# Verifying Uniqueness in a Logical Framework

Penny Anderson[1] and Frank Pfenning[2]*

[1] penny@cs.lafayette.edu, Department of Computer Science, Lafayette College
[2] fp@cs.cmu.edu, Department of Computer Science, Carnegie Mellon University

**Abstract.** We present an algorithm for verifying that some specified arguments of an inductively defined relation in a dependently typed $\lambda$-calculus are uniquely determined by some other arguments. We prove it correct and also show how to exploit this uniqueness information in coverage checking, which allows us to verify that a definition of a function or relation covers all possible cases. In combination, the two algorithms significantly extend the power of the meta-reasoning facilities of the Twelf implementation of LF.

## 1 Introduction

In most logics and type theories, unique existence is not a primitive notion, but defined via existence and equality. For example, we might define $\exists! x.A(x)$ to stand for $\exists x.A(x) \wedge \forall y.A(y) \supset x = y$. Such definitions are usually made in both first-order and higher-order logic, and in both the intuitionistic and the classical case. Expanding unique existence assertions in this manner comes at a price: not only do we duplicate the formula $A$, but we also introduce two quantifiers and an explicit equality. It is therefore natural to ask if we could derive some benefit for theorem proving by taking unique existence as a primitive.

In this paper we consider an instance of this problem, namely verifying and exploiting uniqueness in a logical framework. We show how to establish uniqueness of certain arguments to type families in the logical framework LF [7] as implemented in the Twelf system [15]. We further show how to exploit this uniqueness information to verify meta-theoretic properties of signatures, thereby checking proofs of meta-theorems presented as relations in LF. In particular, we can automatically verify the unique existence of specified output arguments in a relation with respect to some given input arguments. Our algorithm will always terminate, but, since the problem is in general undecidable, will sometimes fail to establish uniqueness even though it holds.

Our algorithm extends prior work on coverage checking [24] and mode checking [18], which in combination with termination checking [16], can verify meta-theoretic proofs such as cut elimination [12], the Church-Rosser theorem [19], logical translations [13], or the soundness of Foundational Typed Assembly Language [3, 4]. The specific motivation for this work came mostly from the latter,

* This research has been supported by NSF Grant CCR-0306313.
To appear at TPHOLS'04

in which a significant portion of the development was devoted to tedious but straightforward reasoning about equality. Our algorithms can automate much of that.

We believe that our techniques can be adapted to other systems of constructive type theory to recognize properties of relations. In that direction, the research can be seen as an extension of the work by McBride [11] and Coquand [2], who present procedures for deciding whether a definition by pattern matching of a dependently typed function consists of cases that are exhaustive and mutually exclusive. Here, we permit not only inputs containing abstractions, but also relational specifications, which are pervasive and unavoidable in constructive type theories. Like the prior work on functions, but unlike prior work on coverage checking [19, 24], we can also verify uniqueness and unique existence.

The remainder of the paper is organized as follows. In Section 2 we briefly introduce the notation of the LF type theory used throughout. In Section 3 we describe our algorithm for verifying uniqueness of specified arguments to relations, and we prove its correctness in Section 4. In Section 5 we briefly review coverage checking, one of the central algorithms in verifying the correctness of meta-theoretic proofs. In Section 6 we show how to exploit uniqueness information to increase the power of coverage checking. We conclude in Section 7 with some further remarks about related and future work.

## 2 The LF Type Theory

We use a standard formulation of the LF type theory [7]; we summarize here only the basic notations. We use $a$ for type families, $c$ for object-level constants, and $x$ for (object-level) variables. We say *term* to refer to an expression from any of the three levels of kinds, types, and objects.

$$\begin{array}{rr@{\;::=\;}l}
\text{Kinds} & K & \text{type} \mid \Pi x{:}A.K \\
\text{Types} & A, B & a\, M_1 \ldots M_n \mid \Pi x{:}A.B \\
\text{Objects} & M, N & c \mid x \mid \lambda x{:}A.M \mid M\, N \\[4pt]
\text{Signatures} & \Sigma & \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A \\
\text{Contexts} & \Gamma, \Delta & \cdot \mid \Gamma, x{:}A \\
\text{Substitutions} & \theta, \sigma & \cdot \mid \theta, M/x
\end{array}$$

Contexts and substitutions may declare a variable at most once; signatures may declare families and constants at most once. We do not distinguish terms from any of the three levels that differ only in the names of their bound variables. Our notion of definitional equality is $\beta\eta$-conversion, and we tacitly exploit the property that every kind, type, and object has a unique long $\beta\eta$-normal form [1, 8] which we call *canonical*. The relatively simple nature of this definitional equality avoids some thorny issues regarding intensional and extensional equality in constructive type theories [10, 9] that would complicate our analysis. We omit type-level $\lambda$-abstractions from the syntax since they do not occur in canonical forms. The principal judgments we use are:

| | |
|---|---|
| $\Gamma \vdash_\Sigma A : \text{type}$ | Type $A$ is valid |
| $\Gamma \vdash_\Sigma M : A$ | Object $M$ has type $A$ |
| $\Gamma \vdash_\Sigma \theta : \Delta$ | Substitution $\theta$ matches context $\Delta$ |

