Algorithms for Equality and Unification in the Presence of Notational Definitions

Frank Pfenning and Carsten Schürmann [⋆]

Carnegie Mellon University
School of Computer Science
fp@cs.cmu.edu carsten@cs.cmu.edu

1 Introduction

Notational definitions are pervasive in mathematical practice and are therefore supported in most automated theorem proving systems such as Coq [B$^+$98], PVS [ORS92], Lego [LP92], or Isabelle [Pau98]. Semantically, notational definitions are transparent, that is, one obtains the meaning of an expression by interpreting the result of expanding all definitions. Pragmatically, however, expanding all definitions as they are encountered is unsatisfactory, since it can be computationally expensive and complicate the user interface.

In this paper we investigate the interaction of notational definitions with algorithms for testing equality and unification. We propose a syntactic criterion on definitions which avoids their expansion in many cases without losing soundness or completeness with respect to $\beta\eta\delta$-conversion. Our setting is the dependently typed λ-calculus [HHP93], but, with minor modifications, our results should apply to richer type theories and logics.

The question when definitions need to be expanded is surprisingly subtle and of great practical importance. Most algorithms for equality and unification rely on decomposing a problem

$$c\,M_1 \ldots M_n \doteq c\,N_1 \ldots N_n$$

into

$$M_1 \doteq N_1, \ldots, M_n \doteq N_n.$$

However, if c is defined this is not necessarily complete. For example, if $k = \lambda x.\ c'$ then

$$k\,M \doteq k\,N$$

for *every* M and N. Always expanding definitions is computationally expensive, especially when they duplicate their arguments. Expanding them only when the equality between the arguments fails often performs much redundant computation, and, moreover, is incomplete in the presence of meta-variables. For example, with the same definition for k,

$$k\,X \doteq k\,c'$$

[⋆] This work was supported by NSF Grant CCR-9619584

would succeed without expanding k with the substitution $X = c'$, even though the most general unifier leaves X uninstantiated.

We identify a class of definitions which we call *strict* and for which decomposition is complete. It also solves a related problem with the completeness of the so-called *occurs-check* during unification. Fortunately, most notational definitions are strict in the sense we define. We do not deal with recursive definitions which require different considerations and have been treated in the literature on functional logic programming [Han94]. Other aspects of notational definitions in mathematical practice have been studied by Griffin [Gri88].

We have implemented a strictness checker and unification algorithm in Twelf [SP98], an implementation of the logical framework LF which supports type reconstruction, logic programming, and theorem proving. It has been applied on a variety of examples from the area of logics and programming languages.

This paper is organized as follows. In Section 2 we describe a spine formulation of LF with definitions, and in Section 3 a small logic as running example. In Section 4 we describe the strictness algorithm and show its correctness. We assess our results in Section 5 and conclude and describe future work in Section 6.

2 Language

The type theory underlying the logical framework LF [HHP93] is divided into three levels: objects, types, and kinds. We deviate from standard formulations by adopting a spine notation for application [CP97] and by adding definitions. The spine notation contributes significantly to the concise presentation of the theory in Section 4 and corresponds closely to the implementation in Twelf. We use a for constant type families, x for object-level variables, and c for constructors (that is, declared constants without a definition) and d for defined constants.

$$
\begin{array}{lrcl}
\text{Kinds:} & K & ::= & \text{type} \mid \Pi x{:}A.\,K \\
\text{Types:} & A & ::= & a \cdot S \mid \Pi x{:}A_1.\,A_2 \\
\text{Objects:} & M & ::= & H \cdot S \mid \lambda x{:}A.\,M \mid M \cdot S \\
\text{Heads:} & H & ::= & x \mid c \mid d \\
\text{Spines:} & S & ::= & \text{nil} \mid M; S \\
\text{Signature:} & \Sigma & ::= & \cdot \mid \Sigma, a : K \mid \Sigma, c : A \mid \Sigma, d : A = M \\
\text{Contexts:} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

$a \cdot S$ and $H \cdot S$ are our notation for the application of a variable or constant to arguments given as a *spine*. Such terms are in weak head-normal form unless the constant at the head is defined. A *β-redex* has the form $M \cdot S$. For the sake of readability, we omit the trailing nil from spines, and if the spine is empty, we also omit the "$\cdot$". $\Pi x{:}A_1.\,A_2$ is a function type, which we may write as $A_1 \to A_2$ if x does not occur free in A_2.

