Linear Higher-Order Pre-Unification

Iliano Cervesato and Frank Pfenning*
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
{*iliano*|*fp*}*@cs.cmu.edu*

Abstract

We develop an efficient representation and a pre-unification algorithm in the style of Huet for the linear λ-calculus $\lambda^{\rightarrow\multimap\&\top}$ which includes intuitionistic functions ($\rightarrow$), linear functions ($\multimap$), additive pairing ($\&$), and additive unit ($\top$). Applications lie in proof search, logic programming, and logical frameworks based on linear type theories. We also show that, surprisingly, a similar pre-unification algorithm does not exist for certain sublanguages.

1 Introduction

Linear logic [10] enriches more traditional logical formalisms with a notion of consumable resource, which provides direct means for expressing and reasoning about mutable state. Attempts at mechanizing this additional expressive power led to the design of several logic programming languages based on various fragments of linear logic. The only new aspect in the operational semantics of most proposals, such as *Lolli* [15], *Lygon* [12] and *Forum* [22], concerns the management of linear context formulas [3]. In particular, the instantiation of logical variables relies on the traditional unification algorithms, in their first- or higher-order variants, depending on the language. More recent proposals, such as the language of the linear logical framework *LLF* [2, 4] and the system *RLF* [17], introduce linearity not only at the level of formulas, but also within terms. Consequently, implementations of these languages must solve higher-order equations on linear terms in order to instantiate existential variables. In this paper we present a complete algorithm for pre-unification in a linear λ-calculus which conservatively extends the ordinary simply-typed λ-calculus and could be used directly for the above languages.

An example will shed some light on the novel issues brought in by linearity. A rewrite rule $r : t_1 \Longrightarrow t_2$ is applicable to a term t if there is an instance of t_1 in t; then, applying r has the effect of replacing it with t_2 (assume t_1 and t_2 ground, for simplicity). This is often formalized by writing $t = \tilde{t}[t_1]$, where the *rewriting context* $\tilde{t}$ is a term containing a unique occurrence of a *hole* ($[_]$) so that replacing the hole with t_1 yields t. We can then express r as the parametric transition $T[t_1] \Longrightarrow T[t_2]$, where T is a variable standing for a rewriting context. The applicability of r to a term t reduces to the problem of whether t and the higher-order expression $(T\ t_1)$, where T is viewed as a functional variable, are unifiable. Traditional higher-order unification does not take into consideration the *linearity constraint* that exactly one occurrence of t_1 must be abstracted away from t. Indeed, matching $(T\ t_1)$ with $(c\ t_1\ t_1)$ has four solutions:

$$\boxed{\begin{array}{l} T \longleftarrow \lambda x.\, c\ t_1\ t_1 \\ T \longleftarrow \lambda x.\, c\ x\ x \end{array}} \qquad \begin{array}{l} T \longleftarrow \lambda x.\, c\ x\ t_1 \\ T \longleftarrow \lambda x.\, c\ t_1\ x \end{array}$$

But the first match in the box does not have any hole (the variable x) in it while the second contains two. Linear unification, on the other hand, returns correctly only the two unboxed solutions. This means also that the natural encoding of a rewrite system based on rewriting contexts in the logical framework in *LF* is unsound, while it would be adequately represented in *LLF*.

The introduction of linear term languages in *LLF* and *RLF* has been motivated by a number of applications. Linear terms provide a statically checkable notation for natural deductions [17] or sequent derivations [4] in substructural logics. In the realm of programming languages, linear terms naturally model *computations* in imperative languages [4] or sequences

* This work was supported by NSF Grant CCR-9303383. The second author was supported by the *Alexander-von-Humboldt-Stiftung* when working on this paper, during a visit to the Department of Mathematics of the Technical University Darmstadt.

This extended abstract has been accepted at the *Twelfth Annual Symposium on Logic in Computer Science - LICS'97*, Warsaw, Poland, June 29th – July 2nd 1997.

Types:	$A ::= a$	*Terms:*	$M ::= c \mid x$		
	$\mid A_1 \to A_2$		$\mid \lambda x{:}A.\, M \mid M_1\, M_2$		(*intuitionistic functions*)
	$\mid A_1 \multimap A_2$		$\mid \hat{\lambda} x{:}A.\, M \mid M_1 \hat{} M_2$		(*linear functions*)
	$\mid A_1 \,\&\, A_2$		$\mid \langle M_1, M_2 \rangle \mid \textsc{fst}\ M \mid \textsc{snd}\ M$		(*additive pairs*)
	$\mid \top$		$\mid \langle\rangle$		(*additive unit*)
Signatures:	$\Sigma ::= \cdot \mid \Sigma, c : A$				
Contexts:	$\Gamma ::= \cdot \mid \Gamma, x : A$				

Figure 1. Syntax of $\lambda^{\to \multimap \& \top}$

of moves in games [2]. When we want to specify, manipulate, or reason about such objects (which is common in logic and the theory of programming languages), then internal linearity constraints are critical in practice (see, for example, the first formalizations of cut-elimination in linear logic and type preservation for *Mini-ML* with references [4]).

Differently from the first-order case, higher-order unification in Church's simply typed λ-calculus $\lambda^{\to}$ is undecidable and does not admit most general unifiers [11]. Nevertheless sound and complete (although possibly non-terminating) procedures have been proposed in order to enumerate solutions [18]. In particular, Huet's pre-unification algorithm [16] computes unifiers in a non-redundant manner as constraints and has therefore been adopted in the implementation of higher-order logic programming languages [23]. Fragments of $\lambda^{\to}$ of practical relevance for which unification is decidable and yields most general unifiers have also been discovered. An example are Miller's higher-order patterns [21], that have been implemented in the higher-order constraint logic programming language *Elf* [25]. Unification in the context of linear λ-calculi has received limited attention in the literature and, to our knowledge, only a restricted fragment of a multiplicative language has been treated [19]. Unification in $\lambda^{\to}$ with linear restrictions on existential variables has been studied in [26].

In this extended abstract, we investigate the unification problem in the linear simply-typed λ-calculus $\lambda^{\to \multimap \& \top}$. We give a pre-unification procedure in the style of Huet and discuss the new sources of non-determinism due to linearity. Moreover, we show that no such algorithm can be devised for linear sublanguages deprived of $\top$ and of the corresponding constructor. $\lambda^{\to \multimap \& \top}$ corresponds, via a natural extension of the Curry-Howard isomorphism, to the fragment of intuitionistic linear logic freely generated from the connectives $\to$, $\multimap$, & and $\top$, which constitutes the propositional core of *Lolli* [15] and *LLF* [4]. $\lambda^{\to \multimap \& \top}$ is also the simply-typed variant of the term language of *LLF* and shares similarities with the calculus proposed in [1]. Its theoretical relevance derives from the fact that it is the largest linear λ-calculus that admits unique long $\beta\eta$-normal forms.

The principal contributions of this work are: (1) a first solution to the problem of linear higher-order unification, currently a major obstacle to the implementation of logical frameworks and logic programming languages relying on a linear higher-order term language; (2) the definition of a new representation technique for generic λ-calculi that permits both simple meta-reasoning and efficient implementations; (3) the elegant and precise presentation of an extension of Huet's pre-unification procedure as a system of inference rules. More details on the topics covered in this extended abstract can be found in the technical reports [6] and [5].

Our presentation is organized as follows. In Section 2, we define $\lambda^{\to \multimap \& \top}$ and give an equivalent formulation better suited for our purposes. The pre-unification algorithm is the subject of Section 3. We study the unification problem in sublanguages of $\lambda^{\to \multimap \& \top}$ and hint at the possibility of a practical implementation in Section 4. In order to facilitate our description in the available space, we must assume the reader familiar with traditional higher-order unification [16] and linear logic [10].