Since all judgments are standard we only show the last one for typing substitutions which is perhaps less widely known. We write $M[\theta]$ and $A[\theta]$ for the application of a substitution.

$$\frac{}{\Gamma \vdash \cdot : \cdot} \qquad \frac{\Gamma \vdash \theta : \Delta \qquad \Gamma \vdash M : A[\theta]}{\Gamma \vdash (\theta, M/x) : (\Delta, x{:}A)}$$

So a substitution $\Gamma \vdash \theta : \Delta$ maps a term defined over a context $\Delta$ to a term over a context $\Gamma$. We write $\theta_1 \circ \theta_2$ for composition of substitutions, so that $M[\theta_1][\theta_2] = M[\theta_1 \circ \theta_2]$, and write $\mathrm{id}_\Gamma$ for the identity substitution $\Gamma \vdash \mathrm{id}_\Gamma : \Gamma$.

As a running example we use natural numbers defined in terms of zero (z) and successor (s), together with relations for inequality (le) and addition (plus).[1] The corresponding signature is given in Figure 1. Note that free variables in a declaration are implicitly universally quantified in that declaration; the Twelf implementation will reconstruct these quantifiers and the types of the free variables [15].

```
nat : type.
z : nat.
s : nat → nat.
le : nat → nat → type.
le_refl : le X X.
le_s : le X Y → le X (s Y).
plus : nat → nat → nat → type.
plus_z : plus z X X.
plus_s : plus X₁ X₂ Y → plus (s X₁) X₂ (s Y).
```

**Fig. 1.** Natural numbers with ordering and addition

## 3 Uniqueness Mode Checking

Logical frameworks that support higher-order abstract syntax, such as LF or hereditary Harrop formulas, are based on a simply typed or dependently typed

[1] This running example does not illustrate the higher-order nature of our analysis, but unfortunately space constraints do not permit us to include larger and more realistic examples. However, we have executed the uniqueness checker against higher-order examples drawn from [3, 4].

$\lambda$-calculus. Function spaces in such a calculus are purposely impoverished in order to support the use of meta-language functions to represent object language abstractions and hypothetical proofs: too many such functions would invalidate the judgments-as-types or judgments-as-propositions methodology. In particular, these frameworks prohibit function definitions by cases or by primitive recursion. Adding such functions appears to require modal types or an explicit stratification of the type theory [5, 23, 20, 21]; related approaches are still a subject of current research (see, for example, [25, 22]).

The traditional and practically tested approach is to represent more complex functions as either type families or relations, depending on whether the framework is a type theory or a logic.[2] In many cases relational representations of functions are sufficient, but there are also many instances where meta-reasoning requires us to know that relations do indeed represent (possibly partial) functions. We can encode this property by defining explicit equality relations. For example, if we need to know that the relation plus is actually a function of its first two arguments, we can define

```
eq : nat → nat → type.
refl : eq X X.
```

We then have to prove: "*If* plus $X_1$ $X_2$ $Y$ *and* plus $X_1$ $X_2$ $Y'$ *then* eq $Y$ $Y'$."

There are two difficulties with this approach: the first is simply that equality predicates need to be threaded through many judgments, and various rather trivial and tedious properties need to be proved about them. The second is that this methodology interferes with dependent typing because the equality between $Y$ and $Y'$ in the example above cannot be exploited by type-checking, since eq is just a user-declared relation.

As uses of the meta-reasoning capabilities of the logical framework become increasingly complex [3, 4], intrinsic support for recognizing and exploiting relations that are indeed functions is becoming more and more important. There are two distinct, interconnected problems to be solved. The first is to verify that particular relations are partial functions. The second is to exploit this information to verify that the same or other relations are total.

In this section we address the former: how can we automatically verify that particular relations are partial functions of some of their inputs. This is a stricter version of *mode checking* familiar from logic programming. There, we designate some arguments to a relation as inputs and others as outputs. The property we verify with mode checking is that if the inputs are given as ground terms, and proof search succeeds, then the outputs will also be ground terms [18]. The sharpened version requires in addition that if proof search succeeds, then some designated outputs are uniquely determined. We refer to this process as *uniqueness mode checking*.

[2] Even though we are working in the LF type theory, we will use the terms *type family*, *relation*, and *predicate* interchangeably, expressing the intended meaning of the type families under consideration.

