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 [Pau94]. 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 $\beta\delta$-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
To appear in TYPES'98, Springer-Verlag LNCS

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 (called *strict*) for which decomposition is complete. It also solves a related problem with the completeness of the so-called *occurs-check* during unification by generalizing Huet's *rigid path* criterion [Hue75]. Fortunately, most notational definitions are strict in the sense we define. We do not deal with recursive definitions, for example, 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 [PS98], an implementation of the logical framework LF which supports type reconstruction, logic programming, and theorem proving. It has been applied to a variety of examples from the area of logics and programming languages. The Twelf system is freely available from the Twelf homepage `http://www.cs.cmu.edu/~twelf`.

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 criterion and show its correctness. We generalize our results from conversion to unification 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. In spine notation, we write $c \cdot M_1; ...; M_n; \mathrm{nil}$ for a term $c\ M_1\ ...\ M_n$ to make its head explicit. It 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. For simplicity, we only allow definitions at the level of objects, but the results also apply to definitions at the level of types.

$$
\begin{array}{ll}
\text{Kinds:} & K ::= \mathrm{type} \mid \Pi x\!:\!A.\, K \\
\text{Types:} & A ::= a \cdot S \mid \Pi x\!:\!A_1.\, A_2 \\
\text{Objects:} & M ::= H \cdot S \mid \lambda x\!:\!A.\, M \mid M \cdot S \\
\text{Heads:} & H ::= x \mid c \mid d \\
\text{Spines:} & S ::= \mathrm{nil} \mid M; S \\[4pt]
\text{Signature:} & \Sigma ::= \cdot \mid \Sigma, a : K \mid \Sigma, c : A \mid \Sigma, d : A = M \\
\text{Contexts:} & \Gamma ::= \cdot \mid \Gamma, x : A
\end{array}
$$

$a \cdot S$ 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. 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. In the examples we sometimes omit types and write definitions as $d = M$.

As in [CP97] we assume throughout that all objects are in η-long form. Note that η-long forms are preserved under $\beta\delta$-conversion. Working only with η-long forms simplifies the presentation of the formal judgments and proofs, but is not essential. Our results still hold if we drop this assumption, both with and without η-conversion. The notion of definitional equality is then based on $\beta\delta$-conversion where δ-reduction expands definitions.

$$\begin{array}{rcll} M \cdot \mathrm{nil} & \longrightarrow_\beta & M & \\ (\lambda x{:}A.\,M) \cdot (N; S) & \longrightarrow_\beta & ([N/x]M) \cdot S & \\ d \cdot S & \longrightarrow_\delta & M \cdot S & \qquad\qquad \text{where } d : A = M \in \Sigma \end{array}$$

A *β-redex* has the form $M \cdot S$, a *δ-redex* the form $d \cdot S$.

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 below. We take $\beta\delta$-conversion as our notion of definitional equality which guarantees that every well-typed object has an equivalent *normal form*. Since we also assume that every object is in η-long form these normal forms are long $\eta\beta\delta$-normal forms. We write $M \overset{\mathrm{whr}}{\longrightarrow} M'$ for weak head reduction which applies local β- or δ-reductions.

We write $\Gamma \vdash M_1 \equiv M_2$ to express that two well-typed objects M_1 and M_2 are equivalent modulo $\beta\delta$-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 $\beta\delta$-conversion.

Property 1 (Equivalence).

1. $\Gamma \vdash M \equiv M$.

2. For all H_1, H_2 of the form 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. $\Gamma \vdash \Pi y{:}A_1.\, B_1 \equiv \Pi y{:}A_2.\, B_2$ iff $\Gamma \vdash A_1 \equiv A_2$ and $\Gamma, y : A_1 \vdash B_1 \equiv B_2$
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 $\Gamma \vdash S_1 \equiv S_2$
8. If $M_1 \xrightarrow{\text{whr}} M_1'$ and $M_2 \xrightarrow{\text{whr}} M_2'$ then $\Gamma \vdash M_1 \equiv M_2$ iff $\Gamma \vdash M_1' \equiv M_2'$

For a well-typed definition $d : A = M$ the head-normal form of M must always exist and have the shape $M = \lambda x_1 : A_1. \ldots \lambda x_n : A_n. H \cdot S$. We call $x_1, \ldots, x_n$ *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 propositional 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.