2 A Linear Simply-Typed λ-Calculus

This section defines the simply-typed linear λ-calculus $\lambda^{\to \multimap \& \top}$ (Section 2.1) and presents an equivalent formulation, $S^{\to \multimap \& \top}$ (Section 2.2), which is more convenient for describing and implementing unification.

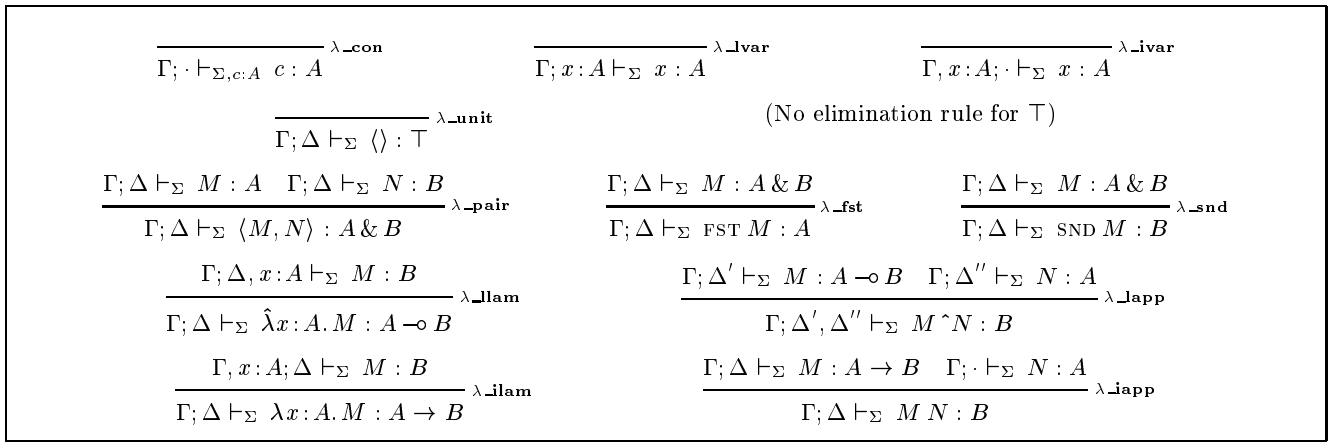

Figure 2. Typing in $\lambda^{\to\multimap\&\top}$

2.1 Basic Formulation

The linear simply-typed λ-calculus $\lambda^{\to\multimap\&\top}$ extends Church's $\lambda^{\to}$ with the three type constructors $\multimap$ (*multiplicative arrow*), & (*additive product*) and $\top$ (*additive unit*), derived from the identically denoted connectives of linear logic. The language of terms is augmented accordingly with constructors and destructors, devised from the natural deduction style inference rules for these connectives. Although not strictly necessary at this level of the description, the inclusion of intuitionistic constants will be convenient in the development of the discussion. Figure 1 presents the resulting grammar in a tabular format that relate each type constructor (left) to the corresponding term operators (center), with constructors preceding destructors. As usual, we rely on signatures and contexts to assign types to constants and free variables, respectively. Here x, c and a range over variables, constants and base types, respectively. In addition to the names displayed above, we will often use N, B and Δ for objects, types and contexts, respectively.

The notions of free and bound variables are adapted from $\lambda^{\to}$. As usual, we identify terms that differ only in the name of their bound variables and write $[M/x]N$ for the capture-avoiding substitution of M for x in the term N. Contexts and signatures are treated as multisets; we promote "," to denote their union and omit writing "$\cdot$" when unnecessary. Finally, we require variables and constants to be declared at most once in a context and in a signature, respectively.

The typing judgment for $\lambda^{\to\multimap\&\top}$ has the form

$$\Gamma;\Delta\vdash_\Sigma\ M:A$$

where Γ and Δ are called the *intuitionistic* and the *linear* context, respectively. The inference rules for this judgment are displayed in Figure 2. Deleting the terms that appear in them results in the usual rules for the $(\to\ \multimap\&\top)$ fragment of intuitionistic linear logic, $ILL^{\to\multimap\&\top}$ [15], in a natural deduction style formulation. $\lambda^{\to\multimap\&\top}$ and $ILL^{\to\multimap\&\top}$ are related by a form of the Curry-Howard isomorphism. Note that the interactions of rules λ**_unit** and λ**_lapp** can flatten distinct proofs to the same $\lambda^{\to\multimap\&\top}$ term.

The reduction semantics of $\lambda^{\to\multimap\&\top}$ is given by the transitive and reflexive closure of the congruence relation built on the following β-reduction rules:

$$\begin{array}{rcl@{\qquad}rcl}\mathrm{FST}\,\langle M,N\rangle & \longrightarrow & M & (\hat{\lambda}x{:}A.\,M)\,\hat{}\,N & \longrightarrow & [N/x]M\\ \mathrm{SND}\,\langle M,N\rangle & \longrightarrow & N & (\lambda x{:}A.\,M)\,N & \longrightarrow & [N/x]M\end{array}$$

Similarly to $\lambda^{\to}$, $\lambda^{\to\multimap\&\top}$ enjoys a number of highly desirable properties [2]. In particular, since every extension (for example with $\otimes$ and multiplicative pairs) introduces commutative conversions, it is the largest linear λ-calculus for which strong normalization holds and yields unique normal forms. We write $\mathrm{Can}(M)$ for the *canonical form* of the term M, defined as the η-expansion of its β-normal form. For reasons of efficiency, we will often refer to the *weak head-normal form* of a term M, written $\overline{M}$, that differs from $\mathrm{Can}(M)$ by the possible presence of redices in the arguments of applications. Notice that $\overline{x}$ corresponds to the η-long form of the variable x. In the following, we will insist in dealing always with fully η-expanded terms. We call a term of base type *atomic*.

2.2 The Spine Calculus

Unification algorithms base a number of choices on the nature of the heads of the terms to be unified. The head is immediately available in the first-order case,

and still discernible in $\lambda^{\to}$ since every η-long normal term has the form

$$\lambda x_1 : A_1. \ldots \lambda x_n : A_n. t\, M_1 \ldots M_m$$

where the head t is a constant or a variable and $(t\, M_1 \ldots M_m)$ has base type. The usual parentheses saving conventions hide the fact that t is indeed deeply buried in the sequence of application and therefore not immediately accessible. A similar notational trick fails in $\lambda^{\to\multimap\&\top}$ since on the one hand a non-atomic term can have several heads (e.g. c_1 and c_2 in $\langle c_1, c_2\rangle$), possibly none (e.g. $\langle\rangle$), and on the other hand destructors can be interleaved arbitrarily in an atomic term (e.g. $\textsc{fst}\,((\textsc{snd}\, c)\,\hat{}\, x\, y)$).