In our applications we actually need to exploit a slightly stronger property: if the designated input arguments to a relation are given as ground terms, then designated output arguments must be ground and uniquely determined, *independent of the proof search strategy*. In other words, our analysis must be based on a non-deterministically complete proof search strategy, rather than depth-first logic program execution (which is incomplete).

We use terminology from logic programming in the description of our algorithm below.

For the sake of simplicity we restrict the relations for which we verify uniqueness to consist of Horn clauses, which means the relations we analyze are inductively defined. However, the domains of quantification in the clauses are still arbitrary LF terms, which may be dependently typed and of arbitrary order.

In our syntax for the Horn fragment, we refer to a constant declaration that is to be analyzed as a *clause*. We group dependently occurring arguments into a quantifier prefix $\Pi\Gamma$ and the non-dependent arguments into a conjunction of *subgoals* $G$. We call the atomic type $Q$ the *head* of a clause $c : \Pi\Gamma.\, G \rightarrow Q$. We sometimes refer to a term with free variables that are subject to unification as a *pattern*. All constructors for type families $a$ appearing at the head of an atomic goal in a program must also be part of the program and satisfy the Horn clause restrictions.

$$\begin{array}{ll} \textit{Atomic Goals} & Q ::= a\ M_1 \ldots M_n \\ \textit{Goals} & G ::= Q \mid G_1 \wedge G_2 \mid \top \\ \textit{Clauses} & D ::= c : \Pi\Gamma.\, G \rightarrow Q \\ \textit{Programs} & \mathcal{P} ::= D_1, \ldots, D_n \end{array}$$

In the implementation, we do not make this restriction and instead analyze arbitrary LF signatures, enriched with *world declarations* [19]. A description and correctness proof of this extension is the subject of current research and beyond the scope of this paper.

*Mode declarations.* In order to verify mode properties of relations, we specify each argument of a relation to be either an input (`+`), an output (`-`), a unique output (`-1`), or unmoded (`*`). Intuitively, the declarations are tied to a non-deterministic proof search semantics and express:

> If all input (`+`) arguments to a predicate are ground when it is invoked, and search succeeds, then all output arguments are ground (`-`). Moreover, in all successful proofs, corresponding unique outputs (`-1`) must not only be ground, but equal. Unmoded arguments remain unconstrained.

Mode information for a type family $a$ is reflected in the functions $\mathrm{ins}(a)$, $\mathrm{outs}(a)$, and $\mathrm{uouts}(a)$, returning the sets of indices for the input arguments, output arguments, and unique output arguments respectively.

In our example, the following declarations would be correct:

```
%mode le +X -Y.
%mode plus +X1 +X2 -1Y.
```

The first one expresses that if a goal of the form le $M$ $Y$ for a ground term $M$ succeeds, then $Y$ must also be ground. The second one expresses that every successful search for a proof of plus $M_1$ $M_2$ $Y$ with ground $M_1$ and $M_2$ yields the same term $N$ for $Y$. In other words, plus represents a partial function from its first two arguments to its third argument. The second declaration yields $\text{ins}(\mathsf{plus}) = \{1, 2\}$, $\text{outs}(\mathsf{plus}) = \{\,\}$, and $\text{uouts}(\mathsf{plus}) = \{3\}$.

Our algorithm for uniqueness mode checking verifies two properties: disjointness of input arguments and uniqueness of output arguments.

*Disjointness of inputs.* For a given relation with some uniqueness modes on its output arguments, we verify that no two clause heads unify on their input arguments. This entails that any goal with ground input arguments unifies with no more than one clause head. As an example, consider the relation plus from Figure 1 with mode `plus +X1 +X2 -1Y`. Uniqueness mode checking verifies that plus z $X$ _ and plus (s $X_1$) $X_2$ _ do not have a unifier. This is easy because z and s in the first argument clash. We use the algorithm in [6] which will always terminate, but may sometimes generate constraints that cannot be solved. In that case, uniqueness mode checking will fail.

*Strictness.* Because we can make the assumption that input arguments are ground, what is most relevant to our analysis is not full unification, but higher-order dependently typed matching. Schürmann [19] has shown that each variable in a higher-order matching problem that has at least one strict occurrence has a unique, ground solution. An occurrence of a variable is *strict* if it is applied to distinct bound variables and it is not in an argument to another unification variable (see [14] for a more formal definition).

Strictness is central in our analysis to conclude that if matching a pattern against a ground term succeeds, variables with at least one strict occurrence in the pattern are guaranteed to be ground. In our specific situation, we actually employ unification of two types $a\,M_1 \ldots M_n \doteq a\,N_1 \ldots N_n$ where certain subproblems (for example, $M_i \doteq N_i$ for $i \in \text{ins}(a)$) are known to be matching problems.