The notion of definitional equality is based on $\beta\eta$-conversion on δ-reduction which expands definitions.

$$d \cdot S \longrightarrow_\delta M \cdot S \qquad \text{where } d : A = M \in \Sigma$$

We assume that constants and variables are declared at most once in a signature and context, respectively. As usual we apply tacit renaming of bound variables to maintain this assumption and to guarantee capture-avoiding substitution.

The LF type theory is defined by a number of mutually dependent judgments which define valid objects, types, kinds, contexts, and signatures, and, in our case, also heads and spines. We will not reiterate the rules here (see [HHP93,CP97]). The main typing judgments are of the form $\Gamma \vdash_\Sigma M : A$ — expressing that object M has type A in context Γ — and $\Gamma \vdash_\Sigma S : A > A'$ — expressing that the spine S acts as a vector of well-typed arguments to a head of type A returning a result of type A'. A definition $d : A = M$ is well-formed in a signature Σ if $\cdot \vdash_\Sigma M : A$.

We generally assume that signature Σ is valid and fixed and therefore omit it from the typing and other related judgments introduced later on. In a slight departure from [HHP93] we take $\beta\eta\delta$-conversion as our notion of definitional equality since this guarantees that every well-typed object has an equivalent *canonical form*, that is, a long $\beta\eta\delta$-normal form. Canonical forms are commonly characterized by three mutual dependent judgments, among them the judgment for weak-head reduction $M \xrightarrow{\text{whr}} M'$ which applies local β- or δ-reductions. The theory developed in [Geu92], for example, still applies since definitions are transparent.

We write $\Gamma \vdash M_1 \equiv M_2$ to express that two definition-free and well-typed objects M_1 and M_2 are equivalent modulo $\beta\eta\delta$-conversion. Similarly, for spines, we write $\Gamma \vdash S_1 \equiv S_2$.

Since all validity judgments are decidable with well-understood algorithms, we tacitly assume that all objects, types, kinds, spines, heads, contexts, and signatures are valid and, for equalities, that both sides have the same type or kind.

Our proofs exploit the following standard properties of definitional equality based on $\beta\eta\delta$-conversion.

Property 1 (Equivalence).

1. For all M, $\Gamma \vdash M \equiv M$.
2. Let H_1, H_2 be heads x or c,
   $\Gamma \vdash H_1 \cdot S_1 \equiv H_2 \cdot S_2$ iff $H_1 = H_2$ and $\Gamma \vdash S_1 \equiv S_2$
3. $\Gamma \vdash a_1 \cdot S_1 \equiv a_2 \cdot S_2$ iff $a_1 = a_2$ and $\Gamma \vdash S_1 \equiv S_2$
4. $\Gamma \vdash \lambda y{:}A_1.\, M_1 \equiv \lambda y{:}A_2.\, M_2$ iff $\Gamma \vdash A_1 \equiv A_2$ and $\Gamma, y : A_1 \vdash M_1 \equiv M_2$
5. $\Gamma \vdash \Pi y{:}A_1.\, M_1 \equiv \Pi y{:}A_2.\, M_2$ iff $\Gamma \vdash A_1 \equiv A_2$ and $\Gamma, y : A_1 \vdash M_1 \equiv M_2$
6. For all M_1, M_2 in which y does not occur free,
   $\Gamma, y : A \vdash M_1 \cdot y \equiv M_2 \cdot y$ iff $\Gamma \vdash M_1 \equiv M_2$
7. $\Gamma \vdash M_1; S_1 \equiv M_2; S_2$ iff $\Gamma \vdash M_1 \equiv M_2$ and $\Gamma \vdash S_1 \equiv S_2$

For a well-typed definition $d : A = M$ the head-normal form of M must always exist and have the shape $M = \lambda x_1 : A_1. \dots \lambda x_n : A_n. H \cdot S$. We call $x_1, \dots, x_n$ *argument parameters*, and all other parameters in the body $H \cdot S$ *local parameters*.