The *spine calculus* $S^{\to\multimap\&\top}$ [6] permits recovering both efficient head accesses and notational convenience. Every atomic term M of $\lambda^{\to\multimap\&\top}$ is written in this presentation as a *root* $H \cdot S$, where H corresponds to the head of M and the *spine* S collects the sequence of destructors applied to it. For example, $M = (t\, M_1 \ldots M_m)$ is written $U = t \cdot (U_1; \ldots U_m; \textsc{nil})$ in this language, where ";" represents application, NIL identifies the end of the spine, and U_i is the translation of M_i. Application and ";" have opposite associativity so that M_1 is the innermost subterm of M while U_1 is outermost in the spine of U. This approach was suggested by an empirical study of higher-order logic programs based on $\lambda^{\to}$ terms [20] and is reminiscent of the notion of abstract Böhm trees [14]; its practical merits in our setting are currently assessed in an experimental implementation. The following grammar describes the syntax of $S^{\to\multimap\&\top}$: we write constructors as in $\lambda^{\to\multimap\&\top}$, but use new symbols to distinguish a spine operator from the corresponding term destructor.

$$\begin{array}{llll} \textit{Terms:} & U ::= & H \cdot S & \textit{Spines:}\quad S ::= \textsc{nil} \\ & \mid & \lambda x : A.\, U & \mid\ U; S \\ & \mid & \hat{\lambda} x : A.\, U & \mid\ U \mathbin{\hat{;}} S \\ & \mid & \langle U_1, U_2\rangle & \mid\ \pi_1 S \mid \pi_2 S \\ & \mid & \langle\rangle & \\ \textit{Heads:} & H ::= & c \mid x \mid U & \end{array}$$

We adopt the same syntactic conventions as in $\lambda^{\to\multimap\&\top}$ and often write V for terms in $S^{\to\multimap\&\top}$. Terms are allowed as heads in order to construct β-redices. Indeed, normal terms have either a constant or a variable as their heads. The typing judgments for terms and spines are denoted $\Gamma; \Delta \vdash_\Sigma U : A$ and $\Gamma; \Delta \vdash_\Sigma S : A > B$ respectively, where the latter expresses the fact that given a head H of type A, the root $H \cdot S$ has type B. For reasons of space, we omit the typing rules for these judgments [6], although they will indirectly appear in the inference system for pre-unification.

There exists a structural translation of terms in $\lambda^{\to\multimap\&\top}$ to terms in $S^{\to\multimap\&\top}$, and vice versa [6]. Space constraints do not allow presenting this mapping and the proofs of soundness and completeness for the respective typing derivations. We instead describe it by means of examples by giving the translation of the β-reduction rules of $\lambda^{\to\multimap\&\top}$ into $S^{\to\multimap\&\top}$:

$$\begin{array}{rcl} \langle U, V\rangle \cdot (\pi_1 S) & \longrightarrow_S & U \cdot S \\ \langle U, V\rangle \cdot (\pi_2 S) & \longrightarrow_S & V \cdot S \\ (\hat{\lambda} x : A.\, U) \cdot (V \mathbin{\hat{;}} S) & \longrightarrow_S & [V/x]U \cdot S \\ (\lambda x : A.\, U) \cdot (V; S) & \longrightarrow_S & [V/x]U \cdot S \end{array}$$

The trailing spine in the reductions for $S^{\to\multimap\&\top}$ is a consequence of the fact that this language reverses the nesting order of $\lambda^{\to\multimap\&\top}$ destructors. The structure of roots in the spine calculus makes one more β-reduction rule necessary, namely:

$$(H \cdot S) \cdot \textsc{nil} \quad \longrightarrow_S \quad H \cdot S$$

As for $\lambda^{\to\multimap\&\top}$, we insist on terms being in η-long form. Consequently, roots have always base type and so do the target types in the spine typing judgment. The β-reduction rules above preserve long forms so that η-expansion steps never need to be performed [6]. We write $\mathrm{Can}(U)$ and $\overline{U}$, respectively, for the canonical form and the weak head-normal form of the term U with respect to these reductions.

3 Linear Higher-Order Unification

In this section, we define the unification problem for $S^{\to\multimap\&\top}$ (Section 3.1), show a few examples (Section 3.2), describe a pre-unification algorithm à la Huet for it (Section 3.3), and discuss new sources of nondeterminism introduced by linearity (Section 3.4).

3.1 The Unification Problem

Two $S^{\to\multimap\&\top}$ terms U_1 and U_2 are *equal* if they can be β-reduced to a common term V. By strong normalization and the Church-Rosser theorem [6], it suffices to compute $\mathrm{Can}(U_1)$ and $\mathrm{Can}(U_2)$ and check whether they are syntactically equal (modulo renaming of bound variables). We have the following equality judgments for terms and spines, respectively:

$$\Gamma; \Delta \vdash_\Sigma U_1 = U_2 : A \qquad \Gamma; \Delta \vdash_\Sigma S_1 = S_2 : A > a$$

The types can be omitted altogether if we assume the two objects in every equation we start from to have the same type. We do not show the deduction rules for these judgments. The interested reader can extract them from the non-flexible cases in Figures 4 or consult [5].

Equality checking becomes a *unification problem* as soon as we admit objects containing *logical vari-*

$$
\begin{array}{rrcl}
\textit{Equation systems:} & \Xi & ::= & \cdot \mid \Xi, (\Gamma; \Delta \vdash U_1 = U_2 : A) \mid \Xi, (\Gamma; \Delta \vdash S_1 = S_2 : A > a) \\
\textit{Flex-flex systems:} & \Xi_{ff} & ::= & \cdot \mid \Xi_{ff}, (\Gamma; \Delta \vdash F_1 \cdot S_1 = F_2 \cdot S_2 : a) \\
\textit{Substitutions:} & \Theta & ::= & \cdot \mid \Theta, U/F \\
\textit{Pools:} & \Phi & ::= & \cdot \mid \Phi, F : A
\end{array}
$$

Figure 3. Syntax of Equations

ables (sometimes called *existential variables* or *meta-variables*), standing for unknown terms. The equalities above, called *equations* in this setting, are *unifiable* if there exists a substitution for the logical variables which makes the two sides equal. These substitutions are called *unifiers*. The task of a *unification procedure* is to determine whether equations are solvable and, if so, report their unifiers. As for $\lambda^{\rightarrow}$, it is undecidable whether two $S^{\rightarrow -\circ \& \top}$ terms can be unified, since its equational theory is a conservative extension of the equational theory for the simply-typed λ-calculus.

Logical variables stand for heads and cannot replace spines or generic terms. Therefore, the alterations to the definition of $S^{\rightarrow -\circ \& \top}$ required for unification are limited to enriching the syntax of heads with logical variables, that we denote F, G and H possibly subscripted. We continue to write U, V and S for terms and spines in this extended language. In order to avoid confusion we will call the proper variables of $S^{\rightarrow -\circ \& \top}$ *parameters* in the remainder of the paper.

The machinery required in order to state a unification problem is summarized in Figure 3. We will in general solve *systems of equations* that share the same signature and a common set of logical variables. A *solution* to a unification problem is a substitution that, when applied to it, yields a *system of flex-flex equations* that is known to be solvable. This notion subsumes unifiers as a particular case. Finally, we record the types of the logical variables in use in a *pool*.

We assume that variables appear at most once in a pool and in the domain of a substitution. Similarly to contexts, we treat equation systems and pools as multisets. We write ξ for individual equations. The context $\Gamma; \Delta$ in an equation ξ enumerates the parameters that the substitutions for logical variables appearing in ξ are not allowed to mention directly. Therefore, legal substitution terms U for a variable $F : A$ must be typable in the empty context, i.e. $\cdot; \cdot \vdash_{\Sigma, \Phi} U : A$ should be derivable where Φ includes the logical variables appearing in U. This is sometimes emphasized by denoting an equation system Ξ as $\forall \Sigma. \exists \Phi. \forall (\Xi)$, where the inner expression means that the context $\Gamma; \Delta$ of every equation ξ is universally quantified in front of it. A term or spine equation ξ can be interpreted as an equality judgment with signature Σ, Φ, where again Φ includes the logical variables appearing in ξ. In the following, we will occasionally view an equation system Ξ as the multiset of the equality judgments corresponding to its equations. Finally, we write dom Θ for the domain of the substitution Θ and $\Theta \circ \Theta'$ for the composition of substitutions Θ and Θ', defined as usual.