*Checking uniqueness of outputs.* Uniqueness of outputs is verified by an abstract non-deterministic logic programming interpreter with left-to-right subgoal selection[3]. The domain used is the space of abstract substitutions with elements *unknown* (u), *ground* (g), and *unique* (q) for each variable $x$ where u carries no information and q the most information. Note that in order for a variable to be unique (q) it must also be ground. Variables of unknown status (u) may become known as ground (g) or unique (q) during analysis in the following situations:

– An unknown variable that occurs in a strict position in an input argument of the clause head becomes known to be unique.

[3] Left-to-right subgoal selection is convenient for this abstract interpretation, but not critical for its soundness.

| | |
|---|---|
| $\Psi \vdash Q^{+1} > \Psi'$ | all strict variables in inputs of $Q$ are $\mathsf{q}$ in $\Psi'$ |
| $\Psi \vdash Q^{-} > \Psi'$ | all strict variables in outputs of $Q$ are at least $\mathsf{g}$ in $\Psi'$ |
| $\Psi \vdash Q^{-1} > \Psi'$ | all strict variables in unique outputs of $Q$ are $\mathsf{q}$ in $\Psi'$ |
| $\Psi \vdash Q^{+1}$ | if all variables in inputs of $Q$ are $\mathsf{q}$ |
| $\Psi \vdash Q^{+}$ | if all variables in inputs of $Q$ are at least $\mathsf{g}$ |
| $\Psi \vdash Q^{-1}$ | if all variables in unique outputs of $Q$ are $\mathsf{q}$ |
| $\Psi \vdash Q^{-}$ | if all variables in outputs of $Q$ are at least $\mathsf{g}$ |

**Fig. 2.** Judgments on abstract substitutions

– An unknown variable becomes known to be ground if it occurs in a strict position in the output of a subgoal all of whose inputs are known to be ground or unique.
– An unknown or ground variable becomes known to be unique if it occurs in a strict position in a unique output of a subgoal all of whose inputs are known to be unique.

We next describe in detail the uniqueness mode checking for the Horn clause fragment of LF. The checker relies on two sets of judgments on *abstract substitutions*, which provide reliable, though approximate, information about the actual substitution at any point during search for a proof of a goal. The corresponding non-deterministic search strategy is explained in Section 4.

$$\begin{array}{ll} \textit{Abstract objects} & \mu ::= \mathsf{u} \mid \mathsf{g} \mid \mathsf{q} \\ \textit{Abstract substitutions} & \Psi ::= \cdot \mid \Psi, \mu/x \end{array}$$

The first set of judgments have the form $\Psi \vdash Q^m > \Psi'$ where $\Psi$ is an abstract substitution with known information, $Q$ is an atomic predicate $a\, M_1 \dots M_n$, $m$ indicates which arguments to $a$ are analyzed, and $\Psi'$ is the result of the analysis of $Q$. Both $\Psi$ and $\Psi'$ will be defined on the free variables in $Q$. Moreover, $\Psi'$ will always contain the same or more information than $\Psi$.

The second set of judgments $\Psi \vdash Q^m$ hold if $Q$ satisfies a property specified by $m$ given the information in $\Psi$. Again, if $\Gamma \vdash Q : \text{type}$, then $\Psi$ will be defined on the variables in $\Gamma$. The various forms of these judgments are given in Figure 2.

The judgments on abstract substitutions are employed by the uniqueness mode checker, which is itself based on two judgments: $\vdash c : \Pi\Gamma.G \to P$ for checking clauses in the program, and $\Psi \vdash G > \Psi'$ for analyzing goals $G$, where $\Psi'$ may contain more information than $\Psi$ and both $\Psi$ and $\Psi'$ are defined on the free variables of $G$.

The mode checker is defined by the inference rules of Figure 3. We view these rules as an algorithm for mode checking by assuming $\Psi$ and $G$ to be given, and constructing $\Psi'$ such that $\Psi \vdash G > \Psi'$, searching for derivations of the premises from first to last. We write $\Psi(\Gamma)$ for the abstract context corresponding to the (concrete) context $\Gamma$, where each variable is marked as unknown ($\mathsf{u}$).

**Definition 1 (Mode correct programs).** *Given a program $\mathcal{P}$, we write $\mathcal{P}(a)$ for the set of clauses in $\mathcal{P}$ with $a$ as head. We say $\mathcal{P}$ is* mode-correct *if*

$$\frac{\begin{array}{l}\Psi(\Gamma) \vdash P^{+1} > \Psi_1 \\ \Psi_1 \vdash G > \Psi_2 \\ \Psi_2 \vdash P^{-} \\ \Psi_2 \vdash P^{-1}\end{array}}{\vdash c : \Pi\Gamma.G \to P} \qquad \frac{\begin{array}{l}\Psi_0 \vdash Q^{+1} \\ \Psi_0 \vdash Q^{-} > \Psi_1 \\ \Psi_1 \vdash Q^{-1} > \Psi_2\end{array}}{\Psi_0 \vdash Q > \Psi_2} \qquad \frac{\begin{array}{l}\Psi_0 \vdash Q^{+} \\ \Psi_0 \vdash Q^{-} > \Psi_1\end{array}}{\Psi_0 \vdash Q > \Psi_1}$$

$$\frac{\begin{array}{l}\Psi_0 \vdash G_1 > \Psi_1 \\ \Psi_1 \vdash G_2 > \Psi_2\end{array}}{\Psi_0 \vdash G_1 \wedge G_2 > \Psi_2} \qquad \frac{}{\Psi_0 \vdash \mathsf{T} > \Psi_0}$$