3 Example

To illustrate our algorithms we use the encoding of a small fragment of first-order intuitionistic logic in LF [HHP93].

$$\text{Formulas: } F ::= \top \mid \bot \mid F_1 \supset F_2$$

Formulas are represented as a type and each connective as a constant.

$$\begin{array}{ll@{\qquad\qquad}ll}
 & & \mathrm{o} & : \mathrm{type} \\
\ulcorner \top \urcorner & = \mathrm{true} & \mathrm{true} & : \mathrm{o} \\
\ulcorner \bot \urcorner & = \mathrm{false} & \mathrm{false} & : \mathrm{o} \\
\ulcorner F_1 \supset F_2 \urcorner & = \mathrm{imp} \cdot (\ulcorner F_1 \urcorner ; \ulcorner F_2 \urcorner) & \mathrm{imp} & : \mathrm{o} \to \mathrm{o} \to \mathrm{o}
\end{array}$$

This simple logic can now be extended by negation in the usual way, by defining $\neg A \overset{\mathrm{def}}{=} A \supset \bot$, which leads to a definition of the constant *not* in terms of the other constants.

$$\mathrm{not} : \mathrm{o} \to \mathrm{o} = \lambda A{:}\mathrm{o}.\, \mathrm{imp} \cdot (A; \mathrm{false})$$

We write $\vdash A$ to express that the formula A has a natural deduction, using the following four rules:

$$\frac{}{\vdash \top}\,\top I \qquad \frac{\vdash \bot}{\vdash C}\,\top E \qquad \frac{\begin{array}{c}\overline{\vdash A}^{\,u} \\ \vdots \\ \vdash B\end{array}}{\vdash A \supset B}\,\supset I^u \qquad \frac{\vdash A \supset B \quad \vdash A}{\vdash B}\,\supset E$$

As shown in [HHP93], there is an adequate encoding of this calculus in LF: The judgment $\vdash A$ is represented as dependent type family, and the four rules as object constants.

$$\begin{array}{ll}
\mathrm{nd} & : \mathrm{o} \to \mathrm{type} \\
\mathrm{truei} & : \mathrm{nd} \cdot \mathrm{true} \\
\mathrm{falsee} & : \mathit{\Pi} C{:}\mathrm{o}.\, \mathrm{nd} \cdot \mathrm{false} \to \mathrm{nd} \cdot C \\
\mathrm{impi} & : \mathit{\Pi} A{:}\mathrm{o}.\, \mathit{\Pi} B{:}\mathrm{o}.\, (\mathrm{nd} \cdot A \to \mathrm{nd} \cdot B) \to \mathrm{nd} \cdot (\mathrm{imp} \cdot (A; B)) \\
\mathrm{impe} & : \mathit{\Pi} A{:}\mathrm{o}.\, \mathit{\Pi} B{:}\mathrm{o}.\, \mathrm{nd} \cdot (\mathrm{imp} \cdot (A; B)) \to \mathrm{nd} \cdot A \to \mathrm{nd} \cdot B
\end{array}$$

The usual introduction and elimination rules of $\neg A$ can then be formulated as derived rules of inference.

$$\dfrac{\begin{array}{c}\overline{\vdash A}^{\,u}\\ \vdots\\ \vdash \bot\end{array}}{\vdash \neg A}\neg I^u \qquad \dfrac{\vdash \neg A \quad \vdash A}{\vdash \bot}\neg E$$

Clearly, $\neg I^u$ is a restriction of $\supset I^u$ and $\neg E$ is a restriction of $\supset E$. We represent these rules as defined constants in LF. This is an example of a notational definition at the level of derivations.