3.2 Examples

The example given in the introduction clearly shows how linearity restricts the set of solutions found by traditional higher-order unification in the absence of linear constructs. We can indeed rewrite this example in the syntax of $\lambda^{\rightarrow -\circ \& \top}$ (chosen over $S^{\rightarrow -\circ \& \top}$ for the sake of clarity) as the following equation

$$\cdot; \cdot \vdash F \hat{} M = c \hat{} M \hat{} M : a.$$

As we saw, only two of the four independent solutions returned by traditional higher-order unification on the corresponding $\lambda^{\rightarrow}$ problem are linearly valid.

More complex situations rule out the simple minded strategy of keeping only the linearly valid solutions returned by a traditional unification procedure on a linear problem. Consider the following equation, written again in the syntax of $\lambda^{\rightarrow -\circ \& \top}$ for simplicity,

$$x : A, y : B; \cdot \vdash F \hat{} x \hat{} y = c \hat{} (G_1\ x\ y) \hat{} (G_2\ x\ y) : a.$$

The parameters x and y are intuitionistic, but F uses them as linear objects. We must instantiate F to a term of the form $\hat{\lambda} x' : A. \hat{\lambda} y' : B. c \hat{} M_1 \hat{} M_2$ where each of the *linear* parameters x' and y' must appear either in M_1 or in M_2, but not in both. Indeed, the following four incomparable substitutions are generated:

$$
\begin{array}{ll}
F \longleftarrow \hat{\lambda} x' : A. \hat{\lambda} y' : B. c \hat{} (F_1 \hat{} x' \hat{} y') \hat{} F_2 \\
F \longleftarrow \hat{\lambda} x' : A. \hat{\lambda} y' : B. c \hat{} (F_1 \hat{} x') \hat{} (F_2 \hat{} y') \\
F \longleftarrow \hat{\lambda} x' : A. \hat{\lambda} y' : B. c \hat{} (F_1 \hat{} y') \hat{} (F_2 \hat{} x') \\
F \longleftarrow \hat{\lambda} x' : A. \hat{\lambda} y' : B. c \hat{} F_1 \hat{} (F_2 \hat{} x' \hat{} y')
\end{array}
$$

Traditional unification on the analogous $\lambda^{\rightarrow}$ equation is unitary and would return the single substitution

$$F \longleftarrow \lambda x' : A. \lambda y' : B. c\ (F_1\ x'\ y')\ (F_2\ x'\ y').$$

which is *not* linearly valid. This example also illustrates one reason why linear term languages and unification are useful. Linearity constraints rule out certain unifiers when compared to the simply-typed formulation of the same expression, which can be used to eliminate ill-formed terms early.

3.3 A Pre-Unification Algorithm

Our adaptation of Huet's pre-unification procedure to $S^{\rightarrow\multimap\&\top}$ is summarized in Figures 4–6. We adopt a structured operational semantics presentation as a system of inference rules, which isolates and makes every step of the algorithm explicit. Although more verbose than the usual formulations, it is, at least in this setting, more understandable and closer to an actual implementation. In this subsection, we describe the general structure of the algorithm. We will discuss the specific aspects brought in by linearity in the Section 3.4.

On the basis of the above definitions, a unification problem is expressed by the following judgment:

$$\Xi \setminus \Xi_{ff}, \Theta$$

where, for the sake of readability, we keep the signature Σ and the current variable pool Φ implicit. The procedure we describe accepts Σ, Φ and Ξ as input arguments and attempts to construct a derivation $\mathcal{X}$ of $\Xi \setminus \Xi_{ff}, \Theta$ for some Θ and Ξ_{ff}. This could terminate successfully (in which case Θ is a unifier if Ξ_{ff} is empty, and only a pre-unifier otherwise). It might also fail (in which case there are no unifiers) or not terminate (in which case we have no information).

Given a system of weak head-normal equations Ξ to be solved with respect to a signature Σ and a logical variables pool Φ, the procedure selects an equation ξ from Ξ and attempts to apply in a bottom up fashion one of the rules in Figure 4. If several rules are applicable, the procedure succeeds if one of them yields a solution. If none applies, we have a local failure. The procedure terminates when all equations in Ξ are flex-flex, as described below.

Well-typed equations in weak head-normal form have a very disciplined structure. In particular, both sides must either be roots, or have the same top-most term or spine constructor. Spine equations and non-atomic term equations are therefore decomposed until problems of base type are produced, as shown in the uppermost and lowermost parts of Figure 4, respectively.

Following the standard terminology, we call an atomic term $H \cdot S$ *rigid* if H is a constant or a parameter, and *flexible* if it is a logical variable. Since the sides of a canonical equation ξ of base type can be only either rigid or flexible, we have four possibilities:

Rigid-Rigid If the head of both sides of ξ is the same constant or parameter, we unify the spines.

Rigid-Flex We reduce this case to the next by swapping the sides of the equation.

Flex-Rigid Consider first the equation $\Gamma; \Delta \vdash F \cdot S_1 = c \cdot S_2 : a$ where the head c is a constant. Solving this equation requires instantiating F to a term V that, relatively to spine S_1, has c as its head; the resulting spines are then unified as in the rigid-rigid case. We can construct V in two manners: the first, *imitation*, puts c in the proper position in V and rearranges the terms appearing in S_1 in order to match the type of c. The second, *projection*, looks for some subterm that might be instantiated, via β-reduction, to c inside S_1 and installs it as the appropriate head of V, again reshuffling S_1 to match its type. Once V has been produced, it is substituted for every occurrence of F in the equation system and weak head-reduction is performed. The pair V/F is added to the current substitution. Flex-rigid equations with a parameter as their rigid head are treated similarly except that imitation cannot be applied since parameters are bound within the scope of logical variables.

Given an equation $\Gamma; \Delta \vdash F \cdot S_1 = c \cdot S_2 : a$, the construction of the instantiating term V in the case of imitation is described in the upper part of Figure 5. The judgment

$$\Gamma'; \Delta' \vdash c_B \cdot S_2 \,/\, A \Uparrow^{\iota} S_1 \hookrightarrow V$$

builds the constructors layer of V on the basis of the type A of F and of the spine S_2. Here, B is the type of c. As V is constructed, the local parameters bound by linear and intuitionistic λ-abstraction (rules **fri_llam** and **fri_ilam**) are stored in the accumulators Γ' and Δ' respectively. When A has the form $A_1 \,\&\, A_2$ (rules **fri_pair1** and **fri_pair2**), V must be a pair $\langle V_1, V_2 \rangle$ and S_2 must start with a projection. The subterm V_i that is projected away can be arbitrary as long as it has type A_i and uses up all local parameters in $\Gamma'; \Delta'$; this is achieved by means of the variable raising judgment discussed below. When a base type is eventually reached (rule **fri_con**), the right-hand side $c \cdot S_2$ of the original equation is accessed, the constant c is installed as the head of V and its spine S is constructed by reshuffling the arguments present in S_2 and inserting the local parameters accumulated in $\Gamma'; \Delta'$. The spine S is built by the judgment

Term traversal

$$\frac{\Xi \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta \vdash \langle\rangle = \langle\rangle : \top) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_unit} \qquad \frac{\Xi, (\Gamma; \Delta \vdash U_1 = V_1 : A), (\Gamma; \Delta \vdash U_2 = V_2 : B) \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta \vdash \langle U_1, U_2\rangle = \langle V_1, V_2\rangle : A \,\&\, B) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_pair}$$

$$\frac{\Xi, (\Gamma; \Delta, x{:}A \vdash U = V : B) \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta \vdash \hat{\lambda} x{:}A.\, U = \hat{\lambda} x{:}A.\, V : A \multimap B) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_llam} \qquad \frac{\Xi, (\Gamma, x{:}A; \Delta \vdash U = V : B) \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta \vdash \lambda x{:}A.\, U = \lambda x{:}A.\, V : A \to B) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_ilam}$$