**Fig. 3.** Uniqueness mode checking

1. *For every type family $a$ in $\mathcal{P}$, if $a$ is declared to have unique outputs, then for any two distinct $c_1 : \Pi\Gamma_1. G_1 \to Q_1$ and $c_2 : \Pi\Gamma_2. G_2 \to Q_2$ in $\mathcal{P}(a)$, $Q_1$ and $Q_2$ are not unifiable on their inputs.*
2. *For every constant $c$ declared in $\mathcal{P}$, we have $\vdash c : \Pi\Gamma. G \to P$*

Part (2) of the definition requires each predicate to have a mode declaration, but we may default this to consider all arguments unmoded (`*`) if none is given.

As an example, consider once again the plus predicate from Figure 1 with mode `plus +X1 +X2 -1Y`. We have to check clauses plus_z and plus_s. We present the derivations in linear style, eliding arguments to predicates that are ignored in any particular judgments.

$$\frac{\begin{array}{l}\mathsf{u}/X \vdash (\mathsf{plus}\ \mathsf{z}\ X\ \_)^{+1} > \mathsf{q}/X \\ \mathsf{q}/X \vdash \mathsf{T} > \mathsf{q}/X \\ \mathsf{q}/X \vdash (\mathsf{plus}\ \_\ \_\ \_)^{-} \\ \mathsf{q}/X \vdash (\mathsf{plus}\ \_\ \_\ X)^{-1}\end{array}}{\vdash \mathsf{plus\_z} : \Pi X{:}\mathsf{nat}.\, \mathsf{T} \to \mathsf{plus}\ \mathsf{z}\ X\ X}$$

$$\frac{\begin{array}{l}\mathsf{u}/X_1, \mathsf{u}/X_2, \mathsf{u}/Y \vdash (\mathsf{plus}\ (\mathsf{s}\ X_1)\ X_2\ \_)^{+1} > \mathsf{q}/X_1, \mathsf{q}/X_2, \mathsf{u}/Y \\ \qquad \mathsf{q}/X_1, \mathsf{q}/X_2, \mathsf{u}/Y \vdash (\mathsf{plus}\ X_1\ X_2\ \_)^{+1} \\ \qquad \mathsf{q}/X_1, \mathsf{q}/X_2, \mathsf{u}/Y \vdash (\mathsf{plus}\ \_\ \_\ \_)^{-} > \mathsf{q}/X_1, \mathsf{q}/X_2, \mathsf{u}/Y \\ \qquad \mathsf{q}/X_1, \mathsf{q}/X_2, \mathsf{u}/Y \vdash (\mathsf{plus}\ \_\ \_\ Y)^{-1} > \mathsf{q}/X_1, \mathsf{q}/X_2, \mathsf{q}/Y \\ \mathsf{q}/X_1, \mathsf{q}/X_2, \mathsf{u}/Y \vdash \mathsf{plus}\ X_1\ X_2\ Y > \mathsf{q}/X_1, \mathsf{q}/X_2, \mathsf{q}/Y \\ \mathsf{q}/X_1, \mathsf{q}/X_2, \mathsf{q}/Y \vdash (\mathsf{plus}\ \_\ \_\ \_)^{-} \\ \mathsf{q}/X_1, \mathsf{q}/X_2, \mathsf{q}/Y \vdash (\mathsf{plus}\ \_\ \_\ (\mathsf{s}\ Y))^{-1}\end{array}}{\vdash \mathsf{plus\_s} : \Pi X_1{:}\mathsf{nat}.\, \Pi X_2{:}\mathsf{nat}.\, \Pi Y{:}\mathsf{nat}.\, \mathsf{plus}\ X_1\ X_2\ Y \to \mathsf{plus}\ (\mathsf{s}\ X_1)\ X_2\ (\mathsf{s}\ Y).}$$

## 4 Correctness of Uniqueness Mode Checking

We next define a non-deterministic operational semantics for the Horn fragment and show that uniqueness mode checking approximates it. The judgment has the form $\theta \models G > \theta'$, where $\theta$ and $\theta'$ are substitutions for the free variables in goal $G$. We think of $\theta$ and goal $G$ as given and construct a derivation and substitution $\theta'$.