$$\begin{array}{lll}
 & & \text{o} \quad : \text{type} \\
\ulcorner \top \urcorner & = \text{true} & \text{true} : \text{o} \\
\ulcorner \bot \urcorner & = \text{false} & \text{false} : \text{o} \\
\ulcorner F_1 \supset F_2 \urcorner & = \text{imp} \cdot (\ulcorner F_1 \urcorner ; \ulcorner F_2 \urcorner) & \text{imp} \ : \text{o} \to \text{o} \to \text{o}
\end{array}$$

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

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

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

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

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

$$\begin{aligned}
&\mathrm{nd} &&: \mathrm{o} \to \mathrm{type}\\
&\mathrm{truei} &&: \mathrm{nd} \cdot \mathrm{true}\\
&\mathrm{falsee} &&: \mathit{\Pi} F\!:\!\mathrm{o}.\, \mathrm{nd} \cdot \mathrm{false} \to \mathrm{nd} \cdot F\\
&\mathrm{impi} &&: \mathit{\Pi} F\!:\!\mathrm{o}.\, \mathit{\Pi} G\!:\!\mathrm{o}.\, (\mathrm{nd} \cdot F \to \mathrm{nd} \cdot G) \to \mathrm{nd} \cdot (\mathrm{imp} \cdot (F; G))\\
&\mathrm{impe} &&: \mathit{\Pi} F\!:\!\mathrm{o}.\, \mathit{\Pi} G\!:\!\mathrm{o}.\, \mathrm{nd} \cdot (\mathrm{imp} \cdot (F; G)) \to \mathrm{nd} \cdot F \to \mathrm{nd} \cdot G
\end{aligned}$$

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

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

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.

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

4 Definitions and Algorithms for Equality

In this paper we study only notational definitions. 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 [HP98] — 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 a potential explosion size unless special implementation techniques are employed. Thirdly, expanding all definitions means that error messages and other output are often rendered 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

$$H \cdot S_1 \equiv H \cdot S_2 \tag{1}$$

into

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

but if $H = d$ and $d : A = M$ 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 Δ and spines S_1 and S_2,*

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

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 could be exponential for first-order terms and worse at higher types. 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). Our notion of strictness is a crude approximation of the notion of strictness found in functional programming.

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

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

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

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

holds even if T and T' are different.

Example 2 (Identity). The definition of the identity at function type, $\mathrm{id} = \lambda F\!:\!\mathrm{o} \to \mathrm{o}.\, \lambda G\!:\!\mathrm{o}.\, F \cdot G$, is not strict for two reasons: the only occurrence of F is at the head of the definition, and the only occurrence of G is as an argument to F. It is also not injective, because

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

can be reduced to

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

Example 3 (Identity at base type). The definition $\mathrm{id}' = \lambda F : \mathrm{o}.\, F$ is not strict since F occurs at the head of the definition. 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 (Application to constant). Consider $\mathrm{at} = \lambda F\!:\!\mathrm{o} \to \mathrm{o}.\, \mathrm{not} \cdot (F \cdot \mathrm{true})$. Note, that the argument to F is not a local parameter but a constant. The definition is hence not strict. The equality problem

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

can be expanded to

$$\begin{aligned}&(\lambda F\!:\!\mathrm{o} \to \mathrm{o}.\, \mathrm{not} \cdot (F \cdot \mathrm{true})) \cdot (\lambda F\!:\!\mathrm{o}.\, F)\\ &\qquad \equiv (\lambda F\!:\!\mathrm{o} \to \mathrm{o}.\, \mathrm{not} \cdot (F \cdot \mathrm{true})) \cdot (\lambda F\!:\!\mathrm{o}.\, \mathrm{true})\end{aligned}$$

which holds 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* [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:*

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

The formal system for strictness is defined by four 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. β-redices and δ-redices involving non-strict defined constants are reduced by $M \longrightarrow M'$.

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

$M \longrightarrow M'$	*M weak head-reduces to M'*
$\Gamma; \Delta \vdash_x M$	*x is locally strict in M*
$\Gamma \Vdash_x M$	*x is strict in M*
$\Gamma \Vdash M$	*M is strict*

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 strict definitions are injective. The proof is non-trivial and requires a sequence of properties sketched below.

Lemma 1 (Pattern spines). *Let S be a spine s.t. $\Delta \vdash S$ pat and M_1 and M_2 be objects valid in Γ disjoint from Δ.*

If $\Gamma, \Delta \vdash M_1 \cdot S \equiv M_2 \cdot S$ then $\Gamma \vdash M_1 \equiv M_2$

Proof. By induction over the derivation of $\Delta \vdash S$ pat.

