 z) \mathcal{S}_{i-2}
\]
\[
\rightarrow clos(\lambda z \cdot \pi_2 \pi_1^{n-i} z) \mathcal{S}_n
\]
\[
= \mathcal{M}_n.
\]

Case 2. \( M = \lambda u_{n+2} \cdot u_1 \). Then

\[
\mathcal{M}_n = clos(\lambda z \cdot \pi_1 + 1 z) \mathcal{S}_n
\]
\[
\rightarrow clos(\lambda z \cdot \pi_1 z)(s_s \circ (\text{map } s_y s_s))
\]
\[
\rightarrow clos(\lambda z \cdot z) s_s
\]
\[
\leftarrow clos(\lambda z \cdot \pi_1 z)(s_s \circ s'_s)
\]
\[
\leftarrow clos(\lambda z \cdot \pi_1 \pi_2 z)((s_s \circ s'_s) \circ (\text{map } s_y s_s))
\]
\[
\rightarrow clos(\lambda z \cdot \pi_1^{n+2} z) \mathcal{S}_n^r.
\]

Case 3. \( M = \lambda u_{n+2} \cdot v \), where \( v \notin \{ u_1 \} \). Then \( \mathcal{M}_n = \mathcal{M}_n = \mathcal{M}_n^r. \)
Case 4. \( M = \lambda u_{n+2} \cdot \text{app}(M_1(u_{n+2}))(M_2(u_{n+2})) \). Then

\[
\mathcal{M}_n = \text{clos}(\lambda z \cdot \text{app}(M_1 \pi_1^{n+1} z(\pi_2 \pi_1^1 z)_{e(n_1)}) \| M_2 \pi_1^{n+1} z(\pi_2 \pi_1^1 z)_{e(n_1)}) \mathcal{S}_n
\]

\[
\Rightarrow \text{app}(\text{clos}(\lambda z : M_1 \pi_1^{n+1} z(\pi_2 \pi_1^1 z)_{e(n_1)}) \mathcal{S}_n)
\]

\[
\times (\text{clos}(\lambda z : M_2 \pi_1^{n+1} z(\pi_2 \pi_1^1 z)_{e(n_1)}) \mathcal{S}_n)
\]

\[
\downarrow \text{app}(\text{clos}(\lambda z : M_1 \pi_1^{n+2} z(\pi_2 \pi_1^1 z)_{e(n_1)}) \mathcal{S}_n)
\]

\[
\times (\text{clos}(\lambda z : M_1 \pi_1^{n+2} z(\pi_2 \pi_1^1 z)_{e(n_1)}) \mathcal{S}_n)
\]

by induction hypothesis

\[
\Leftarrow \text{clos}(\lambda z : \text{app}(M_1 \pi_1^{n+1} z(\pi_2 \pi_1^1 z)_{e(n_1)}) \| M_2 \pi_1^{n+1} z(\pi_2 \pi_1^1 z)_{e(n_1)}) \mathcal{S}_n.
\]

Case 5. \( M = \lambda u_{n+2} \cdot \text{abs}(M(u_{n+2})) \). This is similar to Case 4.

Case 6. \( M = \lambda u_{n+2} \cdot \text{clos}(\lambda w \cdot N \ u_{n+2} \ w) (s_{n+1} u_{n+2}) \). Then

\[
\mathcal{M}_n = \text{clos}(\lambda w \cdot N \pi_1^{n+1} z(\pi_2 \pi_1^1 z)_{e(n_1)} w) (s_{n+1} \pi_1^{n+1} z(\pi_2 \pi_1^1 z)_{e(n_1)}) \mathcal{S}_n
\]

\[
\Rightarrow \text{clos}(\lambda w \cdot N \pi_1^{n+2} z(\pi_2 \pi_1^1 z)_{e(n_1)}) \mathcal{S}_{n+1}
\]

\[
\downarrow \text{clos}(\lambda w \cdot N \pi_1^{n+3} z(\pi_2 \pi_1^1 z)_{e(n+1_1)}) \mathcal{S}_{n+1}
\]

by induction

\[
\Leftarrow \mathcal{M}_n.
\]

We now let \( \sigma(M) (\sigma(s)) \) denote the (unique) normal form for the term \( M \) (substitution \( s \)) under the \( \lambda_n^U \) PRS. The remainder of the proof of confluence for \( \lambda_n^U \) follows very closely that for the \( \lambda \sigma \)-calculus, in particular using Hardin's interpretation technique and confluence for the untyped \( \lambda \)-calculus.

We verify that the PRS \( \lambda_n^U \) is a correct implementation of substitution. The following rules are for the judgement form \( \{N/x\} M \Rightarrow M' \), where \( M, N \) and \( M' \) are restricted to not having closure terms:

\begin{align*}
\text{VAR}_1 & \quad \{N/x\} \ y \Rightarrow y \quad x \neq y \\
\text{VAR}_2 & \quad \{N/x\} \ x \Rightarrow N \\
\text{APP} & \quad \{N/x\} \ M_1 \Rightarrow M_1' \quad \{N/x\} \ M_2 \Rightarrow M_2' \\
& \quad \{N/x\} \ (\text{app } M_1 M_2) \Rightarrow (\text{app } M_1' M_2') \\
\text{ABS} & \quad y \notin \text{FV}(M) \cup \text{FV}(N) \cup \{x\} \quad \{N/x\} (M y) \Rightarrow (M' y) \\
& \quad \{N/x\} \ (\text{abs } M) \Rightarrow (\text{abs } M').
\end{align*}
We can then verify the following lemma by induction on the structure of a term $M$
(or equivalently by induction on a derivation in the inference system just defined):

**Lemma 4.4.** If $M$, $N$, and $M'$ are terms without closures as subterms, and
\[ \{N/x\} M \Rightarrow M', \]
then
\[ \sigma(\text{clos}(\lambda x \cdot M)[[N]]) = M'. \]

**Proof.** For the case where $M = \text{abs } M_1$ (we omit the other easy cases), we have
that \[ \{N/x\} (M_1 y) \Rightarrow (M'_1 y), \]
so by the induction hypothesis \[ \sigma(\text{clos}(\lambda y \cdot M_1 y)[[N]]) = (M'_1 y), \]
so
\[ \sigma(\text{clos}(\lambda y \cdot \text{abs } M_1 y))[[N]] = \text{abs}(\lambda z \cdot \sigma(\text{clos}(\lambda y \cdot M_1 y z)[[N]])) \]
\[ = \text{abs}(\lambda z \cdot \text{clos}(\lambda y \cdot M'_1 y z)[[N]])) \]
\[ = \text{abs}(\lambda z \cdot M' z) \]
\[ = \text{abs } M'. \]

**Corollary 4.1.** If $M \beta \Rightarrow N$ then $M \beta \Rightarrow M$. \hfill \square

**Proof.** By the definition of $\beta$-reduction and the previous lemma, it suffices to
perform a $\beta$-reduction and then normalize with respect to $\lambda^U$, i.e., if $M \Rightarrow N$ then
\[ \exists M'. M \Rightarrow M' \text{ and } M' \Rightarrow N. \]

**Corollary 4.2.** $\beta$-reduction is confluent on $\lambda^U$ normal forms.

**Proof.** For terms this follows from confluence for the original term calculus
(without substitutions). For substitutions, $\beta$-reduction amounts to $\beta$-reducing
the components of substitutions. \hfill \square

**Lemma 4.5.** 1. For closed term $M$ and $N$, if $M \Rightarrow \beta N$ then $\sigma(M) \Rightarrow \beta \sigma(N)$.  
2. For closed substitutions $s$ and $t$, if $s \Rightarrow \beta t$ then $\sigma(s) \Rightarrow \beta \sigma(t)$. 

**Proof.** We verify this by induction on length of reduction sequence, size of term,
or substitution.

1. $M = \text{app } M_1 M_2$, where the redex is in either $M_1$ or $M_2$, then the result
follows by an application of the induction hypothesis.

2. $M = \text{abs } M'$, where the redex is in $M'$, then the result again follows by
an application of induction.

3. $M = \text{app}(\text{abs } M_1) M_2$ and $N = \text{clos } M_1 [[M_2]]$, then
\[
\sigma(M) = \text{app}(\text{abs } (\sigma(M_1)))(\sigma(M_2)) \\
\Rightarrow \beta \sigma(M_1)(\sigma(M_2)) \\
= \sigma(\text{clos}(\sigma(M_1))[[\sigma(M_2)]]) \text{ by Lemma 4.4} \\
= \sigma(N).
\]
4. $M = \text{clos } M'$, then there are several subcases to consider:

(a) $M' = \lambda x \cdot \text{app}(M'_1 x)(M'_2 x)$: If the redex is in $M'_1$ or $M'_2$ or $s$, then apply the induction hypothesis to $\text{clos } M'_1 s$ and $\text{clos } M'_2 s$. If the redex is $M' = \lambda x \cdot \text{app}(M'_1 x)(M'_2 x) = \lambda x \cdot \text{app}(\text{abs}(M'_1 x))(M'_2 x)$, then

$$\sigma(M) = \sigma(\text{clos}(\lambda x \cdot \text{app}(\text{abs}(M'_1 x))(M'_2 x)) s)$$
$$= \text{app}(\sigma(\text{clos}(\lambda x \cdot \text{abs}(M'_1 x)) s))(\sigma(\text{clos}(\lambda x \cdot M'_2 x) s))$$
$$= \text{app}(\lambda y \cdot (\sigma(\text{clos}(\lambda x \cdot M'_1 x) s))(\sigma(\text{clos}(\lambda x \cdot M'_2 x) s)))$$
$$\rightarrow [\sigma(\text{clos}(\lambda x \cdot \text{abs}(M'_1 x) y) s) \rightarrow \sigma(\text{clos}(\lambda x \cdot (\lambda x \cdot M'_2 x) y) s)]$$
$$= \sigma(\text{clos}(\lambda y \cdot (\sigma(\text{clos}(\lambda x \cdot M'_1 x) y) s))) \rightarrow \sigma(\text{clos}(\lambda y \cdot M'_2 x) s))$$
$$\rightarrow [\sigma(\text{clos}(\lambda y \cdot \text{abs}(M'_1 x) y) s) \rightarrow \sigma(\text{clos}(\lambda y \cdot (\lambda y \cdot M'_2 x) y) s)$$
$$= \sigma(\text{clos}(\lambda y \cdot \text{abs}(M'_1 x) y) s) \rightarrow \sigma(\text{clos}(\lambda y \cdot (\lambda y \cdot M'_2 x) y) s)$$
$$= \sigma(\text{clos}(\lambda y \cdot \text{abs}(M'_1 x) y) s) \rightarrow \sigma(\text{clos}(\lambda y \cdot (\lambda y \cdot M'_2 x) y) s)$$
$$= \sigma(N).$$