The semantics is given by the system of Figure 4. In the first rule, $\sigma$ and $\theta_1$ represent substitutions that unify $P$ and $Q[\theta_0]$. The fact that these substitutions may not be computable, or that they may not be most general, does not concern us here, since uniqueness mode checking guarantees that *any* unifier must ground all variables in $\Gamma$ that have a strict occurrence in the input arguments of $P$, provided the input arguments of $Q[\theta_0]$ are ground.

$$\frac{\Pi\Gamma.G \to P \in \mathcal{P} \qquad P[\sigma] = Q[\theta_0][\theta_1] \qquad \sigma \models G > \theta_2}{\theta_0 \models Q > \theta_0 \circ \theta_1 \circ \theta_2}$$

$$\frac{\theta_0 \models G_1 > \theta_1 \qquad \theta_1 \models G_2 > \theta_2}{\theta_0 \models G_1 \wedge G_2 > \theta_2} \qquad \frac{}{\theta_0 \models \top > \theta_0}$$

**Fig. 4.** Operational semantics

**Definition 2 (Approximation).** *We define when an abstract substitution approximates a set of substitutions as follows: Given an abstract substitution $\Psi : \Gamma$ and a set $\Theta$ of substitutions $\Gamma_i' \vdash \theta_i : \Gamma$, we say $\Psi$ approximates $\Theta$ ($\Psi \prec \Theta$) if for every $x$ in the domain of $\Psi$*

1. *if $\Psi(x) = \mathsf{g}$ then for all $\theta_i \in \Theta$, $\theta_i(x)$ is ground, and*
2. *if $\Psi(x) = \mathsf{q}$ then for some ground term $M$ and all $\theta_i \in \Theta$, $\theta_i(x) = M$.*

**Lemma 1 (Soundness of uniqueness mode checking).** *Let $\mathcal{P}$ be a mode-correct program, $G$ a goal, $\Psi : \Gamma$ an abstract substitution such that $\Psi \vdash G > \Psi'$, and $\Theta$ a set of substitutions. If $\Psi \prec \Theta$ then $\Psi' \prec \{\rho \mid \theta \models G > \rho, \theta \in \Theta\}$.*

*Proof.* We let $D$ be the set of all derivations of $\theta \models G > \rho$ for all $\theta \in \Theta$. We show by induction on pairs $(d, d')$ of derivations in $D$, where $d$ derives $\theta \models G > \rho$ and $d'$ derives $\theta' \models G > \rho'$, that if $\Psi \prec \{\theta, \theta'\}$ then $\Psi' \prec \{\rho, \rho'\}$. Since $d, d'$ are arbitrary the lemma follows for the whole set.

The only nontrivial case is that of an atomic goal $Q$ where the mode checking derivation for $Q$ has the form

$$\frac{\begin{array}{ll} \Psi_0 \vdash Q^{+1} & \text{(input variables of } Q \text{ must be mapped to } \mathsf{q}) \\ \Psi_0 \vdash Q^{-} > \Psi_1 & \text{(output variables of } Q \text{ are mapped to } \mathsf{g}) \\ \Psi_1 \vdash Q^{-1} > \Psi_2 & \text{(unique output variables of } Q \text{ are mapped to } \mathsf{q}) \end{array}}{\Psi_0 \vdash Q > \Psi_2}$$

The two derivations $d$ and $d'$ have the form

$$\frac{\begin{array}{l} \Pi\Gamma.G \to P \in \mathcal{P} \\ P[\sigma] = Q[\theta_0][\theta_1] \\ \sigma \models G > \theta_2 \end{array}}{\theta_0 \models Q > \theta_0 \circ \theta_1 \circ \theta_2} \qquad \frac{\begin{array}{l} \Pi\Gamma.G' \to P' \in \mathcal{P} \\ P'[\sigma'] = Q[\theta'_0][\theta'_1] \\ \sigma' \models G' > \theta'_2 \end{array}}{\theta'_0 \models Q > \theta'_0 \circ \theta'_1 \circ \theta'_2}$$

Write $\theta_{out}$ for $\theta_0 \circ \theta_1 \circ \theta_2$ and $\theta'_{out}$ for $\theta'_0 \circ \theta'_1 \circ \theta'_2$.

It is easy to see, for each input variable $x$ of $Q$, that $\theta_{out}(x) = \theta'_{out}(x) = \theta_0(x) = \theta'_0(x)$, so the approximation relation is satisfied for the input variables of $Q$.

For the output variables, there are two subcases: either there is uniqueness information for the type family of $Q$, so that only one clause head can match $Q$, or there is no uniqueness information.