Using inductions over local, relative, and global strictness, we can then show the completeness direction of our claim for strict $d : A = M$:

$$\Gamma \vdash M \cdot S_1 \equiv M \cdot S_2 \quad \text{implies} \quad \Gamma \vdash S_1 \equiv S_2.$$

We cannot prove this directly by induction, but must generalize to the following lemma which requires substitutions σ. We use standard notation for substitutions, which must always be the identity on local parameters (usually declared

$$\frac{d : A = M \in \Sigma \quad \cdot \nVdash M}{d \cdot S \longrightarrow M \cdot S}\,\mathsf{nr\_delta} \qquad \frac{M \cdot S \longrightarrow_{\beta} M'}{M \cdot S \longrightarrow M'}\,\mathsf{nr\_beta}$$

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

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

$$\frac{M \longrightarrow M' \quad \Gamma; \Delta \vdash_x M'}{\Gamma; \Delta \vdash_x M}\,\mathsf{ls\_red} \qquad \frac{d : A = M \in \Sigma \quad \cdot \Vdash M \quad \Gamma; \Delta \vdash_x S}{\Gamma; \Delta \vdash_x d \cdot S}\,\mathsf{ls\_d}$$

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

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

no rule for $\Gamma; \Delta \vdash_x y \cdot S$
for $x \neq y$, $y : A \in \Gamma$

$$\frac{\Gamma; \Delta \vdash_x M}{\Gamma; \Delta \vdash_x M; S}\,\mathsf{ls\_hd} \qquad \frac{\Gamma; \Delta \vdash_x S}{\Gamma; \Delta \vdash_x M; S}\,\mathsf{ls\_sp}$$

$$\frac{M \longrightarrow M' \quad \Gamma \Vdash_x M'}{\Gamma \Vdash_x M}\,\mathsf{rs\_red} \qquad \frac{d : A = M \in \Sigma \quad \cdot \Vdash M \quad \Gamma; \cdot \vdash_x d \cdot S}{\Gamma \Vdash_x d \cdot S}\,\mathsf{rs\_d}$$

$$\frac{\Gamma; \cdot \vdash_x c \cdot S}{\Gamma \Vdash_x c \cdot S}\,\mathsf{rs\_c} \qquad \frac{\Gamma, y : A \Vdash_x M}{\Gamma \Vdash_x \lambda y{:}A.\, M}\,\mathsf{rs\_lam}$$

$$\frac{M \longrightarrow M' \quad \Gamma \Vdash M'}{\Gamma \Vdash M}\,\mathsf{gs\_red} \qquad \frac{d : A = M \in \Sigma \quad \Gamma \Vdash M \cdot S}{\Gamma \Vdash d \cdot S}\,\mathsf{gs\_d}$$

$$\frac{}{\Gamma \Vdash c \cdot S}\,\mathsf{gs\_c} \qquad \frac{\Gamma, x : A \Vdash_x M \quad \Gamma, x : A \Vdash M}{\Gamma \Vdash \lambda x{:}A.\, M}\,\mathsf{gs\_lam}$$

Fig. 1. A formal system for strictness

in Δ). Because of possible dependencies, a substitution which maps variables in Γ to objects with variables in Γ' will map a parameter context Δ to a context Δ' where each declaration $y : A$ in Δ is mapped to $y : A[\sigma]$. We write $\Gamma'; \Delta' \vdash \sigma : \Gamma; \Delta$ for valid substitutions.

Lemma 2 (Completeness). *Let σ_1, σ_2 by substitutions which satisfy $\Gamma'; \Delta' \vdash \sigma_1 : \Gamma; \Delta$ and $\Gamma'; \Delta' \vdash \sigma_2 : \Gamma; \Delta$, respectively.*

1. *If $\Gamma; \Delta \vdash_x M$ and $\Gamma', \Delta' \vdash M[\sigma_1] \equiv M[\sigma_2]$ then $\Gamma' \vdash \sigma_1(x) \equiv \sigma_2(x)$.*
2. *If $\Gamma; \Delta \vdash_x S$ and $\Gamma', \Delta' \vdash S[\sigma_1] \equiv S[\sigma_2]$ then $\Gamma' \vdash \sigma_1(x) \equiv \sigma_2(x)$.*
3. *If $\Gamma \Vdash_x M$ and $\Gamma' \vdash M[\sigma_1] \cdot S \equiv M[\sigma_2] \cdot S$ then $\Gamma' \vdash \sigma_1(x) \equiv \sigma_2(x)$.*
4. *If $\Gamma \Vdash M$ and $\Gamma' \vdash M[\sigma_1] \cdot S_1 \equiv M[\sigma_2] \cdot S_2$ then $\Gamma' \vdash S_1 \equiv S_2$.*

Proof. The four parts are proven by simultaneous induction over the given strictness derivations, using Lemma 1 and Property 1.