Claim 1 extends the projection rules for terms to substitutions. Define

$$\mathcal{S}_n \overset{\text{def}}{=} \mathcal{S}_{n-1} \circ (\text{map}(\lambda x \cdot \text{abs}(\pi_2 \pi'_1 x)_{x \in (n-1)})) \mathcal{S}_{n-1})$$
$$\mathcal{S}_0 \overset{\text{def}}{=} s_x \circ (\text{map}(\lambda y \cdot s_x)) s_x$$

$$\mathcal{S}'_n \overset{\text{def}}{=} \mathcal{S}_{n-1} \circ (\text{map}(\lambda x \cdot s_n \pi'_1 x)_{x \in (n-1)})) \mathcal{S}_{n-1}) \quad \text{for } n > 1$$
$$\mathcal{S}'_0 \overset{\text{def}}{=} (\text{map}(\lambda x \cdot s_1 x)) \mathcal{S}'_0$$
$$\mathcal{S}'_0 \overset{\text{def}}{=} s_x.$$  

Then the claim is that

(i) $\sigma(\text{map}(\lambda y \cdot s_x)) = \sigma(s_x)$
(ii) $\sigma(\text{map}(\lambda x \cdot s_1 x) \mathcal{S}_0) = \sigma(\text{map } s_1; \mathcal{S}'_0)$
(iii) $\sigma(\text{map}(\lambda x \cdot s_n \pi'_1 x)_{x \in (n-1)})) \mathcal{S}_{n-1}) = \sigma(\text{map}(\lambda x \cdot s_n \pi'_1 x)_{x \in (n-1)})) \mathcal{S}_{n-1})$ for $n > 1$
(iv) $\sigma(\text{clos}(\lambda x \cdot M \pi'_1 x)_{x \in (n-1)})) \mathcal{S}_{n-1}) = \sigma(\text{clos}(\lambda x \cdot M \pi'_1 x)_{x \in (n-1)})) \mathcal{S}_{n-1})$. 


We verify the claim by induction of length of reduction sequence, or size of term. We only consider some representative cases:

- For part (i), suppose \( s_x = [M] \), then

\[
\begin{align*}
\sigma(map(\lambda y \cdot [M])_x) &= \sigma(map(\lambda y \cdot M)_x) \\
&= [\sigma(M)].
\end{align*}
\]

- For part (i), suppose \( s_x = s'_x \cdot s''_x \), then

\[
\begin{align*}
\sigma(map(\lambda y \cdot s'_x \cdot s''_x)_x) &= \sigma((map(\lambda y \cdot s'_x)_x) \cdot (map(\lambda y \cdot s''_x)_x)) \\
&= \sigma(s'_x \cdot s''_x) \quad \text{by induction and (i)}.
\end{align*}
\]

- For part (i), suppose \( s_x = map(\lambda z \cdot s' z) s''_x \), then

\[
\begin{align*}
\sigma(map(\lambda y \cdot map(\lambda z \cdot s' z) s''_x)_x) &= \sigma(map(\lambda y \cdot s' y)(s''_x \circ (map(\lambda z \cdot s' z); s_y))) \\
&= \sigma(map(\lambda y \cdot s' y) s'') \quad \text{by induction and (ii)} \\
&= \sigma(s_y).
\end{align*}
\]

The cases for (ii) are essentially similar, but using (iii) in the induction hypothesis for the last case.

- For part (iii), suppose \( s_n = \lambda \overline{u}_n \cdot [M(\overline{u}_n)] \), then

\[
\begin{align*}
\sigma(map(\lambda x \cdot s_n(\pi_2 \pi'_1 x)_{i \in (n-1) \setminus 1})_y) &= \sigma(map(\lambda x \cdot [M(\pi_2 \pi'_1 x)_{i \in (n-1) \setminus 1}] y)_{i \in (n-1) \setminus 1}) \\
&= \sigma(map(\lambda x \cdot \pi_2 \pi'_1 x)_{i \in (n-1) \setminus 1} y)_{i \in (n-1) \setminus 1} \\
&= \sigma(map(\lambda x \cdot [M(\pi_2 \pi'_1 x)_{i \in (n-1) \setminus 1}] y)_{i \in (n-1) \setminus 1}) \\
&= \sigma(map(\lambda x \cdot n \pi_2 \pi'_1 x)_{i \in (n-1) \setminus 1} y)_{i \in (n-1) \setminus 1} \\
&= \sigma(map(\lambda x \cdot [M(\pi_2 \pi'_1 x)_{i \in (n-1) \setminus 1}] y)_{i \in (n-1) \setminus 1}).
\end{align*}
\]

- For part (iii), suppose \( s_n = \lambda \overline{u}_n \cdot map(\lambda y \cdot s' \overline{u}_n y)(s''_n \overline{u}_n) \), then

\[
\begin{align*}
\sigma(map(\lambda x \cdot (map(\lambda y \cdot s(\pi_2 \pi'_1 x)_{i \in (n-1) \setminus 1}) y)_{i \in (n-1) \setminus 1}))_{y} &= \sigma(map(\lambda x \cdot s(\pi_2 \pi'_1 x)_{i \in (n-1) \setminus 1}) y)_{i \in (n-1) \setminus 1} \\
&= \sigma(map(\lambda x \cdot s(\pi_2 \pi'_1 x)_{i \in (n-1) \setminus 1}) y)_{i \in (n-1) \setminus 1} \\
&= \sigma(map(\lambda x \cdot s(\pi_2 \pi'_1 x)_{i \in (n-1) \setminus 1}) y)_{i \in (n-1) \setminus 1} \\
&= \sigma(map(\lambda x \cdot n \pi_2 \pi'_1 x)_{i \in (n-1) \setminus 1} y)_{i \in (n-1) \setminus 1} \\
&= \sigma(map(\lambda x \cdot s_n n \pi_2 \pi'_1 x)_{i \in (n-1) \setminus 1} y)_{i \in (n-1) \setminus 1}.
\end{align*}
\]
• For part (iv), suppose $M = \lambda x_{n} \cdot u_{y}$:

  — If $j = 1$ then

  \[
  \sigma(\text{clos}(\lambda x \cdot M(\pi_{2} \pi_{1}^{n} x)_{i_{\omega((n-1))}} \mathcal{S}_{n-1})) \\
  = \sigma(\text{clos}(\lambda x \cdot \pi_{2} \pi_{1}^{n-1} x) \mathcal{S}_{n-1}) \\
  = \sigma(\text{clos}(\lambda x \cdot \pi_{2} x) \mathcal{S}_{0}) \\
  = \sigma(\text{clos}(\lambda x \cdot x) (\text{map} \lambda y \cdot s_{x} \ y) s_{y}) \\
  = \sigma(\text{clos}(\lambda x \cdot x) s_{x}) \text{ by the induction hypothesis} \\
  = \sigma(\text{clos}(\lambda x \cdot \pi_{1}^{n-1} x) \mathcal{S}_{n-1}) \\
  = \sigma(\text{clos}(\lambda x \cdot M \pi_{1}^{i-1} x(\pi_{2} \pi_{1}^{n} x)_{i_{\omega((n-2))}} \mathcal{S}_{n-1})).
  \]

  — If $2 < j \leq n$ then

  \[
  \sigma(\text{clos}(\lambda x \cdot M(\pi_{2} \pi_{1}^{n} x)_{i_{\omega((n-1))}} \mathcal{S}_{n-1})) \\
  = \sigma(\text{clos}(\lambda x \cdot \pi_{2} \pi_{1}^{n-2} x) \mathcal{S}_{n-1}) \\
  = \sigma(\text{clos}(\lambda x \cdot \pi_{2} x) \mathcal{S}_{j-1}) \\
  = \sigma(\text{clos}(\lambda x \cdot x) (\text{map} \lambda y \cdot s_{x_{j-1}} \ (\pi_{2} \pi_{1}^{n} y)_{i_{\omega((j-2))}} \mathcal{S}_{j-2})) \\
  = \sigma(\text{clos}(\lambda x \cdot \pi_{2} \pi_{1}^{n-2} x) \mathcal{S}_{j-2}) \text{ by induction and (iii)} \\
  = \sigma(\text{clos}(\lambda x \cdot \pi_{1}^{n-1} x) \mathcal{S}_{n-1}) \\
  = \sigma(\text{clos}(\lambda x \cdot M \pi_{1}^{i-1} x(\pi_{2} \pi_{1}^{n} x)_{i_{\omega((n-2))}} \mathcal{S}_{n-1})).
  \]

The verification of Claim 2 completes the proof for this case. Define

\[
\mathcal{M}_{n} \overset{\text{def}}{=} \sigma(\text{clos}(\lambda x \cdot M \pi_{2} \pi_{1}^{n} x \pi_{1}^{n+1} x(\pi_{2} \pi_{1}^{n} x)_{i_{\omega((n-1))}} \mathcal{S}_{n})) \\
\mathcal{S}_{0} \overset{\text{def}}{=} s_{y} \circ s_{x}.
\]
Then the claim is that:

(i) \( \mathcal{M}_n = \mathcal{M}_n \)

(ii) \( \text{map}(\lambda x \cdot s_n \pi_2 \pi_1^{n-1} x \pi_2 \pi_1^1 x) S_{n-1} = \text{map}(\lambda x \cdot s_n \pi_2 \pi_1^{n-1} x \pi_2 \pi_1^1 x) S_{n-1}' \).