$$\begin{aligned}
\mathrm{noti} &: \mathit{\Pi} A{:}o.\, (\mathrm{nd}\cdot A \to \mathrm{nd}\cdot \mathrm{false}) \to \mathrm{nd}\cdot(\mathrm{not}\cdot A)\\
&= \lambda A{:}o.\, \lambda u{:}(\mathrm{nd}\cdot A \to \mathrm{nd}\cdot \mathrm{false}).\, \mathrm{impi}\cdot(A;\mathrm{false};u)\\
\mathrm{note} &: \mathit{\Pi} A{:}o.\, \mathrm{nd}\cdot(\mathrm{not}\cdot A) \to \mathrm{nd}\cdot A \to \mathrm{nd}\cdot \mathrm{false}\\
&= \lambda A{:}o.\, \lambda u_1{:}\mathrm{nd}\cdot(\mathrm{not}\cdot A).\, \lambda u_2{:}\mathrm{nd}\cdot A.\, \mathrm{impe}\cdot(A;\mathrm{false};u_1;u_2)
\end{aligned}$$

4 Definitions and Algorithms for Equality

Notational definitions are pervasive in mathematical practice. They are convenient in many situations and are therefore supported in most automated theorem proving systems. We do not explicitly treat other forms of definitions, such as recursive definitions, but our techniques are applicable in more general circumstances. For example, in MLF [HP] — an implementation of LF extended with a module system — definitions are used to express logical interpretations.

Semantically, definitions are transparent, that is, the meaning of any term can be determined by expanding all definitions. But from a pragmatic point of view, expanding all definitions is unsatisfactory, for several reasons. First of all, even if the definitions are simple, their expansion is likely to be required frequently, in the core of an implementation. Secondly, definitions can duplicate their arguments, leading to an exponential increase in size unless special implementation techniques are employed. Thirdly, expanding all definitions causes error messages and results of computations not to use any defined constants at all, which often renders them illegible.

In this section we characterize a class of definitions whose expansion can frequently be avoided when comparing terms for equality. Based on these results, we show in the next section that the same criterion can be used to even greater benefit in unification.

4.1 Injectivity

Most algorithms for equality and unification rely on decomposing a problem

$$d\cdot S_1 \equiv d\cdot S_2 \qquad (1)$$

into

$$S_1 \equiv S_2 \tag{2}$$

but if $d : A = M$ is a notational definition, then (1) stands for

$$M \cdot S_1 \equiv M \cdot S_2. \tag{3}$$

Since $\equiv$ is a congruence, it follows trivially that (2) always implies (3). But the reverse does not necessarily hold, for example, if M ignores an argument. We call those terms M for which (3) implies (2) *injective*. For definitions which are injective, decomposition is complete. Recall that we assume all signatures, context, objects, equations, *etc.* to be valid.

Definition 1 (Injectivity). *A definition $d : A = M$ is* injective *iff for all contexts Γ_0 and spines S_1 and S_2,*

$$\Gamma_0 \vdash M \cdot S_1 \equiv M \cdot S_2 \quad \textit{implies} \quad \Gamma_0 \vdash S_1 \equiv S_2.$$

4.2 Strictness

Many algorithms for equality avoid expanding definitions in equations of the form $d \cdot S_1 \equiv d \cdot S_2$ until the equality of the arguments $S_1 \equiv S_2$ fails. If that happens, definitions are expanded, and the algorithm continues with the expanded terms, probably redoing much previous computation. Without further improvements such an algorithm would be exponential in the worst case. In contrast, if we know that d is injective, the algorithm can fail immediately.

Since injectivity is a semantic criterion, we have developed a syntactic criterion called *strictness* which guarantees injectivity and which can be easily checked. Informally, a notational definition is said to be strict, if each argument parameter occurs at least once in a *rigid position* [Hue75], applied only to pairwise distinct local parameters. If there are no defined constants, the rigid positions in a β-normal form are those resulting from erasing the spines following argument parameters. If there are defined constants we distinguish (inductively) between strict and non-strict ones: the former are treated like constructors, the latter are expanded. We also do not consider the head of a definition to be a rigid position (see Example 2).

The definition of *not*, for example, is strict, because A appears in a rigid position. *noti* is also strict, because its argument parameters A and u occur in rigid positions. The same holds for *note*, because A, u_1, and u_2 occur in rigid positions.

In the following we analyze some counterexamples to illustrate strictness and its relation to injectivity.