As an immediate corollary, strictness is a sufficient criteria for injectivity.

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

Proof. Using Lemma 2, part 4, for $\sigma_1 = \sigma_2 = id$

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 (ls_pat), it stops and signals success. If the algorithm comes to a defined and strict constant, it applies ls_d or rs_d, otherwise it expands the definition using ls_red or rs_red, respectively. The algorithm terminates for ls_red and rs_red, because definitions 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 ls_d, rs_d, and nr_delta. A minor variant of this algorithm has been implemented in the Twelf system [PS98].

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.

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, $\beta\delta$-convertibility) can be improved by using strictness. In the presence of meta-variables these observations can be generalized to unification. We write $\Psi; \Delta \vdash M_1 \approx M_2$ for a unification problem, where M_1, M_2 are well-typed objects of the same type which can contain meta-variables declared in Ψ. All other parameters which are not subject to instantiation are declared in Δ. So this corresponds to a $\exists\forall$ prefix of a unification problem.

Deciding 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 failure, yet another source of incompleteness. The following two examples show these situations.

Example 5 (Most-general unifier). Let $\mathrm{tr} : \mathrm{o} \to \mathrm{o} = \lambda F{:}\mathrm{o}.\, \mathrm{true}$ a definition, and X a meta variable. The unification problem $X : \mathrm{o}; \cdot \vdash \mathrm{tr} \cdot \mathrm{false} \approx \mathrm{tr} \cdot X$ has as solution $\Theta = \mathrm{false}/X$ if tr is not expanded. Obviously, this solution 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 $X : \mathrm{o}; \cdot \vdash X \approx \mathrm{tr} \cdot X$ 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, $\Theta = \mathrm{true}/X$.

Most unification algorithms decompose a unification problem of the form

$$\Psi; \Delta \vdash H \cdot S_1 \approx H \cdot S_2 \tag{4}$$

into

$$\Psi; \Delta \vdash S_1 \approx S_2 \tag{5}$$

where H is not a defined constant, otherwise they expand the definition. The unification algorithm for the higher-order pattern fragment [DHKP96] which is employed in Twelf follows the same technique. But strict definitions do not need to be be expanded since, because of injectivity, every unifier Θ of (4) is also a unifier of (5) and vice versa. This is expressed in the following theorem.

Theorem 2 (Most general unifiers). *Let $d : A = M$ be a strict definition. Then the unification problems*

$$\Psi; \Delta \vdash d \cdot S_1 \approx d \cdot S_2$$

and

$$\Psi; \Delta \vdash S_1 \approx S_2$$

have the same set of solutions.

Proof. Let Θ be a unifier, satisfying $\Psi'; \Delta' \vdash \Theta : \Psi; \Delta$.

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

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

In addition, we can extend this algorithm to also treat the occurs-check problem correctly: We say that $\Psi; \Delta \vdash X\ y_1\ ..\ y_k \approx M$, where X is defined in Ψ and $y_1, .., y_k$ are parameters in Δ, fails the *occurs-check* if X has a strict occurrence in M (not to be confused with a locally strict one). This is a generalization of Huet's original *rigid path* criterion for non-unifiability by allowing some arguments to X. Note also that this definition of occurs-check does not need to expand strict definitions. We show that unification problems which fail the occurs-check do not have a unifier.

Informally, one assumes a solution Θ for X and then counts the number of constructor and parameter occurrences in the normal form of $(X\ y_1\ ..\ y_k)[\Theta]$ and $M[\Theta]$ to arrive at a contradiction, a similar argument as in [Pfe91]. In addition, we make use of two further properties. First, rigid positions in the arguments are preserved under normalization, and second, meta-variables can never occur in the head position of these normal forms.

The proof of the first property is rather difficult because definitions can be nested. In our proof we resolve this problem by first showing the admissibility of eliminating definitions and then inductively normalize each defined constant starting from the inside out. We write $nf(M)$ for the normal form of an object M, based on $\beta\delta$-conversion.