The verification is similar to that for Lemma 4.4. For example, for the case where \( M = \lambda u_{n+2} \cdot u_i \) for some \( i \in \{2, ..., n+2\} \), we have the following cases:

(i) \( i = 1 \):

\[
\mathcal{M}_n = \mathcal{M}_n
\]

(ii) \( i = 2 \):

\[
\mathcal{M}_n = \mathcal{M}_n
\]

(iii) \( 2 < i \leq n+2 \):

\[
\mathcal{M}_n = \mathcal{M}_n
\]
The other case we consider is the base case for substitutions:

\[
\sigma(map(\lambda x \cdot [M \pi_2 \pi_1^n x \pi_1^n x(\pi_2 \pi_1^n x)_{\epsilon \in \{n-2,1\}}]) S_{n-1}) \\
= [\sigma(clos(\lambda x \cdot M \pi_2 \pi_1^n x \pi_1^n x(\pi_2 \pi_1^n x)_{\epsilon \in \{n-2,1\}}) S_{n-1}]) \\
= [\sigma(clos(\lambda x \cdot M \pi_2 \pi_1^n x \pi_1^n x(\pi_2 \pi_1^n x)_{\epsilon \in \{n-2,1\}}) S_{n-1}]) \\
\]

by the induction hypothesis

\[
= \sigma(map(\lambda x \cdot [M \pi_2 \pi_1^n x \pi_1^n x(\pi_2 \pi_1^n x)_{\epsilon \in \{n-2,1\}}]) S_{n-1}).
\]

(b) \( M' = \lambda x \cdot abs(M^n x) \) and \( N = clos(\lambda x \cdot abs(N^n x)) s \), where \( N^n \) results from \( M^n \) by reducing a Beta-redex. Then

\[
\sigma(M) = abs(\lambda x \cdot \sigma(clos(M^n x)) s) \\
= abs(\lambda x \cdot \sigma(clos(N^n x)) s) \quad \text{by induction} \\
= \sigma(N).
\]

(c) \( M' = \lambda x \cdot clos(M^n x)(s') x \) and \( N' = \lambda x \cdot clos(N^n x)(s' x) \), then

\[
\sigma(M) = \sigma(clos(\lambda z \cdot M^n \pi_1 z \pi_2 z)(s \cdot (map s')) s)) \\
= \sigma(clos(\lambda z \cdot N^n \pi_1 z \pi_2 z)(s \cdot (map s')) s)) \\
= \sigma(clos(\lambda x \cdot clos(N^n x)(s' x)) s).
\]

**FIGURE 2**
Theorem 2. \( \lambda^U_{\beta_0} \) is confluent on closed terms.

Proof. This is proved using Hardin’s interpretation technique (Abadi et al. 1991) and Lemma 4.1, Lemma 4.3, Corollary 4.2, and Lemma 4.5. Hardin’s technique amounts to verifying the diagram on the right in Fig. 2. In this diagram, single-headed arrows denote one-step reductions while double-headed arrows denote the reflexive transitive closure of the corresponding relation. Dashed edges denote reduction sequences whose existence can be verified.

Lemma 4.5 verifies the diagram on the left in Fig. 2, and repeated use of the lemma verifies that the two upper parallelograms in the right-hand diagram commute. Confluence for the original \( \beta \)-reduction relation verifies that the lower diamond commutes.

5. A TYPED \( \lambda\sigma\beta_0 \)-CALCULUS

In this section we develop a version of a higher-order-typed \( \lambda \)-calculus based on explicit higher-order substitutions, and use this as the basis for a type-checker for the calculus. This will serve to demonstrate the use of higher-order substitutions both for implementing \( \beta \)-reduction and for type-checking with dependent types.

There are several reasons for considering a type system such as that presented here. One is that one of the original applications of the \( \lambda\sigma \)-calculus was in the design of a type-checker for the second-order \( \lambda \)-calculus (Abadi et al. 1991), so this provides a point of comparison between the two systems. Type checkers for various higher-order \( \lambda \)-calculi are also offered as examples of the use of \( \beta_0 \)-unification (Miller 1991a), so this again serves as a point of comparison between our approach and other approaches to higher-order abstract syntax. Finally the type-checker provides an example of the use of the \( \lambda\sigma\beta_0 \)-calculus in a wider context involving (for example) the introduction of local constants. Such a facility is provided in the theory of hereditary Harrop formulae (Miller et al. 1991) that underlies languages such as \( \lambda \)-Prolog, Elf, and L\_ (Nadathur and Miller 1988, Pfenning 1990, Miller 1991a); recently proposals have been made for its addition to functional languages (Pitts and Stark 1993, Odersky 1994).

5.1. A Typed \( \lambda \)-Calculus

The calculus we use is a calculus of dependent function types and a constant denoting the type of all types. For simplicity we give this constant the type \( \text{type} \) itself. This has the effect that reduction in our calculus is not necessarily terminating (Girard 1972). However our focus here is on the use of explicit higher-order substitutions in implementing a type-checker, and so we do not pursue the complications that could be introduced to make the calculus terminating (Coquand 1986). We refer to this system as \( \lambda^H \); it is close to calculi proposed by Martin-Löf (1971) and Cardelli (1986).

We provide a presentation of \( \lambda^H \) using higher-order abstract syntax. A representation for terms of our \( \lambda^H \) subset is given by the metalanguage signature
Terms in the object language have the form

\[
\text{type, } (\text{pi } A B), (\text{abs } A M), (\text{apply } M N),
\]

representing respectively the type of all types, the dependent function type, \(\lambda\)-abstraction, and application. We have the usual \(\beta\)-reduction rule for this term calculus:

\[
(\text{apply}(\text{abs } A M) N) \rightarrow_{\beta} M(N).
\]

This gives rise to an untyped reduction relation, denoted \(M \rightarrow_{\beta} N\). The Church–Rosser property holds for this untyped reduction relation, due to the absence of \(\eta\)-reduction (Luo 1989).

Appendix B.1 gives the typing rules for the object language. We use judgements of the forms

Environments

\[ \Gamma ::= \text{nil} | \Gamma, x : A \]

Judgements

\[ \mathcal{J} ::= \mathcal{G}, \Gamma \vdash \text{env} \mid \Gamma \vdash \mathcal{G} M = N \in A. \]

To keep the number of rules to a minimum we present the system using equality judgements \(\Gamma \vdash \mathcal{G} M = N \in A\), with the abbreviation

\[ \Gamma \vdash \mathcal{G} M \in A \overset{\text{def}}{=} \Gamma \vdash \mathcal{G} M = M \in A. \]

Let \(\Gamma \vdash \mathcal{G} \mathcal{F}\) denote derivability of the judgement \(\Gamma \vdash \mathcal{G} \mathcal{F}\) using the rules of this type system.

**Proposition 5.1 (Church–Rosser).** If \(M \rightarrow_{\beta} N_1\) and \(M \rightarrow_{\beta} N_2\), then there is some \(N\) such that \(N_1 \rightarrow_{\beta} N\) and \(N_2 \rightarrow_{\beta} N\).

**Proof.** This is standard result, since reduction is \(\beta\)-reduction on the untyped raw calculus of \(\lambda^H\).

### 5.2. Typed Explicit Substitutions

The formulation of the \(\lambda^H\) object language in the previous section relied in several places on the use of \(\beta\)-reduction to implement substitution. In this section we remove this reliance on \(\beta\)-reduction in the metalanguage by making substitutions explicit in the object language. For brevity we refer to the resulting system as \(\lambda^H_{\text{ex}}\).
As for the untyped calculus, we introduce a new type constructor Subst into the object language for substitutions. The additions to the object language signature of the previous subsection are

\[\text{Subst}: \text{Type} \rightarrow \text{Type} \]

\[\text{clos}: \text{AS} \cdot (S \rightarrow \text{Term}) \rightarrow \text{Subst} S \rightarrow \text{Term} \]

\[\text{map}: \text{AS}_1 \cdot \text{AS}_2 \cdot (S_1 \rightarrow \text{Subst} S_2) \rightarrow \text{Subst} S_1 \rightarrow \text{Subst} S_2 \]

\[\boxed{\_ \circ \_}: \text{Term} \rightarrow \text{Term} \rightarrow \text{Subst Term} \]

\[\_ \circ \_ : \text{AS}_1 \cdot \text{AS}_2 \cdot \text{Subst} S_1 \rightarrow \text{Subst} S_2 \rightarrow \text{Subst}(S_1 \times S_2) \]

Here the clos term constructor represents the application of a substitution to a term. Basic substitutions are built using the \(\boxed{\_ \circ \_}\) constructor. Thus whereas in \(\_ \circ \_\) we had

\[
\text{Beta} \quad \Gamma, x: A \triangleright x M(x) \in B(x) \quad \Gamma \triangleright x N \in A \quad \Gamma \triangleright x (\text{apply}(\text{abs} A M) N) = \text{M}(N) \in B(N)
\]

In \(\_ \circ \_\) the rule is formulated as

\[
\text{Beta} \quad \Gamma, x: A \triangleright x M(x) \in B(x) \quad \Gamma \triangleright x N \in A \quad \Gamma \triangleright x (\text{apply}(\text{abs} A M) N) = (\text{clos} M[N, A]) \in (\text{clos} B[N, A])
\]

Appendix B.2 gives the basic type rules for \(\_ \circ \_\). Aside from the introduction of explicit substitutions, these rules do not differ much from the original type rules in Appendix B.1. The type rule which is noticeable by its absence is a rule for typing closures. In fact since our substitutions are essentially untyped at the object level such a rule is not sound with respect to the original system. Instead (as with the \(\_ \circ \_\)-calculus for System F (Abadi et al. 1991)) we present rules for pushing substitutions inside of terms and typing the result. In general deciding well typedness is inextricably tied with applying substitutions.

Appendix B.3 gives the rules for permuting substitutions with term constructors (including the ClosSubst rule for composing substitutions). Appendix B.4 gives the equivalence rules for substitutions, including rules for pushing substitutions inside of other substitutions. Here again we have a rule (MapSubst) for composing substitutions, analogous to ClosSubst.

The rules Beta, ClosConst, ClosVar, ClosL, ClosR, ClosP, ClosApp, ClosSubst, MapTerm, MapComp, and MapSubst constitute a PRS similar to that for the untyped \(\_ \circ \_\)-calculus. We again separate the PRS into two subsystems: \(\_ \circ \_\) (consisting of only the Beta rule) and \(\_ \circ \_\) (consisting of the remaining rules). We denote (untyped) reduction under \(\_ \circ \_\) and \(\_ \circ \_\) by \(\rightarrow_{\_ \circ \_}\) and \(\rightarrow_{\_ \circ \_}\), respectively. Untyped \(\beta\)-reduction over terms of \(\_ \circ \_\) is again denoted by \(\rightarrow_{\_ \circ \_}\).