Example 1 (Universal quantification). The logic presented in Section 3 can be extended to first-order by introducing terms T and a universal quantifier

$$F ::= ... \mid \forall x.F$$

In LF terms are represented by objects of a new type i, and the universal quantifier by a new constructor

$$\mathrm{forall} : (\mathrm{i} \to \mathrm{o}) \to \mathrm{o}.$$

The (true) formula $(\forall x.F(x)) \supset F(t)$ can be defined as

$$\mathrm{allinst} = \lambda A\!:\!\mathrm{i} \to \mathrm{o}.\, \lambda T\!:\!\mathrm{i}.\, \mathrm{imp} \cdot (\mathrm{forall} \cdot A; A \cdot T)$$

allinst is not strict because T does not occur in a rigid position. Indeed, if $F(x)$ does not actually depend on x, then t is not uniquely determined and

$$\mathrm{allinst} \cdot (F; T) \equiv \mathrm{allinst} \cdot (F; T')$$

would be true even if T and T' were different.

Example 2 (Identity). The definition of the identity at function type, $\mathrm{id} = \lambda A\!:\!\mathrm{o} \to \mathrm{o}.\, \lambda B\!:\!\mathrm{o}.\, A \cdot B$, is not strict because the only occurrence of A is at the head of the definition, which is not a rigid position. It is also not injective, because

$$\mathrm{id} \cdot (\lambda x\!:\!\mathrm{o}.\, \mathrm{true}; \mathrm{false}) \equiv \mathrm{id} \cdot (\lambda x\!:\!\mathrm{o}.\, \mathrm{true}; \mathrm{true})$$

can be reduced to

$$\mathrm{true} \equiv \mathrm{true}.$$

Example 3 (Identity at base type). The definition $\mathrm{id}' = \lambda A\!:\!\mathrm{o}.\, A$ is not strict for the same reason as in the preceding example. However, the identity at base type is injective. We must rule it out for different reasons (see the discussion of the occurs-check in unification in Section 5).

Example 4 (Evaluation). Consider $\mathrm{at} = \lambda x\!:\!\mathrm{o} \to \mathrm{o}.\, \mathrm{not} \cdot (x \cdot \mathrm{true})$. Note, that the argument to x is not a local parameter but a constant. The definition is hence not strict. The equality problem

$$\mathrm{at} \cdot (\lambda x\!:\!\mathrm{o}.\, x) \equiv \mathrm{at} \cdot (\lambda x\!:\!\mathrm{o}.\, \mathrm{true})$$

can be expanded to

$$(\lambda x\!:\!\mathrm{o} \to \mathrm{o}.\, \mathrm{not} \cdot (x \cdot \mathrm{true})) \cdot (\lambda x\!:\!\mathrm{o}.\, x) \equiv (\lambda x\!:\!\mathrm{o} \to \mathrm{o}.\, \mathrm{not} \cdot (x \cdot \mathrm{true})) \cdot (\lambda x\!:\!\mathrm{o}.\, \mathrm{true})$$

which is solvable because $\mathrm{not} \cdot \mathrm{true} \equiv \mathrm{not} \cdot \mathrm{true}$. Hence, the definition is not injective.

The first part in the definition of strictness formalizes the requirement that arguments to rigid occurrences of argument parameters must be pairwise distinct local parameters. This is exactly the requirement imposed on *higher-order patterns* by Miller [Mil91]. In the judgments below we generally use Γ for a context consisting of argument parameters to a definition, and Δ consisting of local parameters.

Definition 2 (Pattern spine). *Let Δ be a context, S be a spine. S is a pattern spine iff $\Delta \vdash S\ pat$ holds which is defined by the following rules:*

$$\frac{}{\Delta \vdash \mathrm{nil}\ pat}\ \mathsf{ps\_nil} \qquad \frac{\Delta_1, \Delta_2 \vdash S\ pat}{\Delta_1, x : A, \Delta_2 \vdash x; S\ pat}\ \mathsf{ps\_;}$$

The formal system for strictness is defined by three mutually dependent judgments. The central judgment of *local strictness*, $\Gamma; \Delta \vdash_x M$, enforces that the argument parameter x occurs in a rigid position in M where it is applied to a pattern spine. Every argument parameter must be locally strict, which is enforced by *global strictness*, $\Gamma \Vdash M$. As an auxiliary judgment we use *relative strictness*, $\Gamma \Vdash_x M$ where the leading abstractions in M are treated as argument parameters.