For the first subcase $P = P'$ and $G = G'$. We use the mode correctness of the program to obtain the subgoal mode check $\Psi'_1 \vdash G > \Psi'_2$, where $\Psi'_2$ enforces the mode annotations for the input and output variables of $P$. $\Psi'_1 \prec \{\sigma, \sigma'\}$, so by induction $\Psi'_2 \prec \{\theta_2, \theta'_2\}$. Then $\theta_{out}$ and $\theta'_{out}$ satisfy the mode annotations for the output variables of $Q$, as required.

For the second subcase the reasoning is similar, but there are no output uniqueness requirements and more than one clause head can match $Q$. □

**Lemma 2 (Completeness of non-deterministic search).** *Given $\Delta \vdash Q$ : type. If $Q$ contains only ground terms in its input positions, and there is a substitution $\theta$ and term $M$ such that $\cdot \vdash M : Q[\theta]$, then $\mathrm{id}_\Delta \models Q > \theta'$ and there is a substitution $\theta''$ such that $\theta = \theta' \circ \theta''$.*

*Proof.* The proof is standard, using induction on the structure of $M$, exploiting the non-deterministic nature of the operational semantics to guess the right clauses and unifying substitutions. □

## 5 Coverage

Coverage checking is the problem of deciding whether any closed term of a given type is an instance of at least one of a given set of patterns. Our work on exploiting uniqueness information in coverage checking is motivated by its application to proof assistants and proof checkers, where it can be used to check that all possible cases in the definition of a function or relation are covered. The coverage problem and an approximation algorithm for coverage checking in LF are described in [24], extending prior work by Coquand [2] and McBride [11].

More precisely, a coverage problem is given by a coverage goal and a set of patterns. In our setting it is sufficient to consider coverage goals that are types with free variables $\Delta \vdash A$ : type; it is straightforward to translate general coverage goals to this form.

**Definition 3 (Immediate Coverage).** *We say a coverage goal* $\Delta \vdash A : \mathrm{type}$ *is* immediately covered *by a collection of patterns* $\Delta_i \vdash A_i : \mathrm{type}$ *if there is an* $i$ *and a substitution* $\Delta \vdash \sigma_i : \Delta_i$ *such that* $\Delta \vdash A \equiv A_i[\sigma_i] : \mathrm{type}$*.*

Coverage requires immediate coverage of every ground instance of a goal.

**Definition 4 (Coverage).** *We say* $\Delta \vdash A : \mathrm{type}$ *is* covered *by a collection of patterns* $\Delta_i \vdash A_i : \mathrm{type}$ *if every ground instance* $\cdot \vdash A[\tau] : \mathrm{type}$ *for* $\cdot \vdash \tau : \Delta$ *is immediately covered by the collection* $\Delta_i \vdash A_i : \mathrm{type}$*.*

As an example, consider again the plus predicate from Figure 1. We have already shown that the output of plus, if it exists, is unique. In order to show that plus is a total function of its first two arguments, we need to show that it always terminates (which is easy—see [18, 16]), and that the inputs cover all cases. For the latter requirement, we transform the signature into coverage patterns by eliding the outputs:

```
X:nat ⊢ plus z X _.
X₁:nat, X₂:nat  ⊢ plus (s X₁) X₂ _.
```

The coverage goal:

```
Y₁:nat, Y₂:nat ⊢plus Y₁ Y₂ _.
```

In this example, the goal is covered by the two patterns since every ground instance of the goal plus $M_1$ $M_2$ _ will be an instance of one of the two patterns. However, the goal is not immediately covered because $Y_1$ clashes with z in the first pattern and s in the second.

When a goal $\Delta \vdash A : \mathrm{type}$ is not immediately covered by any pattern, the algorithm makes use of an operation called *splitting*, which produces a set of new coverage goals by partially instantiating free variables in $\Delta$. Each of the resulting goals is covered if and only if the original goal is covered. Intuitively, splitting works by selecting a variable $u$ in $\Delta$, and instantiating it to all possible top-level structures based on its type.

In the example, the clashes of $Y_1$ with z and s suggest splitting of $Y_1$, which yields two new coverage goals

```
Y:nat ⊢ plus z Y _.
Y₁:nat, Y₂:nat ⊢ plus (s Y₁) Y₂ _.
```

These are immediately covered by the first and second pattern, respectively, but in general many splitting operations may be necessary.

The process of repeated splitting of variables in goals that are not yet covered immediately will eventually terminate according to the algorithm in [24], namely when the failed attempts to immediately cover a goal no longer suggest any promising candidates for splitting. Unfortunately, this algorithm is by necessity incomplete, since coverage is in general an undecidable property. Sometimes, this is due to a variable $x{:}B$ in a coverage goal which has no ground instances, in which case the goal is vacuously covered. Sometimes, however, the coverage

```
preserv : plus X₁ X₃ Y → plus X₂ X₃ Y′ → le X₁ X₂ → le Y Y′ → type.
preserv_refl : preserv S₁ S₂ le_refl le_refl.
preserv_s : preserv S₁ S₂ L L′ → preserv S₁ (plus_s S₂) (le_s L) (le_s L′).
```

**Fig. 5.** Addition preserves ordering

checker reaches a situation where several terms must be equal in order to obtain immediate coverage. It is in these situations that uniqueness information can help, as we explain in the next section.