\footnote{We omit type annotations from terms in the examples which follow, relying on type reconstruction to recover the missing type information.}
Lemma 5.1 (Termination of $\lambda^*_\alpha$). The PRS $\lambda^*_\alpha$, i.e., terminating.

Proof: This is similar to the proof for Lemma 4.1.

The following result justifies the use of $\sigma(M) (\sigma(A))$ to denote the $\sigma$-normal form of the term $M$ (type $A$) in $\lambda^*_\alpha$.

Lemma 5.2 (Confluence of $\lambda^*_\alpha$). $\lambda^*_\alpha$ is confluent on closed terms, i.e., if $M \Rightarrow^* N_1$ and $M \Rightarrow^* N_2$, then there exists an $N$ such that $N_1 \Rightarrow^* N$ and $N_2 \Rightarrow^* N$.

Proof: This is similar to the proof for Lemma 4.3.

The following result is the basis for the type-checking algorithm presented in the next subsection.

Theorem 3. $\lambda^*_\alpha$ is confluent on closed terms.

Proof: This is similar to the proof for Theorem 2.

Theorem 4 (Soundness of $\lambda^*_\alpha$). 1. If $\Gamma \vdash M \sim N$ env then $\Rightarrow^\alpha \sigma(\Gamma) = \sigma(\Gamma')$ env.
2. If $\Gamma \vdash M \sim N \in A$ then $\Rightarrow^\alpha \sigma(M) = \sigma(N) \in \sigma(A)$.
3. If $\Gamma \vdash \sigma s \sim t \in \text{Subst}(S)$ then
   
   (a) if $S = \text{Term}$ then $\sigma(s) = [M, A]$ and $\sigma(t) = [M', A']$ for some $M, A, M', A'$ such that $\sigma(\Gamma) \vdash^\alpha A = A' \in \text{type}(i)$ (for some $i$) and $\sigma(\Gamma) \vdash^\alpha M = M' \in A$.
   
   (b) Otherwise $S = (S_1 \times S_2)$ for some $S_1, S_2$; then $\sigma(s) = s_1 \circ s_2$ and $\sigma(t) = s'_1 \circ s'_2$ for some $s_1, s_2, s'_1, s'_2$ such that $\Gamma \vdash^\alpha s_1 \sim s'_1 \in \text{Subst}(S_1)$ and $G \vdash^\alpha s_2 \sim s'_2 \in \text{Subst}(S_2)$.

Proof: This is routine induction on the height of derivations in $\lambda^*_\alpha$, using the fact that the typing of closures and substitutions involves applying the rewrite rules of $\lambda^*_\alpha$ and then typing the result. For Part 3, the base cases in the induction for (a) and (b) are SubstTerm and SubstComp respectively. For SubstTerm, apply Part 2 of the induction hypothesis to the premises of the rule.

It is unclear how to obtain an analogous completeness result. Although it has been suggested that this can be done for the $\lambda_\sigma$-calculus by rewriting closures to Beta-redices (Abadi et al. 1991), this does not seem to adequately handle definitional equality in the type system.

5.3. A Type-Checking Algorithm

Finally we briefly present a type-checking algorithm for $\lambda^*_\alpha$ based on the system presented in the previous section. Appendix B.5 presents the type-checker as a collection of inference rules, where closures are type-checked essentially by pushing substitutions inside of terms and type-checking the result.

These rules use numerous auxiliary algorithms. The rules for checking for convertibility (in Appendix B.6) amounts to interleaving reductions to WHNF with recursive checking of subterms. Appendices B.7 and B.9 give the algorithms for reducing terms and substitutions, respectively, to WHNF. Finally Appendix B.8 gives the algorithm
for type-checking substitutions. We allow $\Gamma \triangleright \ M \rightarrow N$ and $G \triangleright M \leftrightarrow N$ as abbreviations for $\Gamma \triangleright \ M \rightarrow N \in A$ and $\Gamma \triangleright M \leftrightarrow N \in A$, respectively, where the inferred type is not important.

Lemma 5.3. 1. If $\Gamma \vdash \ A \in A$ then $\Gamma \vdash \ A \in type$.
2. If $\Gamma \vdash \ M \in A$ and $\Gamma \vdash \ M \in B$, then $\Gamma \vdash \ A \sim B \in type$.

Proof. A standard proof using induction over type derivations.

For the judgement forms $\Gamma \triangleright \ M \in A$, $\Gamma \triangleright M \leftrightarrow N \in A$, $\Gamma \triangleright s \in Subst(S)$, and $\Gamma \triangleright s \rightarrow t \in Subst(S)$, let $\Gamma \vdash_{\text{alg}} M \in A$, $\Gamma \vdash_{\text{alg}} M \rightarrow N \in A$, $\Gamma \vdash_{\text{alg}} M \rightarrow N \in A$, $\Gamma \vdash_{\text{alg}} s \in Subst(S)$, and $\Gamma \vdash_{\text{alg}} s \rightarrow t \in Subst(S)$, respectively, denote derivability according to the inference rules of the type-checking algorithm. The statement of soundness for the type-checker is then given by:

Theorem 5. Suppose $\vdash \ \Gamma \sim \ \Gamma_{\text{env}}$. Then:

1. If $\Gamma \vdash_{\text{alg}} M \in A$ then $\Gamma \vdash_{\text{alg}} M \in A$.
2. If $\Gamma \vdash_{\text{alg}} M \rightarrow N \in A$ then $\Gamma \vdash_{\text{alg}} M \sim n \in A$.
3. If $\Gamma \vdash_{\text{alg}} M \rightarrow N \in A$ then $\Gamma \vdash_{\text{alg}} M \sim N \in A$.
4. If $\Gamma \vdash_{\text{alg}} s \in Subst(S)$ then $\Gamma \vdash_{\text{alg}} s \in Subst(S)$.
5. If $\Gamma \vdash_{\text{alg}} s \rightarrow t \in Subst(S)$ then $\Gamma \vdash_{\text{alg}} s \sim t \in Subst(S)$.

Proof. By induction on derivation trees describing executions of the type-checking algorithm. We give some representative cases.

For ClosAbs, applying the induction hypothesis (Part 1) to the first premise ($\Gamma \vdash_{\text{alg}} clos A \ s \in A'$) gives a derivation for $\Gamma \vdash_{\text{alg}} clos A \ s \in A'$. Applying the induction hypothesis (Part 4) to the second premise ($\Gamma \vdash_{\text{alg}} A' \rightarrow \text{type} \in C$) gives a derivation for $\Gamma \vdash_{\text{alg}} A' \sim \text{type} \in C$. Applying the Cum rule for $\lambda M \ s$ gives a derivation for $\Gamma \vdash_{\text{alg}} clos A \ s \sim (\Gamma, x : clos A) s$ env, allowing us to apply the induction hypothesis to the third premise to obtain ($\Gamma, x : clos A) s$ $\vdash_{\text{alg}} clos(\lambda y : M \ y) s \in B x$. An application of Abs gives a derivation for

$$\Gamma \vdash_{\text{alg}} (\text{abs}(clos A) s)(\lambda x : clos(\lambda y : M \ y) x) s \in (p(\text{clos} A s) B).$$

Then an application of ClosAbs gives

$$\Gamma \vdash_{\text{alg}} clos(\lambda y : \text{abs}(A y) \ y) s$$

$$\sim \text{abs}(clos A) s)(\lambda x : clos(\lambda y : M \ y) x) s \in (p(\text{clos} A s) B),$$

and a final application of the symmetry and transitivity rules gives the required conclusion:

$$\Gamma \vdash_{\text{alg}} clos(\lambda x : clos(A x) \ M) s \in (p(\text{clos} A s) B).$$
For EqApp, applying the induction hypothesis and Eq to the premises gives derivations for

\[ \Gamma \vdash_{\sigma} M_1 \sim N_1 \in (\pi A B) \]
\[ \Gamma \vdash_{\sigma} M_2 \sim N_2 \in A. \]

Applying App gives the conclusion \( \Gamma \vdash_{\sigma} (\text{apply } M_1 M_2) \sim (\text{apply } N_1 N_2) \in (\text{clos } B[M_2, A]). \) Then applying the induction hypothesis again, Lemma 5.3, Eq, and transitivity give \( \Gamma \vdash_{\sigma} M \sim N \in (\text{clos } B[M_2, A]). \)

For RedAppAbs, applying the induction hypothesis and Eq gives \( \Gamma \vdash_{\sigma} M \sim (\text{abs } A M) \in (\pi A B) \) and \( \Gamma \vdash_{\sigma} N \sim A \), for some \( B \). Applying rule App gives \( \Gamma \vdash_{\sigma} (\text{apply } (\text{abs } A M) N) \sim (\text{clos } B[M, A]) \in (\text{clos } B[N, A]). \) Applying App again and transitivity gives \( \Gamma \vdash_{\sigma} (\text{apply } M N) \sim (\text{clos } B[N, A]) \in (\text{clos } B[N, A]). \) Applying the induction hypothesis again, Lemma 5.3, Eq, and transitivity give \( \Gamma \vdash_{\sigma} (\text{apply } M N) \sim \text{M''} \in C. \)

5.4. Discussion

We conclude by comparing our approach to metaprogramming (as exemplified by the type-checker in this section) to that of HOAS and of the \( \lambda \sigma \)-calculus. In HOAS object language variables are represented by \( \lambda \)-bound metalanguage variables (replaced by locally introduced constants), whereas in the \( \lambda \sigma \)-calculus object language variables are represented by de Bruijn indices. The \( \lambda \sigma \_0 \)-calculus to some extent combines aspects of both: object language variables are represented by metalanguage variables (as in HOAS), whereas variables that are “bound” by substitutions are represented by locators applied to the \( \lambda \)-bound variable of a closure (somewhat analogous to the \( \lambda \sigma \)-calculus). This latter might appear to suggest some similarity between located variables in the \( \lambda \sigma \_0 \)-calculus and de Bruijn indices in the \( \lambda \sigma \)-calculus (as already noted in Section 4, they share a common categorical explanation). However as discussed at the conclusion of this subsection, a better comparison is between located variables and the AT-tree approach to substitutions also introduced by de Bruijn (de Bruijn 1987, 1993).