Definition 3 (Strictness). *Let Γ be a context of argument parameters, and Δ a context of local parameters. We define*

1. *local strictness for $x : A \in \Gamma$: $\Gamma; \Delta \vdash_x M$,*
2. *relative strictness for $x : A \in \Gamma$: $\Gamma \Vdash_x M$,*
3. *global strictness: $\Gamma \Vdash M$*

by the rules in Figure 1. We say that the definition $d : A = M$ is strict if $\cdot \Vdash M$ holds.

The main technical contribution of this paper is that the system of strictness is strong enough to guarantee injectivity.

Theorem 1 (Injectivity). *If $d : A = M$ is strict, that is, $\cdot \Vdash M$, then $d : A = M$ is injective.*

The proof is non-trivial and requires a sequence of lemmas which are omitted from this technical summary.

The rules of strictness implicitly define an algorithm to decide if a definition is strict or not. The algorithm traverses the structure of a term visiting all rigid positions. If it finds at least one occurrence of every argument parameter of the definition applied to a pattern spine ($\mathsf{ls\_pat}$), it stops and signals success. If the algorithm comes to a defined and strict constant, it applies $\mathsf{ls\_d}$ or $\mathsf{rs\_d}$, otherwise it expands the definition using $\mathsf{ls\_red}$ or $\mathsf{rs\_red}$, respectively. The algorithm terminates for $\mathsf{ls\_red}$ and $\mathsf{rs\_red}$, because definitions can be only finitely often expanded since they cannot be recursive. In an implementation of this algorithm, one would annotate each definition with strictness information, and hence no redundant computation is necessary for $\mathsf{ls\_d}$ and $\mathsf{rs\_d}$. This algorithm has been implemented in the Twelf system [SP98].

It is easy to verify that all definitions from Section 3 satisfy the strictness condition. Definitions at base type are always strict. Definitions in normal form whose argument parameters are of base type are strict if each argument parameters occurs and it is not the identity. Most notational definitions of these two forms are thus accepted by our criterion.

$$\dfrac{\Gamma; \Delta \vdash_x A}{\Gamma; \Delta \vdash_x \lambda y{:}A.\, M}\,\textsf{ls\_ld} \qquad \dfrac{\Gamma; \Delta, y : A \vdash_x M}{\Gamma; \Delta \vdash_x \lambda y{:}A.\, M}\,\textsf{ls\_lb}$$

$$\dfrac{\Gamma; \Delta \vdash_x A_1}{\Gamma; \Delta \vdash_x \Pi y{:}A_1.\, A_2}\,\textsf{ls\_pd} \qquad \dfrac{\Gamma; \Delta, y : A_1 \vdash_x A_2}{\Gamma; \Delta \vdash_x \Pi y{:}A_1.\, A_2}\,\textsf{ls\_pb}$$

$$\dfrac{M \cdot S \xrightarrow{\text{whr}} M' \quad \Gamma; \Delta \vdash_x M'}{\Gamma; \Delta \vdash_x M \cdot S}\,\textsf{ls\_red} \qquad \dfrac{d : A = M \in \Sigma \quad \cdot \Vdash M \quad \Gamma; \Delta \vdash_x S}{\Gamma; \Delta \vdash_x d \cdot S}\,\textsf{ls\_d}$$

$$\dfrac{\Gamma; \Delta \vdash_x S}{\Gamma; \Delta \vdash_x c \cdot S}\,\textsf{ls\_c} \qquad \dfrac{\Gamma; \Delta \vdash_x S}{\Gamma; \Delta \vdash_x a \cdot S}\,\textsf{ls\_a}$$

$$\dfrac{\Delta \vdash S \text{ pat}}{\Gamma; \Delta \vdash_x x \cdot S}\,\textsf{ls\_pat} \qquad \dfrac{y : A \in \Delta \quad \Gamma; \Delta \vdash_x S}{\Gamma; \Delta \vdash_x y \cdot S}\,\textsf{ls\_var}$$

$$\dfrac{\Gamma; \Delta \vdash_x M}{\Gamma; \Delta \vdash_x M; S}\,\textsf{ls\_hd} \qquad \dfrac{\Gamma; \Delta \vdash_x S}{\Gamma; \Delta \vdash_x M; S}\,\textsf{ls\_sp}$$