Lemma 3 (Admissibility of eliminating definitions). *Let σ be substitution satisfying $\Gamma'; \Delta' \vdash \sigma : \Gamma; \Delta$. Furthermore, let x be in Γ, y in Γ', and M, S, σ in normal form.*

1. *If $\Gamma; \Delta \vdash_x M$, and $\Gamma'; \Delta' \vdash_y \sigma(x)$ then $\Gamma'; \Delta' \vdash_y nf(M[\sigma])$.*
2. *If $\Gamma; \Delta \vdash_x S$ and $\Gamma'; \Delta' \vdash_y \sigma(x)$ then $\Gamma'; \Delta' \vdash_y nf(S[\sigma])$.*
3. *If $\Gamma \Vdash_x M$ and $\Gamma'; \Delta' \vdash_y \sigma(x)$ then $\Gamma'; \Delta' \vdash_y nf(M[\sigma] \cdot S)$.*
4. *If $\Gamma \Vdash M$ and $\Gamma'; \Delta' \vdash_y S$ then $\Gamma'; \Delta' \vdash_y nf(M[\sigma] \cdot S)$.*

Proof. The four parts are proven by simultaneous induction over the given strictness derivations.

A direct consequence of the admissibility of eliminating definitions is that the property of being strict is preserved under normalization.

Lemma 4 (Eliminating definitions).

1. *If $\Gamma; \Delta \vdash_x M$ then $\Gamma; \Delta \vdash_x nf(M)$.*

2. *If $\Gamma; \Delta \vdash_x S$ then $\Gamma; \Delta \vdash_x nf(S)$.*
3. *If $\Gamma \Vdash_x M$ then $\Gamma \Vdash_x nf(M)$.*
4. *If $\Gamma \Vdash M$ then $\Gamma \Vdash nf(M)$.*

Proof. The proof proceeds by simultaneous induction over the given strictness derivations, using Lemma 3.

To arrive at the contradiction described above, we must ensure that the head of a definition is never a meta-variable (the head of a λ-term is defined as the head of its body). We call such objects *rigid*.

Definition 4 (Rigid objects). *An object M defined in Ψ, Δ, where parameters in Δ are not subject to instantiation, is called a* rigid object *iff $head(nf(M))$ is either a constant or a parameter defined in Δ.*

The head of a definition, no matter to which arguments it is applied, cannot be a meta-variable.

Lemma 5 (Head). *If $d : A = M$ is a strict definition ($\Gamma \Vdash M$), and σ a substitution with domain Γ, then $M[\sigma] \cdot S$ is a rigid object.*

Proof. By induction over the strictness derivation of $\Gamma \Vdash M$.

The other part of the argument involves counting the number of parameter and constructor occurrences in a term M which we write as $|M|$. It can be easily shown that this measure satisfies the following property on the unification problem in question.

Lemma 6 (Size). *Let $\Psi; \Delta \vdash X\ y_1\ ..\ y_k \approx M$ be a unification problem, where M is strict in X ($\Psi; \Delta \vdash_X M$), and Θ be a unifier. Then*

$$|nf((X\ y_1\ ..\ y_k)[\Theta])| \leq |nf(M[\Theta])|$$

Proof. By induction over the strictness derivation $\Psi; \Delta \vdash_X M$, using Lemma 4.

The third technical result of our paper can now be stated and proven: If a unification problem fails the occurs-check, it cannot have any unifiers.

Theorem 3 (Occurs-check). *Let M be a rigid object, and Ψ a context of free variables. Furthermore, let X occur strictly in M ($\Psi; \Delta \vdash_X M$). Then the unification problem*

$$\Psi, \Delta \vdash X\ y_1\ ..\ y_k \approx M$$

has no unifiers.

Proof. Assume the unification problem fails the occurs-check and has the unifier Θ. By Lemma 6, it follows that

$$|nf((X\ y_1\ ..\ y_k)[\Theta])| \leq |nf(M[\Theta])|$$

but because of Lemma 5 we can show, that

$$|nf((X\ y_1\ ..\ y_k)[\Theta])| < |nf(M[\Theta])|$$

contradicting the assumption that Θ is a unifier.

Hence, a unification problem which fails the occurs-check does not have any unifiers. 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).

Therefore, 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. Notational definitions must be expanded only in the case of constant clash. This property can be exploited to make many implementations of these algorithms more efficient, while preserving completeness and soundness with respect to $\beta\delta$-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 in 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.

[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.

[HP98] Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*, 8(1):5–31, 1998. 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.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer-Verlag LNCS 828, 1994.

[Pfe91] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[PS98] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide*, 1.2 edition, September 1998. Available as Technical Report CMU-CS-98-173, Carnegie Mellon University.