Although the \( \lambda \sigma \)-calculus is an important and seminal development in the theory of metaprogramming, it has the drawback that it is fairly low level, working at the level of the implementation of object language variables. It also has the unfortunate drawback that there are surprising and subtle issues to be considered in the manipulation of substitutions. Consider for example the typing of type closures in the second-order \( \lambda \sigma \)-calculus provided by Abadi et al. (1991). Intuitively we would like this type rule to be (where \( A[x] \) denotes a closure in the \( \lambda \sigma \)-calculus with substitution \( x \))

\[
\frac{\Gamma \vdash A[x] \in \text{Type} \quad \Gamma \vdash B \in \text{Type}}{\Gamma \vdash (A \to B)[x] \in \text{Type}}.
\]

but there is a subtlety here: \( A \to B \) denotes the type of a function \( \lambda x : A \cdot M \), for some \( M \). Even though the type \( B \) does not depend on the term variable \( x \) (in
System F, the above rule must reflect the fact that the type $B[x]$ is “kind-checked” in a context extended with the type binding for $x$. Therefore we have

$$
\frac{\Gamma \triangleright_{\mathcal{X}} A[x] \in Type \quad \Gamma \triangleright_{\mathcal{X}} B[1 : A \cdot (s \uparrow)] \in Type}{\Gamma \triangleright_{\mathcal{X}} (A \rightarrow B)[x] \in Type}
$$

The corresponding rule in the $\lambda\sigma\beta_\alpha$-calculus would be the original intuitive rule above.

Languages such as $\lambda$-Prol and Elf attempt to provide a more high-level approach to manipulating object languages. We have already discussed in Section 1 the motivation for providing substitution as an implementable operation in the metalanguage, rather than relying on it as a built-in primitive operation, with the approach of HOAS. The $L_\lambda$ language (Miller 1991a), based on $\lambda_\alpha$-unification, attempts to do just this. For example $\beta$-reduction to weak head normal form in a simple $\lambda$-calculus object language is represented in $L_\lambda$ as

$$
copy(app M N)(app M' N') :\text{-} copy M M', \ copy N N',
$$
$$
copy(abs M)(abs M') :\forall x. \forall y. \ copy x N \Rightarrow \copy(M x)(M' y),
$$
$$
subst M N M' :\forall x. \ copy x N \Rightarrow \copy(M x) M',
$$
$$
red(app(abs M) N) M' :\ subst M N M',
$$
$$
red(app M N)(app M' N) :\ red M M',
$$
$$
whnf M M' :\ red M M', \ whnf M' M'',
$$
$$
whnf M M.
$$

A similar implementation is possible using Nipkow’s higher-order rewrite systems. Such an implementation is given by rules ClosConst, ClosVar, ClosApp, and ClosAbs in Fig. 1. As mentioned in Section 1, this implementation of substitution is considerably less efficient than the corresponding “built-in” implementations (which are typically based on the $\lambda\sigma$-calculus (Nadathur and Wilson 1990)). The Elf implementation explicitly chose not to follow this approach because of this problem (Michaylov and Pfenning 1992), relying on built-in substitution (although not resorting to full higher-order unification, because of its poor operational behavior). Moreover it is not clear how to reason about such metaprograms (as is done for example with the $\lambda\sigma$-calculus).

De Bruijn (1987, 1993) has proposed an alternative to de Bruijn numbers for reasoning about variable-binding with substitutions. This approach is based on terms as binary trees, with application vertices labeled with $A$ and $\lambda$-abstraction nodes labeled with $T$ (hence the name $AT$-tree). Associated with such a tree is a labeling function lab from leaf vertex tree addresses to the tree addresses of the corresponding binding $T$-vertices. The main tree operation is implantation, replacing
a leaf vertex by a subtree, with a new labeling function computed for the leaf vertices of this subtree. Implantation is used to define local $\beta$-reductions, which replace a leaf vertex in a $T$-tree with the corresponding $A$-tree in an $AT$-pair (see Fig. 3). $\beta$-reduction is then defined as a series of local $\beta$-reductions, followed by the removal of the redundant $AT$-tree.

A closure can be viewed as a tree whose right spine consists of a path of $A$-vertices followed by a path of $T$-vertices of the same length. The corresponding labeling function maps variables (leaf vertices) in the body of the closure to the tree address of the corresponding $\lambda$-binder, from which the tree address of the subtree to which it is bound in the closure may be computed. Calculi of explicit substitutions allow such a closure tree to be inverted, turning it into a binary tree where the rightmost tree is the usual $AT$-tree while the leftmost tree records the substitution; the tree address labels associated with variables bound by the closure refer to the corresponding subtrees in the substitution subtree. The contribution of the $\lambda\alpha\beta_0$-calculus is to manage to “hide” the details of the implementation of “free” variables, using HOAS, while also allowing closures to be inverted in this manner.

6. IMPLEMENTATION

Finally we consider the implementation of higher-order substitutions. There are some complications with obtaining an efficient implementation of the unification algorithm, primarily with flexible-flexible pairs. We concentrate therefore on the case where the equations for the calculus of higher-order substitutions in Fig. 1, oriented as a rewrite rule system, are guaranteed to be ground. In this scenario, matching alone is sufficient for an implementation of the rewrite system.

The observation to be made is that for the most part high-order matching is only required to maintain consistent variable bindings on the left and right hand sides of a rewrite rule. This suggests an implementation of the underlying variable binding in terms of explicit first-order renaming substitutions. We define an untyped first-order metalanguage in which to express this implementation.
where the rewrite rules for this calculus are

\[ L_1 \circ (L_2 \circ L_3) = (L_1 \circ L_2) \circ L_3 \]  \hspace{1cm} (1)
\[ \text{id} \circ L = L \]  \hspace{1cm} (2)
\[ L \circ \text{id} = L \]  \hspace{1cm} (3)
\[ L_1(L_2 A) = (L_1 \circ L_2) A \]  \hspace{1cm} (4)
\[ \text{id} A = A \]  \hspace{1cm} (5)
\[ (M_1 M_2)[s] = M_1[s] M_2[s] \]  \hspace{1cm} (6)
\[ (A M)[s] = A[M[1 \circ \uparrow]] \]  \hspace{1cm} (7)
\[ (M[s_1][s_2]) = M[s_1 \circ s_2] \]  \hspace{1cm} (8)
\[ (L n)[\uparrow] = L(n + 1) \]  \hspace{1cm} (9)
\[ (L n)[\uparrow \circ s] = (L(n + 1))[s] \]  \hspace{1cm} (10)
\[ (L 1)[A s] = LA \]  \hspace{1cm} (11)
\[ (L(n + 1))[A s] = (L n)[s] \]  \hspace{1cm} (12)
\[ (s_1 \circ s_2) \circ s_3 = s_1 \circ (s_2 \circ s_3) \]  \hspace{1cm} (13)
\[ (A s_1) \circ s_2 = A[s_2] \circ s_2 \]  \hspace{1cm} (14)
\[ \uparrow \circ (A s) = s \]  \hspace{1cm} (15)
\[ \text{id} \circ s = s \]  \hspace{1cm} (16)
\[ s \circ \text{id} = s \]  \hspace{1cm} (17)
\[ M[\text{id}] = M. \]  \hspace{1cm} (18)

In this implementation calculus for the metalanguage there is a variable binding operation \( A(M) \) and a renaming operation \( M[s] \). Although we use a calculus of substitutions, it should be noted that substitution is only being used here as a form of renaming (the extended \( \beta \)-reduction relation described by Duggan (1997)). Then for example the translation of the rewrite rule for \texttt{ClosApp} in Fig. 1 is (ignoring type annotations)

\[ \text{clos}(A(\text{app}(M; N)); s) = \text{app}(\text{clos}(A(M)); s; \text{clos}(A(N)); s)), \]
reflecting the fact that \( \lambda \)-abstraction is being used in this rule primarily to permute a variable binder with a term constructor. The translation for \texttt{ClosAbs} is

\[
\text{clos}(\text{abs}(\text{A}(M))); s) = \text{abs}(\text{clos}(\text{A}(M[2.1 .id ]); s[\uparrow ])).
\]

As with the original first-order calculus, variable bindings in the substitution \( s \) must be shifted to reflect the fact that it is being moved inside of a variable binder. In addition we incur the overhead of performing a renaming substitution on \( M \), because two variable binders have been permuted. The translation for the \texttt{ClosSubst} rule is

\[
\text{clos}(\text{A}(\text{clos}(\text{A}(M)); s_1)); s_2) = \text{clos}(\text{A}(M[\pi_2 1.\pi_1 1 .id ]); s_2 \circ \text{map}(\text{A}(s_1); s_2)).
\]

Finally we consider the cases for \texttt{ClosConst}, \texttt{ClosVar}, \texttt{ClosL}, and \texttt{ClosR}. These are essentially the projection step in Nipkow’s implementation of patterns (Nipkow 1993a). If we treat these as the “default rules” in the rewrite system, then we obtain an acceptably efficient implementation:

\[
\begin{align*}
\text{clos}(\text{A}(\pi_1 1); s_1 \circ s_2) &= \text{clos}(\text{A}(1); s) \\
\text{clos}(\text{A}(L \cdot \pi_1 1); s_1 \circ s_2) &= \text{clos}(\text{A}(L 1); s) \\
\text{clos}(\text{A}(1); [M]) &= M \\
\text{clos}(\text{A}(n + 1); s) &= n + 1 \\
\text{clos}(\text{A}(L(n + 1)); s) &= L(n + 1).
\end{align*}
\]

7. CONCLUSIONS

We have presented an approach to incorporating explicit substitutions into higher-order abstract syntax. Our approach shares the same theoretical properties as the \( \lambda \sigma \)-calculus, and shows how substitutions may be reified in \( \lambda \text{-Prolog} \)-like languages to the program level. As such our approach provides a facility for manipulating abstract syntax at a high level, while allowing the programmer to explicitly manipulate substitutions where necessary without needing to modify the run-time implementation (as is typically done with \( \lambda \text{-Prolog} \)-like languages). We have presented an implementation of our calculus using a first-order calculus of renaming substitutions, which allows our approach to be as efficient as the first-order approach of the \( \lambda \sigma \)-calculus.