……

$$\dfrac{M \cdot S \xrightarrow{\text{whr}} M' \quad \Gamma \Vdash_x M'}{\Gamma \Vdash_x M \cdot S}\,\textsf{rs\_red} \qquad \dfrac{d : A = M \in \Sigma \quad \cdot \Vdash M \quad \Gamma; \cdot \vdash_x d \cdot S}{\Gamma \Vdash_x d \cdot S}\,\textsf{rs\_d}$$

$$\dfrac{\Gamma; \cdot \vdash_x c \cdot S}{\Gamma \Vdash_x c \cdot S}\,\textsf{rs\_c} \qquad \dfrac{\Gamma, y : A \Vdash_x M}{\Gamma \Vdash_x \lambda y{:}A.\, M}\,\textsf{rs\_pi}$$

……

$$\dfrac{}{\Gamma \Vdash M \cdot S}\,\textsf{gs\_atom} \qquad \dfrac{\Gamma, x : A \Vdash_x M \quad \Gamma, x : A \Vdash M}{\Gamma \Vdash \lambda x{:}A.\, M}\,\textsf{gs\_pi}$$

Fig. 1. A formal system for strictness

At higher types, one more frequently encounters definitions which are not injective. Consequently, they cannot be strict according to our definition. A more accurate extension would have to analyze the structure of functional arguments to higher-order definitions, as in the case of strictness analysis for functional programming languages (see, for example, [HM94]). However, we suspect one quickly reaches the point of diminishing returns for this kind of complex analysis.

5 Results for Unification

So far we have shown how algorithms for testing equality (that is, $\beta\eta\delta$-convertibility) can be improved by using strictness. In the presence of meta-variables these observations can be generalized to unification. We write $\Gamma \vdash M_1 \approx M_2$ for a unification problem, where M_1, M_2 are well-typed objects of the same type which can contain meta-variables.

When to expand definitions is in this setting more subtle than for plain equality algorithms. Expanding them only in the case of failure may return a unifier which is not most general and hence renders the algorithm incomplete. Not expanding them may cause an unnecessary occurs-check failures, yet another source of incompleteness. The following two examples show these situations.

Example 5 (Most-general unifier). Let $\mathrm{tr} : \mathrm{o} \to \mathrm{o} = \lambda x\!:\!\mathrm{o}.\,\mathrm{true}$ a definition, and X a meta variable. The unification problem $\cdot \vdash \mathrm{tr} \cdot \mathrm{false} \approx \mathrm{tr} \cdot X$ has as solution $X = \mathrm{false}$ if tr is not expanded, which is not most general, since the most general solution leaves X uninstantiated.

Example 6 (Occurs-check). Let tr be the same definition as above, and X a meta variable. The unification problem $\cdot \vdash X \approx \mathrm{tr} \cdot X$ has no solution if tr is not expanded, because X occurs on its left-hand side and as an argument to tr. But obviously the problem has a solution, $X = \mathrm{true}$.

Most unification algorithm decompose a unification problem of the form

$$\Gamma_0 \vdash d \cdot S_1 \approx d \cdot S_2 \tag{4}$$

into

$$\Gamma_0 \vdash S_1 \approx S_2 \tag{5}$$

and so does the unification algorithm for the higher-order pattern fragment [DHKP96] which is employed in Twelf. If $d : A = M$ is a strict definition, then because of injectivity, every unifier Θ of (4) is also a unifier of (5) and vice versa.

$$\begin{array}{rc}
 & \Gamma_0 \vdash (d \cdot S_1)[\Theta] \equiv (d \cdot S_2)[\Theta] \\
\text{iff} & \Gamma_0 \vdash d \cdot (S_1[\Theta]) \equiv d \cdot (S_2[\Theta]) \\
\text{iff} & \Gamma_0 \vdash S_1[\Theta] \equiv S_2[\Theta]
\end{array}$$

This guarantees that the unifier determined by the unification algorithm which does not expand strict definitions unless the two heads differ, is most general.