Rigid−Rigid

$$\frac{c{:}A \ in\ \Sigma \quad \Xi, (\Gamma; \Delta \vdash S_1 = S_2 : A > a) \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta \vdash c \cdot S_1 = c \cdot S_2 : a) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_rr_con}$$

$$\frac{\Xi, (\Gamma; \Delta \vdash S_1 = S_2 : A > a) \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta, x{:}A \vdash x \cdot S_1 = x \cdot S_2 : a) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_rr_lvar} \qquad \frac{\Xi, (\Gamma, x{:}A; \Delta \vdash S_1 = S_2 : A > a) \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma, x{:}A; \Delta \vdash x \cdot S_1 = x \cdot S_2 : a) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_rr_ivar}$$

Rigid−Flex

$$\frac{\Xi, (\Gamma; \Delta \vdash F \cdot S_2 = c \cdot S_1 : a) \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta \vdash c \cdot S_1 = F \cdot S_2 : a) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_rf_con} \qquad \frac{\Xi, (\Gamma; \Delta \vdash F \cdot S_2 = x \cdot S_1 : a) \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta \vdash x \cdot S_1 = F \cdot S_2 : a) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_rf_var}$$

Flex−Rigid

$$\frac{F{:}A \ in\ \Phi \quad c{:}B \ in\ \Sigma \quad \cdot;\cdot \vdash c_B \cdot S_2 \,/\, A \Uparrow^{\iota} S_1 \hookrightarrow V \quad \overline{[V/F](\Xi, (\Gamma; \Delta \vdash F \cdot S_1 = c \cdot S_2 : a))} \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta \vdash F \cdot S_1 = c \cdot S_2 : a) \setminus \Xi_{ff}, (\Theta, V/F)}\ \textbf{pu_fr_con_imit}$$

$$\frac{F{:}A \ in\ \Phi \quad c{:}B \ in\ \Sigma \quad \cdot;\cdot \vdash A \Uparrow^{\pi} S_1 \hookrightarrow V \quad \overline{[V/F](\Xi, (\Gamma; \Delta \vdash F \cdot S_1 = c \cdot S_2 : a))} \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta \vdash F \cdot S_1 = c \cdot S_2 : a) \setminus \Xi_{ff}, (\Theta, V/F)}\ \textbf{pu_fr_con_proj}$$

$$\frac{F{:}A \ in\ \Phi \quad \cdot;\cdot \vdash A \Uparrow^{\pi} S_1 \hookrightarrow V \quad \overline{[V/F](\Xi, (\Gamma; \Delta, x{:}B \vdash F \cdot S_1 = x \cdot S_2 : a))} \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta, x{:}B \vdash F \cdot S_1 = x \cdot S_2 : a) \setminus \Xi_{ff}, (\Theta, V/F)}\ \textbf{pu_fr_lvar_proj}$$

$$\frac{F{:}A \ in\ \Phi \quad \cdot;\cdot \vdash A \Uparrow^{\pi} S_1 \hookrightarrow V \quad \overline{[V/F](\Xi, (\Gamma, x{:}B; \Delta \vdash F \cdot S_1 = x \cdot S_2 : a))} \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma, x{:}B; \Delta \vdash F \cdot S_1 = x \cdot S_2 : a) \setminus \Xi_{ff}, (\Theta, V/F)}\ \textbf{pu_fr_ivar_proj}$$

Flex−Flex

$$\frac{}{\Xi_{ff} \setminus \Xi_{ff}, \cdot}\ \textbf{pu_ff}$$

Spine traversal

$$\frac{\Xi \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \cdot \vdash \text{NIL} = \text{NIL} : a > a) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_nil}$$

$$\frac{\Xi, (\Gamma; \Delta \vdash S_1 = S_2 : A_1 > a) \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta \vdash \pi_1 S_1 = \pi_1 S_2 : A_1 \,\&\, A_2 > a) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_fst} \qquad \frac{\Xi, (\Gamma; \Delta \vdash S_1 = S_2 : A_2 > a) \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta \vdash \pi_2 S_1 = \pi_2 S_2 : A_1 \,\&\, A_2 > a) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_snd}$$

$$\frac{\Xi, (\Gamma; \Delta' \vdash \overline{U_1} = \overline{U_2} : A_1), (\Gamma; \Delta'' \vdash S_1 = S_2 : A_2 > a) \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta', \Delta'' \vdash U_1 \,\hat{;}\, S_1 = U_2 \,\hat{;}\, S_2 : A_1 \multimap A_2 > a) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_lapp}$$

$$\frac{\Xi, (\Gamma; \cdot \vdash \overline{U_1} = \overline{U_2} : A_1), (\Gamma; \Delta \vdash S_1 = S_2 : A_2 > a) \setminus \Xi_{ff}, \Theta}{\Xi, (\Gamma; \Delta \vdash U_1; S_1 = U_2; S_2 : A_1 \to A_2 > a) \setminus \Xi_{ff}, \Theta}\ \textbf{pu_iapp}$$

Figure 4. Pre-Unification in $S^{\to \multimap \& \top}$, Equation Manipulation

$$\Gamma'; \Delta' \vdash B \downarrow^{\iota} S_2 \hookrightarrow S$$

by mimicking the structure of S_2. Notice the use of the variable raising judgment in rules **fri_lapp** and **fri_iapp** to construct appropriate η-long arguments with new logical variables as heads applied to the parameters in $\Gamma'; \Delta'$.