Explicit substitutions have seen much application in the implementation of type checkers and proof checkers. For example the MLF system (Harper and Pfenning 1996) uses explicit substitutions in the implementation of a dependent-type based logical frameworks. Explicit substitutions have also seen application in the implementation of higher-order unification and constraint-solving engines for higher-order abstract syntax (Dowek \textit{et al.} 1996). The ALF proof editor (Magnusson 1995) makes substitutions available in the proof editor as explicit objects of the metalanguage. However these substitutions are more akin to higher-order closures, and there does
not appear to be any mechanism for composing substitutions at the programmer level (the central concern of the $\lambda\sigma$-calculus, and the main motivation for this work).

Although we have used Girard’s System $F^\omega$ (with products) as a metalanguage, our calculus is not tied to an impredicative type system and could as well be formulated using, say, the two-level predicative type system underlying the ML programming language. On the other hand, the $\lambda\sigma\beta_0$-calculus does rely crucially on polymorphic recursion. As is now known, ML-style type inference with polymorphic recursion is undecidable (Henglein 1993, Kfoury et al. 1993). If the $\lambda\sigma\beta_0$-calculus is to be implemented in a language with decidable-type inference, some degree of explicit typing appears necessary. Such a facility is in the Haskell language design (Hudak et al. 1992), allowing the definition of polymorphically recursive functions provided such functions are given explicit type signatures.

The $\lambda\sigma\beta_0$-calculus makes use of an extension of $\beta_0$-unification developed in another paper (Duggan 1997). Although the matching algorithm implied by this algorithm is sufficient for higher-order substitutions, the full unification algorithm is still useful as part of a tool for checking for critical pairs in rules which make use of explicit higher-order substitutions. Provided this rewrite system is terminating, the critical pair lemma verified in Section 2 provides a way for checking for local confluence and hence confluence. Finally, because explicit higher-order substitutions allow the right-hand sides as well as the left-hand sides of pattern rewrite systems to satisfy the extended pattern restriction, the unification algorithm described in (Duggan 1997) and the critical pair lemma verified in Section 2 provide a generalization of Knuth–Bendix completion (Knuth and Bendix 1970) to our framework of higher-order rewrite systems with explicit substitutions.

It appears clear that higher-order substitutions can be generated automatically from a second-order signature for an abstract syntax, along the lines of the systems in Sections 4 and 5. It appears plausible that this could be extended to a signature of any order, using the approach demonstrated by Miller for reducing higher-order unification to programming in $L_\downarrow$ (Miller 1991b).

Since the $\lambda\sigma\beta_0$-calculus introduces explicit names into a calculus of explicit substitutions, an important topic for future work is to develop the calculus to incorporate some notion of sharing. Field (1990) suggests some ways to take advantage of sharing in actual implementations of substitutions, and the importance of sharing is for example emphasized by Huet in his implementation of the Calculus of Constructions (Huet 1988), and by experience with the implementation of the Standard ML module system (MacQueen 1988). Recently proposals have been made for adding local naming to the $\lambda$-calculus (Pitts and Stark 1993, Odersky 1994), and this appears to be a plausible direction for future work.

APPENDIX A: TYPE RULES FOR METALANGUAGE

A.1. Description of the Metalanguage

In this appendix we present the functional subset of the type theory underlying our metalanguage. We use judgements of the form $\Gamma \mathsf{env}_\Sigma$ and $\Gamma \vdash_\Sigma \mathcal{F}$, where $\Sigma$
is a signature of typings for constants, $\Gamma$ is a typing environment, and $\mathcal{F}$ is a formula of the metalogic. Formulæ of the metalogic and terms of the language are organized into the categories

$$\mathcal{F} \in \text{Formulæ ::= } K \text{ kind } | A \in K | M \in A | A = B | M = N$$

$$\Gamma \in \text{Envs ::= } \text{nil} | \Gamma, t : K | \Gamma, x : A$$

$$K \in \text{Kinds ::= } Type | K_1 \rightarrow K_2$$

$$A, B \in \text{Types ::= } \text{tc} | t | A \rightarrow B | A t : K \cdot A | (A B)$$

$$M, N \in \text{Terms ::= } \text{c} | x | \lambda x : A \cdot M | (M N) | \lambda t : K \cdot M | M[ A ].$$
Subject reduction for types. If $\Gamma \vdash_{\Sigma} A \rightarrow A' \in K$, then $\Gamma \vdash_{\Sigma} A' \in K$.

Strong normalization for types. There does not exist an infinite sequence of rewrite steps $\Gamma \vdash_{\Sigma} A_i \rightarrow A_{i+1} \in K$ for $i \in \omega$.

Confluence for terms. If $\Gamma \vdash_{\Sigma} N_1 \in A$, $\Gamma \vdash_{\Sigma} N_2 \in A$, $\Gamma \vdash_{\Sigma} M \rightarrow N_1 \in A$ and $\Gamma \vdash_{\Sigma} N_2 \rightarrow M' \in A$, then there exists some $M'$ such that $\Gamma \vdash_{\Sigma} N_1 \rightarrow M' \in A$ and $\Gamma \vdash_{\Sigma} N_2 \rightarrow M' \in A$.

Subject reduction for terms. If $\Gamma \vdash_{\Sigma} M \rightarrow M' \in A$, then $\Gamma \vdash_{\Sigma} M' \in A$.

Strong normalization for terms. There does not exist an infinite sequence of rewrite steps $\Gamma \vdash_{\Sigma} M_i \rightarrow M_{i+1} \in A$ for $i \in \omega$.

A.2. Formation Rules for Environments

\[
\begin{align*}
\text{Const} & \quad \frac{\text{tc} \in \text{dom}(\Sigma)}{\Gamma \vdash_{\Sigma} \text{tc} \in \Sigma(\text{tc})} \quad \frac{c \in \text{dom}(\Sigma)}{\Gamma \vdash_{\Sigma} c \in \Sigma(c)} \\
\text{Env} & \quad \frac{\{ \} \ \text{env}_\Sigma \quad \Gamma \vdash_{\Sigma} K \ \text{kind}_\Sigma \quad \Gamma \vdash_{\Sigma} A \in \text{Type}}{\Gamma, \ t : K \ \text{env}_\Sigma} \\
\text{Var} & \quad \frac{\Gamma \vdash_{\Sigma} t \in \text{dom}(\Gamma)}{\Gamma, \ t \in \text{dom}(\Gamma)} \quad \frac{\Gamma \vdash_{\Sigma} x \in \text{dom}(\Gamma)}{\Gamma, \ x : A \ \text{env}_\Sigma} \\
\text{Type} & \quad \frac{\Gamma \vdash_{\Sigma} \text{type} \ \text{kind}_\Sigma}{\Gamma \vdash_{\Sigma} \text{type} \ \text{kind}_\Sigma}
\end{align*}
\]

A.3. Formation Rules for Types and Type Operators

\[
\begin{align*}
\rightarrow F & \quad \frac{\Gamma \vdash_{\Sigma} K_1 \ \text{kind}_\Sigma \quad \Gamma \vdash_{\Sigma} K_2 \ \text{kind}_\Sigma}{\Gamma \vdash_{\Sigma} K_1 \rightarrow K_2 \ \text{kind}_\Sigma} \\
\rightarrow I & \quad \frac{\Gamma, \ t : K_1 \ \text{env}_\Sigma \quad A \in K_2}{\Gamma \vdash_{\Sigma} \lambda t : K_1 : A \rightarrow K_2} \\
\rightarrow E & \quad \frac{\Gamma \vdash_{\Sigma} A \rightarrow K_1 \quad \Gamma \vdash_{\Sigma} B \in K_1}{\Gamma \vdash_{\Sigma} (A \ B) \in K_2} \\
\rightarrow \beta & \quad \frac{\Gamma, \ t : K_1 \ \text{env}_\Sigma \quad A \in K_2 \quad \Gamma \vdash_{\Sigma} B \in K_1}{\Gamma \vdash_{\Sigma} (\lambda t : K_1 : A) \ B = [B/t] \ A} \\
\rightarrow \eta & \quad \frac{\Gamma \vdash_{\Sigma} A \rightarrow K_1 \quad \Gamma \vdash_{\Sigma} B \in K_2 \quad t \notin \text{dom}(\Gamma)}{\Gamma \vdash_{\Sigma} \lambda t : K_1 : A \ t = A}
\end{align*}
\]

A.4. Formation Rules for Terms

\[
\begin{align*}
\rightarrow F & \quad \frac{\Gamma \vdash_{\Sigma} A \in \text{Type} \quad \Gamma \vdash_{\Sigma} B \in \text{Type}}{\Gamma \vdash_{\Sigma} A \rightarrow B \in \text{Type}} \\
\rightarrow I & \quad \frac{\Gamma, \ x : A \ \text{env}_\Sigma \quad M \in B}{\Gamma \vdash_{\Sigma} \lambda x : A : M \in A \rightarrow B}
\end{align*}
\]
→ E
\[ \frac{\Gamma \vdash M \in A \rightarrow B \quad \Gamma \vdash N \in A}{\Gamma \vdash (MN) \in B} \]

→ β
\[ \frac{\Gamma, x : A \vdash M \in B \quad \Gamma \vdash N \in A}{\Gamma \vdash (\lambda x : A \cdot M) N = \{N[x] \cdot M\}} \]

→ η
\[ \frac{\Gamma \vdash M \in A \rightarrow B \quad x \notin \text{dom}(\Gamma)}{\Gamma \vdash \lambda x : A \cdot M \ x = M} \]

ΔF
\[ \frac{\Gamma, t : K \vdash A \in \text{Type}}{\Gamma \vdash \Delta t : K \cdot A \in \text{Type}} \]

ΔI, ΔE
\[ \frac{\Gamma, t : K \vdash M \in A \quad \Gamma \vdash M \in \Sigma t : K \cdot A \quad \Gamma \vdash B \in K}{\Gamma \vdash \Delta t \cdot M([B/t]) \in [B/t] \cdot A} \]

Δβ, Δη
\[ \frac{\Gamma, t : K \vdash M \in A \quad \Gamma \vdash B \in K \quad \Gamma \vdash M \in \Delta t : K \cdot A \quad t \notin \text{dom}(\Gamma)}{\Gamma \vdash \Delta t \cdot M([B/t]) = M} \]

TyConv
\[ \frac{\Gamma \vdash M \in A \quad \Gamma \vdash A' \in \text{Type} \quad \Gamma \vdash A = A'}{\Gamma \vdash M \in A'} \]