Theorem 2 (Most general unifiers). *Let $d : A = M$ be a strict definition, Γ_0 a context, and S_1, S_2 spines. Then the unification problems*

$$\Gamma_0 \vdash M \cdot S_1 \approx M \cdot S_2$$

and

$$\Gamma_0 \vdash S_1 \approx S_2$$

have the same set of solutions.

In addition, such a unification algorithm also treats the occurs-check problem correctly: We say that $\Gamma_0 \vdash X \approx M$ fails the *occurs-check* if X has a strict occurrence in M. This is a generalization of Huet's original *rigid path* criterion for non-unifiability by allowing some arguments to X. Note also that our version of occurs-check does not need to expand strict definitions. We conjecture, but at this moment we have not formally proven, that unification problems which fail the occurs-check have no unifier. Informally, one assumes a solution Θ for X and then counts the number of constructor and parameter occurrences in the normal form of $\Theta(X)$ and $\Theta(M)$ to arrive at a contradiction. The occurs-check is also the reason why identity functions are not considered strict. An equation $X \equiv \mathrm{id}' \cdot X$ would fail the occurs-check but have a solution (where X is uninstantiated).

If our conjecture holds, strict definitions can be treated mostly as constructors in a unification algorithm. They must be expanded only in the case of a constant clash at the head during decomposition of so-called rigid-rigid equations. The unification algorithm remains sound and complete. Note that this observation is independent of whether one uses an algorithm based on Miller's higher-order patterns or Huet's original algorithm for higher-order unification.

6 Conclusion

We have identified a class of strict notational definitions and analyzed the way they interact with algorithms for equality and unification. Only in the case of constant clash notational definitions must be expanded. This property can be exploited to make many implementations of those algorithms more efficient, while preserving completeness and soundness with respect to $\beta\eta\delta$-conversion. We also presented an algorithm to efficiently check definitions for strictness.

Many theorem provers rely on an *ad hoc* treatment of definitions. We believe that these systems can benefit from the results of this paper in terms of efficiency and robustness.

In future work we plan to evaluate the concept of strictness empirically in our implementation. If warranted by the results, we may investigate *partially* strict definitions, that is, definitions, where some of the argument parameters are locally strict, and others are not. In such a situation, definitions may only need to be "partially expanded", comparing the strict and reducing the non-strict argument positions.

References

[B$^+$98] Bruno Barras et al. *The Coq Proof Assistant, Reference Manual, Version 6.2.* INRIA, CNRS, France, 1998.

[CP97] Iliano Cervesato and Frank Pfenning. A linear spine calculus. Technical Report CMU-CS-97-125, CMU, 1997.

[DHKP96] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In *Joint International Conference and Symposium on Logic Programming (JICSLP'96), Bonn, Germany*, 1996.

[Geu92] Herman Geuvers. The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi. In A. Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 453–460, Santa Cruz, California, June 1992.

[Gri88] Timothy G. Griffin. Notational definition — a formal account. In *Third Annual Symposium on Logic in Computer Science, Edinburgh, Scotland*, pages 372–383. IEEE, July 1988.

[Han94] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HM94] Chris Hankin and Daniel Le Métayer. Deriving algorithms from type inference systems: Application to strictness analysis. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages, Portland*, pages 202–212. ACM, January 1994.

[HP] Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*. To appear. A preliminary version is available as Technical Report CMU-CS-92-191, September 1992.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[LP92] Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Pau98] Lawrence C. Paulson. *Introduction to Isabelle*, 1998.

[SP98] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In *Proceedings of the Fifteenth International Conference on Automated Deduction*, pages 269–283, Lindau, Germany, 1998. too appear.