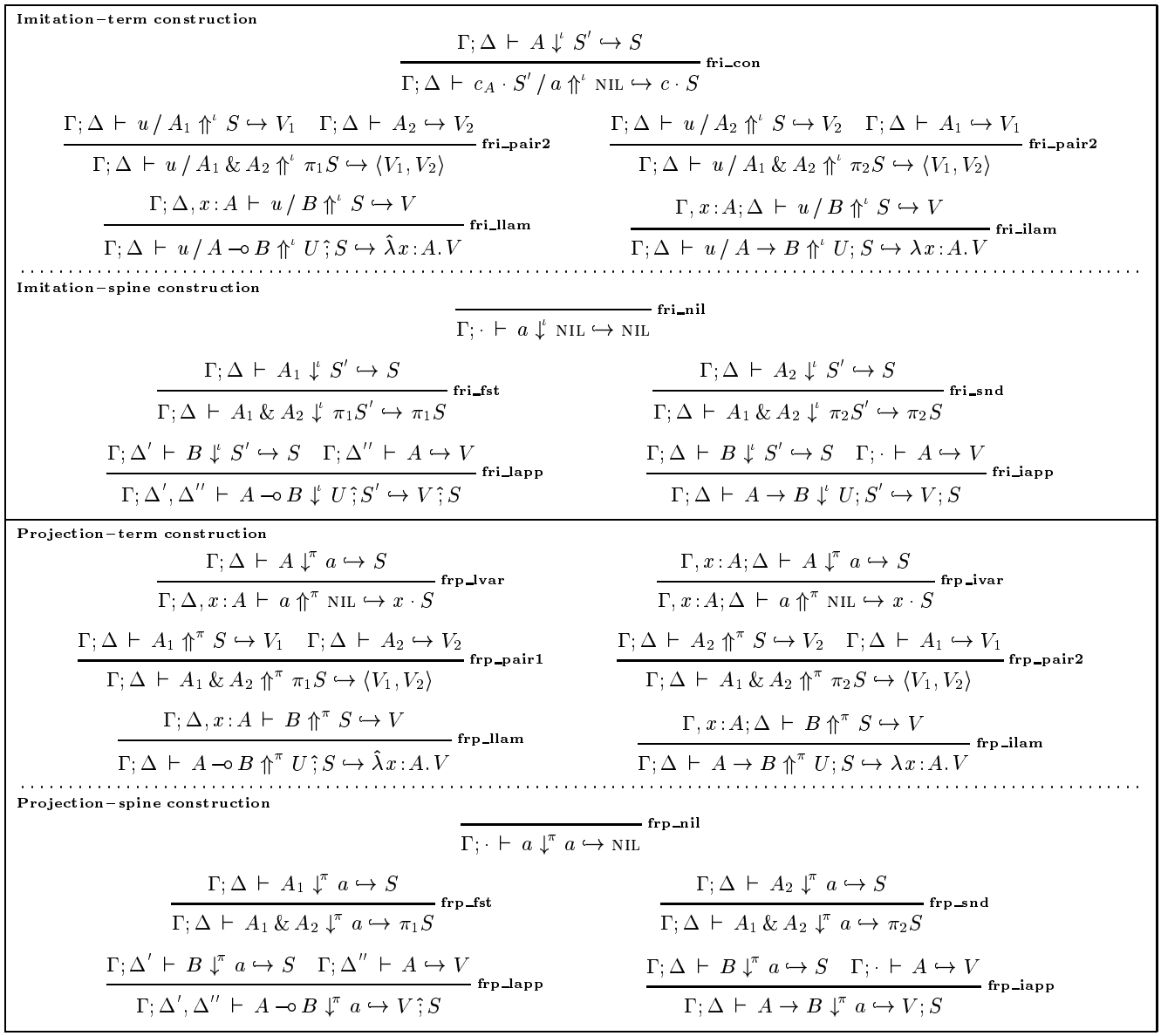

Figure 5. Pre-Unification in $S^{\to \multimap \& \top}$, Generation of Substitutions

The construction of V in the case of projection, displayed in the lower part of Figure 5, is similar except that its spine S is built on the basis of the type A of the projected parameter (rules **frp_lvar** and **frp_ivar**). This leads to a form of non-determinism for product types not present in the case of imitation (rules **frp_fst** and **frp_snd**).

The purpose of the variable raising judgment

$$\Gamma';\Delta' \vdash A \hookrightarrow V,$$

displayed in Figure 6, is to produce an η-long term V of type A with new logical variables as its heads (rule **raise_root**) and the parameters accumulated in $\Gamma';\Delta'$ in the corresponding spines. Notice that functional types yield new local parameters (rules **raise_llam** and **raise_ilam**). The spines themselves are constructed by means of the judgment

$$\Gamma';\Delta' \vdash a \hookrightarrow S, A$$

which builds a spine S mapping heads of type A to roots of type a by rearranging non-deterministically the parameters present in $\Gamma';\Delta'$.

Flex-Flex Similarly to $\lambda^{\to}$, a system composed uniquely of flex-flex equations is always solvable

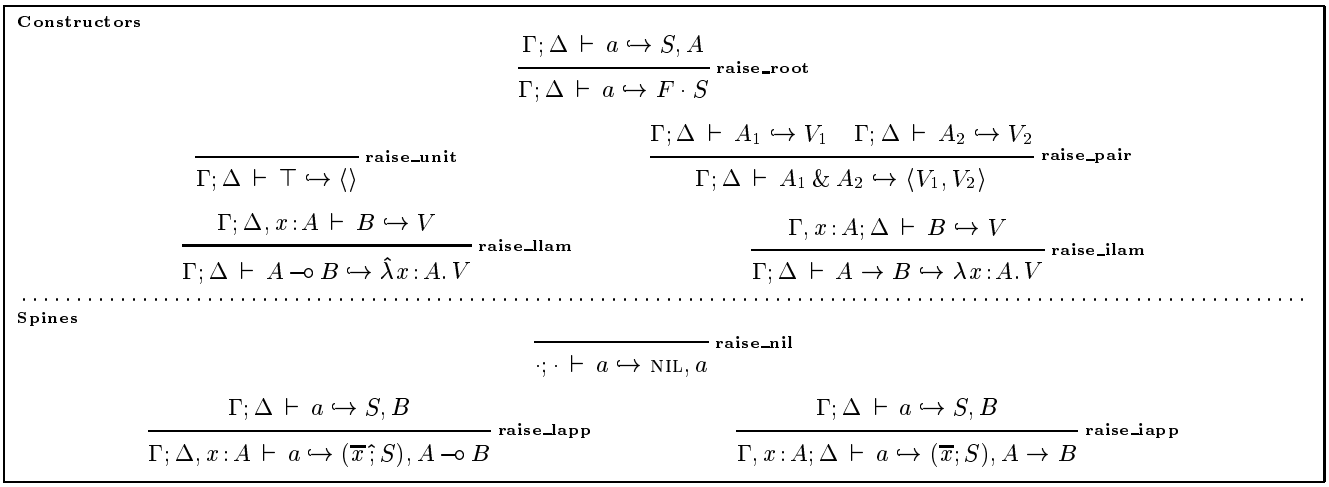

Figure 6. Pre-Unification in $S^{\to\multimap\&\top}$, Raising Variables

in $S^{\to\multimap\&\top}$. Indeed, every logical variable F in it can be instantiated to a term V_F consisting of a layer of constructors as dictated by the type of F, but with every root set to $H_a \cdot \langle\rangle \,\hat{;}\, \mathrm{NIL}$ (i.e. $H_a \,\hat{}\, \langle\rangle$ in $\lambda^{\to\multimap\&\top}$), where H_a is a common new logical variable of type $\top \multimap a$, for each base type a. Then, after normalization, every equation ξ reduces to $\Gamma_\xi;\Delta_\xi \vdash H_a \cdot \langle\rangle \,\hat{;}\, \mathrm{NIL} = H_a \cdot \langle\rangle \,\hat{;}\, \mathrm{NIL} : a$ which is linearly valid, although extensionally solvable only if a ground substitution term for each needed H_a can indeed be constructed. When this situation is encountered, the procedure terminates with success, but without instantiating the logical variables appearing in it. The substitution constructed up to this point, called a *pre-unifier*, is returned.

The possibility of achieving an algorithms à la Huet depend crucially on flex-flex equations being always solvable. If this property does not hold, as in some sublanguages of $S^{\to\multimap\&\top}$ we will discuss shortly, these equations must be analyzed with techniques similar to [18] or [21].

The procedure we just described is not guaranteed to terminate for generic equation systems since flex-rigid steps can produce arbitrarily complex new equations. However, it is sound in the sense that if a unifier or pre-unifier is returned the system is solvable (where free variables are allowed in the second case). It is also non-deterministically complete, i.e., every solution to the original system is an instance of a unifier or pre-unifier which can be found with our procedure. These properties are expressed by the theorems below. Detailed proofs can be found in [5]. We write $\mathcal{D} :: J$ if $\mathcal{D}$ is a derivation of the judgment J, and $[\Theta]\Xi$ for the result of applying the substitution Θ to each equation in Ξ.

Theorem 3.1 (*Soundness of linear pre-unification*)
If $\mathcal{X} :: \Xi \setminus \Xi_{ff}, \Theta$ and there is a substitution Θ_{ff} such that the multiset of equality judgments $[\Theta_{ff}]\Xi_{ff}$ has a derivation, then $[\Theta_{ff} \circ \Theta]\Xi$ is derivable.