APPENDIX B: TYPE RULES AND TYPE CHECKER FOR $\lambda^N$

B.1. Type Rules for $\lambda^N$

EnvNil
\[ \triangledown_\emptyset \text{nil} = \text{nil env} \]

EnvExt
\[ \frac{\triangledown_\emptyset \Gamma = \Gamma' \text{ env}}{\triangledown_\emptyset (\Gamma, x : A) = (\Gamma', x : A') \text{ env}} \quad (x \text{ new}) \]

EnvSym
\[ \frac{\triangledown_\emptyset \Gamma = \Gamma' \text{ env}}{\triangledown_\emptyset \Gamma' = \Gamma \text{ env}} \]

EnvTrans
\[ \frac{\triangledown_\emptyset \Gamma_1 = \Gamma_2 \text{ env} \quad \triangledown_\emptyset \Gamma_2 = \Gamma_3 \text{ env}}{\triangledown_\emptyset \Gamma_1 = \Gamma_3 \text{ env}} \]

Var
\[ \frac{\triangledown_\emptyset (\Gamma, x : A, \Gamma') = (\Gamma, x : A, \Gamma') \text{ env}}{\Gamma, x : A, \Gamma' \vdash x = x \in A} \]

Type
\[ \frac{\triangledown_\emptyset \Gamma = \Gamma' \text{ env}}{\Gamma \vdash \text{type} \in \text{type}} \]

Pi
\[ \frac{\Gamma \vdash A = A' \in \text{type} \quad \Gamma, x : A \vdash B(x) = B'(x) \in \text{type}}{\Gamma \vdash (\lambda A B) = (\lambda A' B') \in \text{type}} \]

Abs
\[ \frac{\Gamma \vdash A = A' \in \text{type} \quad \Gamma, x : A \vdash M(x) = M'(x) \in B(x)}{\Gamma \vdash (\text{abs} A M) = (\text{abs} A' M') \in (\text{abs} A B)} \]

App
\[ \frac{\Gamma \vdash M = M' \in (\text{abs} A B) \quad \Gamma \vdash N = N' \in A}{\Gamma \vdash (\text{apply} M N) = (\text{apply} M' N') \in B \cdot N} \]
**B.2. Type Rules for $\beta^H$**

\[ \text{Type} \]
\[ \Gamma \vdash A \sim A' \quad \text{type} \sim \text{type} \]
\[ \text{Pi} \]
\[ \Gamma \vdash A \sim A' \quad \text{type} \quad \Gamma, x : A \vdash B(x) \sim B'(x) \quad \text{type} \]
\[ \Gamma \vdash (\text{pi} A B) \sim (\text{pi} A' B') \quad \text{type} \]
\[ \text{Abs} \]
\[ \Gamma \vdash A \sim A' \quad \text{type} \quad \Gamma, x : A \vdash M(x) \sim M'(x) \quad \text{type} \]
\[ \Gamma \vdash _x (\text{abs} A M) \sim (\text{abs} A' M') \quad \text{type} \]
\[ \text{App} \]
\[ \Gamma \vdash M \sim M' \quad (\text{pi} A B) \quad \Gamma \vdash N \sim N' \quad A \]
\[ \Gamma \vdash \text{apply}(M N) \sim (\text{apply} M' N') \quad (\text{clos} B[N, A]) \]
\[ \text{Beta} \]
\[ \Gamma \vdash (\text{apply}(A M) N) \sim (\text{clos} M[N, A]) \quad (\text{clos} B[N, A]) \]
\[ \text{Eq} \]
\[ \Gamma \vdash M \sim M' \quad \text{type} \quad \Gamma \vdash A \sim A' \quad \text{type} \]
\[ \Gamma \vdash M \sim M' \quad (A') \]
B.3. Substitution Rules for $\lambda^N_{\text{var}}$

\[
\begin{align*}
\text{ClosCong} & \quad \Gamma \triangleright s \sim t \in \text{Subst}(S) \quad \Gamma \triangleright \text{clos}(\text{clos} M s') \in A \\
& \quad \Gamma \triangleright \text{clos}(\text{clos} M s) \sim \text{clos}(\text{clos} M t) \in A \\
\text{ClosConst} & \quad \Gamma \triangleright s \in \text{Subst}(S) \quad \Gamma \triangleright M \in A \\
& \quad \Gamma \triangleright \text{clos}(\lambda y \cdot M) s \sim M \in A \\
\text{ClosVar} & \quad \Gamma \triangleright [M, A] \in \text{Subst Term} \\
& \quad \Gamma \triangleright \text{clos}(\lambda y \cdot [M, A]) s \sim M \in A \\
\text{ClosL} & \quad \Gamma \triangleright s_2 \in \text{Subst}(S) \quad \Gamma \triangleright \text{clos} M s_1 \in B \\
& \quad \Gamma \triangleright \text{clos}(\lambda x \cdot M \pi_1 x(s_1 \cdot s_2)) \sim \text{clos} M s_1 \in B \\
\text{ClosR} & \quad \Gamma \triangleright s_1 \in \text{Subst}(S) \quad \Gamma \triangleright \text{clos} M s_2 \in B \\
& \quad \Gamma \triangleright \text{clos}(\lambda x \cdot M \pi_2 x(s_1 \cdot s_2)) \sim \text{clos} M s_2 \in B \\
\text{ClosPi} & \quad \Gamma \triangleright \text{pi} \text{(clos} A s)(\lambda y \cdot \text{clos}(\lambda x \cdot B x y) s) \in C \\
& \quad \Gamma \triangleright \text{clos}(\lambda x \cdot \text{pi}(A x)(B x)) s \sim \text{pi} \text{(clos} A s)(\lambda y \cdot \text{clos}(\lambda x \cdot B x y) s) \in C \\
\text{ClosAbs} & \quad \Gamma \triangleright \text{abs} \text{(clos} A s)(\lambda y \cdot \text{clos}(\lambda x \cdot M x y) s) \in C \\
& \quad \Gamma \triangleright \text{clos}(\lambda x \cdot \text{abs} A s)(\lambda y \cdot \text{clos}(\lambda x \cdot M x y) s) \sim \text{abs} \text{(clos} A s)(\lambda y \cdot \text{clos}(\lambda x \cdot M x y) s) \in C \\
\text{ClosApp} & \quad \Gamma \triangleright \text{apply} \text{(clos} M s)(\text{clos} N s) \in C \\
& \quad \Gamma \triangleright \text{clos}(\lambda x \cdot \text{apply}(M x)(N x)) s \sim \text{apply} \text{(clos} M s)(\text{clos} N s) \in C \\
\text{ClosSubst} & \quad \Gamma \triangleright \text{clos}(\lambda x \cdot M \pi_1 x \pi_2 x(s_1 \cdot s_2) \map s_1 s_2) \in C \\
& \quad \Gamma \triangleright \text{clos}(\lambda x \cdot \text{clos}(M x(s_1 x))(s_2 \map s_1 s_2)) \sim \text{clos}(\lambda x \cdot M \pi_1 x \pi_2 x(s_2 \map s_1 s_2)) \in C
\end{align*}
\]

B.4 Equivalence Rules for Substitutions

\[
\begin{align*}
\text{MapCong} & \quad \Gamma \triangleright s_2 \sim s_2' \in \text{Subst}(S_2) \quad \Gamma \triangleright \text{map} s_1 s_2' \in \text{Subst}(S_1) \\
& \quad \Gamma \triangleright \text{map} s_1 s_2 \sim \text{map} s_1 s_2' \in \text{Subst}(S_2) \\
\text{SubstTerm} & \quad \Gamma \triangleright A \sim B \in \text{type} \quad \Gamma \triangleright M \sim N \in A \\
& \quad \Gamma \triangleright [M, A] \sim [N, B] \in \text{Subst Term} \\
\text{SubstComp} & \quad \Gamma \triangleright s_1 \sim t_1 \in \text{Subst}(S_1) \quad \Gamma \triangleright s_2 \sim t_2 \in \text{Subst}(S_2) \\
& \quad \Gamma \triangleright s_1 \cdot s_2 \sim t_1 \cdot t_2 \in \text{Subst}(S_1 \times S_2)
\end{align*}
\]
B.5. Type-Checking Algorithm for $S^H_{th}$

\begin{align*}
\text{VAR} & \\
\Gamma, x : A, & \quad \Gamma \triangleright x \in A \\
\text{TYPE} & \\
\Gamma & \triangleright \text{type} \in \text{type} \\
\Pi & \\
\Gamma & \triangleright A' \rightarrow \text{type} \quad \Gamma, x : A \triangleright B' x \rightarrow \text{type} \quad \Gamma \triangleright \text{(pi } A B \text{)} \in \text{type} \\
\text{ABS} & \\
\Gamma & \triangleright A \in A' \quad \Gamma \triangleright A' \rightarrow \text{type} \quad \Gamma, x : A \triangleright M x \in B x \quad \Gamma \triangleright \text{(abs } A M \text{)} \in \text{(pi } A B \text{)} \\
\text{APP} & \\
\Gamma & \triangleright M \in B' \quad \Gamma \triangleright B' \rightarrow \text{(pi } A B \text{)} \quad \Gamma \triangleright N \in A' \quad \Gamma \triangleright A' \rightarrow A \quad \Gamma \triangleright \text{(apply } M N \text{)} \in \text{(clo } \text{[B}[N, A']]) \\
\text{CLOS} & \\
\Gamma & \triangleright s \in \text{Subst}(S) \quad \Gamma \triangleright M \in A \\
\Gamma & \triangleright \text{clo}(\lambda x : M) s \in A \\
\text{CLOSVAR} & \\
\Gamma & \triangleright s \rightarrow [M, A] \quad \Gamma \triangleright M \in A \\
\Gamma & \triangleright \text{clo}(\lambda x : x) s \in A \\
\text{CLOS} & \\
\Gamma & \triangleright s \rightarrow s_1 : s_2 \\
\Gamma & \triangleright s_2 \in \text{Subst}(S) \\
\Gamma & \triangleright \text{clo } s_1 : s_2 \in A \\
\Gamma & \triangleright \text{clo}(\lambda x : M) s_1 x \in A \\
\text{CLOS} & \\
\Gamma & \triangleright s \rightarrow s_1 : s_2 \\
\Gamma & \triangleright s_1 \in \text{Subst}(S) \\
\Gamma & \triangleright \text{clo } s_2 \in A \\
\Gamma & \triangleright \text{clo}(\lambda x : M) s_2 x \in A
\end{align*}
\[
\begin{align*}
\text{ClosPi} & \quad \Gamma \triangleright \text{clos } A s \in A' & \quad \Gamma, x : A \triangleright \text{clos}(\lambda y. B y x) s \in B x \\
& \quad \Gamma \triangleright A' \to \text{type} & \quad \Gamma, x : A \triangleright \text{clos}(\lambda y. B y x) \rightarrow \text{type} \\
& \quad \Gamma \triangleright \text{clos}(\lambda y. p(A y B y)) s \in \text{type} \\
\text{ClosAbs} & \quad \Gamma \triangleright \text{clos } A s \in A' & \quad \Gamma \triangleright A' \to \text{type} & \quad \Gamma, x : \text{clos } A s \triangleright \text{clos}(\lambda y. M y x) s \in B x \\
& \quad \Gamma \triangleright \text{clos}(\lambda y. \text{abs}(A y)((M y) s) s \in \text{type } (A B)) \\
\text{ClosApp} & \quad \Gamma \triangleright \text{clos } M s \in B & \quad \Gamma \triangleright B' \to (\pi A B) & \quad \Gamma \triangleright \text{clos } N s \in A' & \quad \Gamma \triangleright A' \to A \\
& \quad \Gamma \triangleright \text{clos}(\lambda x. \text{apply}(M x)(N x)) s \in \text{clos } B(\text{clos } N x, A') \\
\text{ClosSubst} & \quad \Gamma \triangleright \text{clos } \pi_1 x \pi_2 x s \in \text{type } 1 \quad \Gamma \triangleright \text{clos } A s \triangleright \text{clos}(\pi_1 x \pi_2 x) s_2 \in C \\
& \quad \Gamma \triangleright \text{clos } \pi_1 x \pi_2 x s \in \text{type } 1 \quad \Gamma \triangleright \text{clos } A s \triangleright \text{clos}(\pi_1 x \pi_2 x) s_2 \in C \\
& \quad \Gamma \triangleright \text{clos } B s_1 \triangleright \text{clos } M s_2 \triangleright \text{clos } N s_2 \triangleright \text{type } 1 \quad \Gamma \triangleright \text{clos } \pi_1 x \pi_2 x s \in \text{type } 1 \\
& \quad \Gamma \triangleright \text{clos } B s_1 \triangleright \text{clos } M s_2 \triangleright \text{clos } N s_2 \triangleright \text{type } 1 \\
& \quad \Gamma \triangleright \text{clos } B s_1 \triangleright \text{clos } M s_2 \triangleright \text{clos } N s_2 \triangleright \text{type } 1 \\
\end{align*}
\]
\[ \text{RedAbs} \quad \Gamma \vdash (\text{abs } A \ M) \in C \] 
\[ \Gamma \vdash (\text{abs } A \ M) \rightarrow (\text{abs } A \ M) \in C \]

\[ \text{RadAppVar} \quad \Gamma, x : A, \Gamma' \vdash M \rightarrow x. \quad \Gamma \vdash A, \Gamma' \vdash (\text{apply } x \ N) \in C \] 
\[ \Gamma, x : A, \Gamma' \vdash (\text{apply } M \ N) \rightarrow (\text{apply } x \ N) \in C \]

\[ \text{RedAppAbs} \quad \Gamma \vdash M \rightarrow (\text{abs } A \ M'), \quad \Gamma' \vdash N \in \Gamma', \quad \Gamma' \vdash A' \rightarrow A' \vdash (\text{close } M'[N, A]) \rightarrow M' \in C \] 
\[ \Gamma \vdash (\text{apply } M \ N) \rightarrow M' \in C \]

\[ \text{SubstComp} \quad \Gamma \vdash M \rightarrow M' \in C \] 
\[ \Gamma \vdash \text{close } (\lambda x : M) x \rightarrow M' \in C \]

\[ \text{RedClosConst} \quad \Gamma \vdash M \rightarrow M' \in C \] 
\[ \Gamma \vdash (\text{close } \lambda x : M) x \rightarrow M' \in C \]

\[ \text{RedClosVar} \quad \Gamma \vdash x \rightarrow [M, A] \in \text{Subst } \text{Term} \] 
\[ \Gamma \vdash (\text{close } \lambda y : M) x \rightarrow M \in A \]

\[ \text{RedClosL} \quad \Gamma \vdash x \rightarrow x_1 : x_2 \in \text{Subst } S \] 
\[ \Gamma \vdash (\text{close } \lambda x : M) x_1 \rightarrow M' \in C \] 
\[ \Gamma \vdash (\text{close } \lambda x : M \pi_1 x) x_1 \rightarrow M' \in C \]

\[ \text{RedClosR} \quad \Gamma \vdash x \rightarrow x_1 : x_2 \in \text{Subst } S \] 
\[ \Gamma \vdash (\text{close } \lambda x : M) x_2 \rightarrow M' \in C \] 
\[ \Gamma \vdash (\text{close } \lambda x : M \pi_2 x) x_2 \rightarrow M' \in C \]

\[ \text{RedClosPi} \quad \Gamma \vdash (\text{pick } \text{close } A x)(\lambda y : \text{close } (\lambda x : B x y) x)) \in C \] 
\[ \Gamma \vdash (\text{close } \lambda x : (\text{pick } A x)(\lambda y : \text{close } (\lambda x : B x y) x)) \in C \]

\[ \text{RedClosA} \quad \Gamma \vdash (\text{abs } \text{close } A x)(\lambda y : \text{close } (\lambda x : M x y) x)) \in C \] 
\[ \Gamma \vdash (\text{close } \lambda x : (\text{abs } A x)(\lambda y : (\lambda x : (\lambda x : M x y))) x)) \in C \]

\[ \text{RedClosR} \quad \Gamma \vdash (\text{apply } \text{close } M s)(\text{close } N s) \in C \] 
\[ \Gamma \vdash (\text{close } \lambda x : \text{apply } (M x)(N x)) x \rightarrow (\text{apply } \text{close } M s)(\text{close } N s) \in C \]

\[ \text{RedClosComp} \quad \Gamma \vdash (\text{close } \lambda x : M \pi_1 x \pi_2 x)(x_2 : \text{map } s_1 s_2)) \rightarrow M' \in C \] 
\[ \Gamma \vdash (\text{close } \lambda x : (\text{close } M s)(s_1 x x_2)) \rightarrow M' \in C \]

**B.8. Type Inference for Substitutions**

\[ \text{SubstTerm} \quad \Gamma \vdash A \in A' \] 
\[ \Gamma \vdash A \rightarrow \text{type } \Gamma \vdash M \in B \] 
\[ \Gamma \vdash [M, A] \in \text{Subst } \text{Term} \]

\[ \text{SubstComp} \quad \Gamma \vdash s_1 \in \text{Subst } S_1 \] 
\[ \Gamma \vdash s_2 \in \text{Subst } S_2 \] 
\[ \Gamma \vdash s_1 / s_2 \in \text{Subst } S_1 \times S_2 \]

\[ \text{MapTerm} \quad \Gamma \vdash \text{close } A x \in A' \] 
\[ \Gamma \vdash A' \rightarrow \text{type } \Gamma \vdash \text{close } M s \in B \] 
\[ \Gamma \vdash \text{close } A x \rightarrow B \] 
\[ \Gamma \vdash (\lambda x : \text{map } [M x, A x]) x \in \text{Subst } \text{Term} \]

\[ \text{MapComp} \quad \Gamma \vdash s_1 x \in \text{Subst } S_1 \] 
\[ \Gamma \vdash s_2 x \in \text{Subst } S_2 \] 
\[ \Gamma \vdash (\lambda x : \text{map } s_1 x)(s_2 x) x \in \text{Subst } S_1 \times S_2 \]

\[ \text{MapSubst} \quad \Gamma \vdash (\lambda x : \text{map } s_1 x)(s_2 x) x \in \text{Subst } S \] 
\[ \Gamma \vdash (\lambda x : \text{map } s_1 x)(s_2 x) x \in \text{Subst } S \]
B.9. WHNF Reduction Algorithm for Substitutions

\[
\text{RedMapTerm} \quad \frac{\Gamma \triangleright [(\text{clos } M \, s), (\text{clos } A \, s)] \in \text{Subst Term}}{\frac{T \triangleright (\text{map} \lambda x \cdot [M \, x, A \, x]) \, s \leadsto [(\text{clos } M \, s), (\text{clos } A \, s)] \in \text{Subst Term}}}{T \triangleright [(\text{map} \lambda x \cdot [M \, x, A \, x]) \, s \leadsto [(\text{clos } M \, s), (\text{clos } A \, s)] \in \text{Subst Term}}}
\]

\[
\text{RedMapComp} \quad \frac{\Gamma \triangleright ((\text{map} \lambda x \cdot s_1 \, s_2)) \in \text{Subst S}}{\frac{T \triangleright (\text{map} \lambda x \cdot (s_1 \, x)) \, (s_2 \, x) \leadsto ((\text{map} \lambda x \cdot s_1 \, s_2)) \in \text{Subst S}}}{T \triangleright ((\text{map} \lambda x \cdot (s_1 \, x)) \, (s_2 \, x) \leadsto ((\text{map} \lambda x \cdot s_1 \, s_2)) \in \text{Subst S}}}
\]

\[
\text{RedMapCompose} \quad \frac{\Gamma \triangleright (\text{map} \lambda x \cdot \text{map} \lambda x \cdot s_1 \, s_2) \in \text{C}}{\frac{T \triangleright \text{map} \lambda x \cdot (\text{map} \lambda x \cdot s_1 \, s_2) \leadsto s' \in C}{\frac{T \triangleright \text{map} \lambda x \cdot (\text{map} \lambda x \cdot s_1 \, s_2) \leadsto s' \in C}}}
\]

Received February 2, 1995; final manuscript received December 12, 1997

REFERENCES


Church, A. (1940), A formulation of the simple theory of types, J. Symb. Logic 5, 56–68.


Leroy, X. (1990), The ZINC experiment: An economical implementation of the ML language, Technical Report 117, INRIA.