Proof: By induction on the structure of $\mathcal{X}$. □

Note that it is not difficult to generalize this procedure to full unification (as, for example, in [27]), although we fail to see its practical value.

Theorem 3.2 (*Completeness of linear pre-unification*)
Given a system of equations Ξ and a substitution Θ such that $[\Theta]\Xi$ has a derivation, there are substitutions Θ_{ff} and Θ', and a system of flex-flex equations Ξ_{ff} such that $\Theta = (\Theta_{ff} \circ \Theta')_{|\,\mathrm{dom}\Theta}$, and $[\Theta_{ff}]\Xi_{ff}$ and $\Xi \setminus \Xi_{ff}, \Theta'$ are derivable.

Proof: By induction on the structure of a multiset of derivations $\vec{\mathcal{E}}$ for $[\Theta]\Xi$ excluding flex-flex equations, and on an appropriate measure on the image of the substitution Θ. □

3.4 Non-Determinism

Huet's pre-unification algorithm for $\lambda^{\to}$ is inherently non-deterministic since unification problems in this language usually do not admit most general unifiers. Indeed, when solving flex-rigid equations, we may have to choose between imitation and projection steps and, in the latter case, we might be able to project on

different arguments. The presence in $S^{\rightarrow -\circ \& \top}$ of a linear context and of constructs that operate on it gives rise to a number of new phenomena not present in $\lambda^{\rightarrow}$ unification.

First of all, the manner equations are rewritten in Figure 4 is constrained by the usual context management policy of linear logic. In particular, linear heads in rigid-rigid equations are removed from the context prior to unifying their spines (rule **pu_rr_lvar**). Moreover, when simplifying equations among pairs, the linear context is copied to the two subproblems (**pu_pair**), and equations involving $\langle\rangle$ can always be elided (**pu_unit**). Finally, when solving spine equations, the linear context must be distributed among the linear operands (**pu_lapp**) so that it is empty when the end of the spine is reached (**pu_nil**). As expected, equations among intuitionistic operands are created with an empty linear context (**pu_iapp**). Context splitting in rule **pu_lapp** represents a new form of non-determinism not present in Huet's algorithm. Standard techniques of lazy context management [3] can however be used in order to handle it efficiently and deterministically in an actual implementation.

A new inherent form of non-determinism arises in the generation of the spine of substitution terms. Recall that such a term V is constructed in two phases: first, we build its constructor layer, recording local intuitionistic and linear parameters in two accumulators Γ' and Δ', respectively, as λ-abstractions are introduced (first and third parts of Figure 5). Then, we construct a spine on the basis of the available type informations (second and fourth quarter of Figure 5), installing a fresh logical variable as the head of every operand. The contents of Γ' and Δ' must then be distributed as if they were contexts. In particular, we must split Δ' among the linear operands (rules **fri_llam** and **frp_llam**) so that, when the end of spine is generated, no linear parameter is left (rules **fri_nil** and **frp_nil**). Lazy strategies are not viable in general this time because the heads of these operands are logical variables. Therefore, we must be prepared to non-deterministically consider all possible splits.

This situation is illustrated by the equation

$$x{:}A, y{:}B; \cdot \vdash F \hat{} x \hat{} y = c \hat{} (G_1\ x\ y) \hat{} (G_2\ x\ y) : a.$$

discussed in Section 3.2. An imitation step instantiates F to a term of the form $\hat{\lambda} x'{:}A.\, \hat{\lambda} y'{:}B.\, c \hat{} M_1 \hat{} M_2$ where each of the *linear* parameters x' and y' must appear either in M_1 or in M_2, but not in both. This produces the four solutions presented in Section 3.2. An actual implementation would avoid this additional non-determinism by postponing the choices between the four imitations. A detailed treatment of the necessary constraints between variables occurrences is beyond the scope of this paper (see Section 4.2 for further discussion; a similar technique is used in [13]).

4 Discussion

In this section, we consider various sublanguages of $S^{\rightarrow -\circ \& \top}$ (or equivalently $\lambda^{\rightarrow -\circ \& \top}$) obtained by eliding some of the type operators and the corresponding term constructors and destructors (Section 4.1). We also discuss problems and sketch solutions towards the efficient implementation of a unification procedure for $\lambda^{\rightarrow -\circ \& \top}$ (Section 4.2).

4.1 Sublanguages

The omission of one or more of the type operators $\rightarrow$, $-\circ$, & and $\top$ and of the corresponding term constructs from $\lambda^{\rightarrow -\circ \& \top}$ (or $S^{\rightarrow -\circ \& \top}$) results in a number of λ-calculi with different properties.

First of all, the elision of $-\circ$, & and $\top$ reduces $\lambda^{\rightarrow -\circ \& \top}$ to $\lambda^{\rightarrow}$. The few applicable rules in Figures 4–5 constitute then a new presentation of Huet's procedure. The combined use of inference rules and of a spine calculus results in an elegant formulation that can be translated almost immediately into an efficient implementation.

Since linear objects in $\lambda^{\rightarrow -\circ \& \top}$ are created and consumed by linear abstraction and application, respectively, every sublanguage not containing $-\circ$ is purely intuitionistic. In particular, $\lambda^{\rightarrow \&}$ coincides with the simply-typed λ-calculus with pairs while $\lambda^{\rightarrow \& \top}$ corresponds to its extension with a unit type and unit element. Unification in the restricted setting of higher-order patterns has been studied for these two languages in [8] and [9], respectively. The appropriate restrictions of the rules in Figures 4–6 implement a general pre-unification procedure for these calculi.

The languages $\lambda^{\rightarrow -\circ \&}$ and $\lambda^{\rightarrow -\circ}$ are particularly interesting since the natural restriction of our pre-unification procedure is unsound for them in the following sense: We cannot apply our success criterion since not all flex-flex equations are solvable in this setting. Consider, for example,

$$x{:}A, y{:}B; \cdot \vdash F \hat{} x = G \hat{} y : a.$$

This equation has no solution since F must be instantiated with a term that, after β-reduction, will use explicitly x, and G to a term that must mention y. Furthermore, whether a flex-flex equation has a solution in $\lambda^{\rightarrow -\circ \&}$ or $\lambda^{\rightarrow -\circ}$ is in general undecidable, since, for example, $F \hat{} M_1 = F \hat{} M_2$ is equivalent to the generic unification problem $M_1 = M_2$. The situation is clearly different in $\lambda^{\rightarrow -\circ \& \top}$ where $\langle\rangle$ is always available as an

information sink in order to eliminate unused linear parameters. However, the usual assumption that there exist closed terms of every type may not be reasonable in $\lambda^{\rightarrow -\circ \& \top}$, and care must be taken in each application regarding the treatment of logical variables which may have no valid ground instances. In conclusion, pre-unification procedures in the sense of Huet are not achievable in the calculi with $-\circ$ but without $\top$.

Finally, a restricted form of unification in the purely linear calculus $\lambda^{-\circ}$ has been studied in [19]. The above counterexamples clearly apply also in this setting, but we have no result about the decidability of higher-order unification in this fragment.

4.2 Towards a Practical Implementation

Huet's algorithm for pre-unification in $\lambda^{\rightarrow}$ has been implemented in general proof search engines such as *Isabelle* [24] and logic programming languages such as *λProlog* [23] and shown itself to be reasonably efficient in practice. However, the non-determinism it introduces remains a problem, especially in logic programming. This issue is exacerbated in $\lambda^{\rightarrow -\circ \& \top}$ due to its additional resource non-determinism during imitation and projections.

For $\lambda^{\rightarrow}$, this problem has been addressed by Miller's language of higher-order patterns L_λ [21], which allows occurrences of logical variables to be applied to distinct parameters only. This syntactic restriction guarantees decidability and most general unifiers. An algorithm that solves equations in the pattern fragment but postpones as constraints any non L_λ constraint has been successfully implemented in the higher-order logic programming language *Elf* [25]. Unfortunately, an analogous restriction for $\lambda^{\rightarrow -\circ \& \top}$ which would cover the situations arising in practice does not admit most general unifiers. A simple example illustrating this is

$$x{:}a; \cdot \vdash F \hat{} x = c \hat{} (F_1 \hat{} x) \hat{} (F_2 \hat{} x) : a.$$

which has the two most general solutions

$F \longleftarrow \hat{\lambda} x'{:}a.\, c \hat{} (F_1 \hat{} x') \hat{} (H_2 \hat{} \langle\rangle), F_2 \longleftarrow \hat{\lambda} x''{:}a.\, H_2 \hat{} \langle\rangle$
$F \longleftarrow \hat{\lambda} x'{:}a.\, c \hat{} (H_1 \hat{} \langle\rangle) \hat{} (F_2 \hat{} x'), F_1 \longleftarrow \hat{\lambda} x''{:}a.\, H_1 \hat{} \langle\rangle$

neither of which is an instance of the other. This situation is common and occurs in several of our case studies. For certain flex-flex pattern equations, the set of most general unifiers cannot even be described finitely in the language of patterns, as illustrated by

$$x{:}a, y{:}a; \cdot \vdash F \hat{} \langle x, y\rangle = G \hat{} x \hat{} y : a.$$

for which the generic solution

$F \longleftarrow \hat{\lambda} w{:}a \,\&\, a.\, H \hat{} \langle H_1 \hat{} (\textsc{fst}\ w) \hat{} \langle\rangle, H_2 \hat{} (\textsc{snd}\ w) \hat{} \langle\rangle\rangle$
$G \longleftarrow \hat{\lambda} u{:}a.\, \hat{\lambda} v{:}a.\, H \hat{} \langle H_1 \hat{} u \hat{} \langle\rangle, H_2 \hat{} v \hat{} \langle\rangle\rangle$

yields one pattern substitution for each possible instantiation of the new logical variable H.

Despite these difficulties, the natural generalization of the notion of higher-order pattern introduced by [8] and [9] for products to the linear case, leads to a decidable unification problem for $\lambda^{\rightarrow -\circ \& \top}$. On this fragment (whose description is beyond the scope of the present paper), termination of the pre-unification algorithm in Section 3 is assured if we also incorporate an appropriate occurs-check as in the simply-typed case. Branching can furthermore be avoided by maintaining linear flex-flex equations as constraints and by using additional constraints between occurrences of parameters. In the first example above, the solution would be

$$F \longleftarrow \hat{\lambda} x'{:}a.\, c \hat{} (F_1 \hat{} x') \hat{} (F_2 \hat{} x')$$

with the additional constraint that if x' occurs in $F_1 \hat{} x'$ then it must be absorbed (by $\langle\rangle$) in $F_2 \hat{} x'$ and *vice versa* [13]. The second equation above would simply be postponed as a solvable equational constraint. Based on our experience with constraint simplification in *Elf* [25] and preliminary experiments, we believe that this will be a practical solution. In particular, the use of explicit substitutions, investigated in [7] relatively to *Elf*, seems to provide a hook for the required linearity constraints.

5 Conclusion and Future Work

In this extended abstract, we have studied the problem of higher-order unification in the context of the linear simply typed λ-calculus $\lambda^{\rightarrow -\circ \& \top}$. A pre-unification algorithm in the style of Huet has been presented for the equivalent spine calculus $S^{\rightarrow -\circ \& \top}$ and new sources of inherent non-determinism due to linearity were pointed out. Moreover, sublanguages of $\lambda^{\rightarrow -\circ \& \top}$ were analyzed and it was shown that pre-unification procedures are not achievable for some of them.

We are currently investigating the computational properties of the natural adaptation of Miller's higher-order patterns to $\lambda^{\rightarrow -\circ \& \top}$. Preliminary examples show that many common unifiable equations do not have most general unifiers due to non-trivial interferences among $-\circ$, & and $\top$. However, we believe that these problems can be solved through constraint simplification and propagation techniques in a calculus of explicit substitutions.

References

[1] A. Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Laboratory for Foundations of Computer Sciences, University if Edinburgh, Sept. 1996.

[2] I. Cervesato. *A Linear Logical Framework*. PhD thesis, Dipartimento di Informatica, Università di Torino, Feb. 1996.

[3] I. Cervesato, J. S. Hodas, and F. Pfenning. Efficient resource management for linear logic proof search. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proceedings of the 5th International Workshop on Extensions of Logic Programming*, pages 67–81, Leipzig, Germany, Mar. 1996. Springer-Verlag LNAI 1050.

[4] I. Cervesato and F. Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[5] I. Cervesato and F. Pfenning. Linear higher-order preunification, 1997.

[6] I. Cervesato and F. Pfenning. A linear spine calculus. Technical Report CMU-CS-97-125, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Apr. 1997.

[7] G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, Sept. 1996. MIT Press.

[8] D. Duggan. Unification with extended patterns. Technical Report CS-93-37, University of Waterloo, Waterloo, Ontario, Canada, July 1993. Revised March 1994 and September 1994.

[9] R. Fettig and B. Löchner. Unification of higher-order patterns in a simply typed lambda-calculus with finite products and terminal type. In H. Ganzinger, editor, *Proceedings of the Seventh International Conference on Rewriting Techniques and Applications*, pages 347–361, New Brunswick, New Jersey, July 1996. Springer-Verlag LNCS 1103.

[10] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[11] W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[12] J. Harland and D. Pym. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, Apr. 1994.

[13] J. Harland and D. Pym. Resource distribution via boolean constraints. In W. McCune, editor, *Proceedings of the Fourteenth International Conference on Automated Deduction — CADE-14*, Townsville, Australia, July 1997. To appear.

[14] H. Herbelin. *Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes.* PhD thesis, Université Paris 7, 1995.

[15] J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.

[16] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[17] S. Ishtiaq and D. Pym. A relevant analysis of natural deduction, Dec. 1996. Manuscript.

[18] D. C. Jensen and T. Pietrzykowski. Mechanizing ω-order type theory through unification. *Theoretical Computer Science*, 3:123–171, 1976.

[19] J. Levy. Linear second-order unification. In H. Ganzinger, editor, *Proceedings of the Seventh International Conference on Rewriting Techniques and Applications*, pages 332–346, New Brunswick, New Jersey, July 1996. Springer-Verlag LNCS 1103.

[20] S. Michaylov and F. Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania. Available as Technical Report MS-CIS-92-86.

[21] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In P. Schroeder-Heister, editor, *Proceedings of the International Workshop on Extensions of Logic Programming*, pages 253–281, Tübingen, Germany, 1989. Springer-Verlag LNAI 475.

[22] D. Miller. A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[23] G. Nadathur and D. Miller. An overview of λProlog. In K. A. Bowen and R. A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, Aug. 1988. MIT Press.

[24] T. Nipkow and L. C. Paulson. Isabelle-91. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 673–676, Saratoga Springs, NY, 1992. Springer-Verlag LNAI 607. System abstract.

[25] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[26] C. Prehofer. *Solving Higher-Order Equations: From Logic to Programming.* PhD thesis, Technische Universität München, Mar. 1995.

[27] W. Snyder and J. H. Gallier. Higher order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(1-2):101–140, 1989.