## 6 Uniqueness in Coverage

We begin with an example that demonstrates failure of coverage due to the absence of uniqueness information.

Given type families for natural numbers, addition, and ordering, a proof that addition of equals preserves ordering can be encoded as the relation preserv in Figure 5. Note that, as before, free variables are implicitly quantified on each clause. Moreover, arguments to type families whose quantifiers were omitted earlier (as, for example, $\Pi X{:}\mathsf{nat}$ in the clause $\mathsf{le\_refl} : \mathsf{le} X X$) are also omitted, and determined by type reconstruction as in the Twelf implementation [15].

In order to verify that preserv constitutes a meta-theoretic proof, we need to verify that for all inputs $S_1 : \mathsf{plus}\ X_1\ X_3\ Y$, $S_2 : \mathsf{plus}\ X_2\ X_3\ Y'$, and $L : \mathsf{le}\ X_1\ X_2$ there exists an output $L' : \mathsf{le}\ Y\ Y'$ which witnesses that $x_1 + x_3 \leq x_2 + x_3$ if $x_1 \leq x_2$.

The initial coverage goal has the form

```
X₁:nat, X₂:nat, X₃:nat, Y:nat, Y′:nat,
      S₁:plus X₁ X₃ Y, S₂:plus X₂ X₃ Y′, L:le X₁ X₂ ⊢ preserv S₁ S₂ L _.
```

This fails, and after one step of splitting on the variable $L$ we obtain two cases, the second of which is seen to be covered by the preserv_s clause after one further splitting step, while the first has the form

```
X₁:nat, X₃:nat, Y:nat, Y′:nat, S₁:plus X₁ X₃ Y, S₂:plus X₁ X₃ Y′.
      ⊢ preserv S₁ S₂ le_refl _.
```

The clause preserv_refl does not immediately cover this case, because the types of the two variable $\mathrm{S}_1$ and $\mathrm{S}_2$ in this clause are the same, namely $\mathsf{plus}\ X_1\ X_3\ Y$. This is because the use of reflexivity for inequality in the third and fourth arguments of the clause requires $X_1 = X_2$ and $Y = Y'$. Our extended coverage checker will allow us to show automatically that this case is covered by exploiting the uniqueness information for plus.

We first define the situations in which uniqueness information may potentially be helpful, depending on the outcome of a unification problem. We then show how to exploit the result of unification to specialize a coverage goal.

**Definition 5 (Specializing a coverage goal).** *Given a mode-correct program $\mathcal{P}$ containing a type family $a$ with unique outputs, and a coverage goal $\Delta \vdash A$ :* type*, uniqueness specialization for $a$ may be applicable if*

1. *$\Delta$ contains distinct assumptions $x_1 : a\ M_1 \ldots M_n$ and $x_2 : a\ N_1 \ldots N_n$, and*
2. *for all $i \in ins(a)$, $M_i = N_i$, and*
3. *for some $k \in uouts(a)$, $M_k \neq N_k$.*

*To specialize the goal, attempt simultaneous higher-order unification of $M_k$ with $N_k$ for all $k \in uouts(a)$. If a most general pattern unifier (mgu) for this problem exists, write it as $\Delta' \vdash \sigma : \Delta$, and generate a new specialized goal $\Delta' \vdash A[\sigma]$ :* type*.*

There are three possible outcomes of the given higher-order unification problem, with the algorithm in [6]: (1) it may yield an mgu, in which case the specialized coverage goal is equivalent to the original one but has fewer variables, (2) it may fail, in which case the original goal is vacuously covered (that is, it has no ground instances), or (3) the algorithm may report remaining constraints, in which case this specialization is not applicable. Assertions (1) and (2) are corollaries of the next two lemmas.

**Lemma 3.** *If uniqueness information for a type family $a$ is potentially applicable to a coverage goal $g = \Delta \vdash A$ :* type*, but no unifier exists, then there are no ground instances of $g$ (and thus $g$ is vacuously covered by any set of patterns).*

*Proof.* Assume we had a substitution $\cdot \vdash \theta : \Delta$ (so that $A[\theta]$ is ground). Using the notation from Definition 5, we have $M_i = N_i$ for all $i \in \text{ins}(a)$ and therefore $M_i[\theta] = N_i[\theta]$. By Lemma 2, we have $\cdot \models (a\ M_1 \ldots M_n)[\theta] > \theta_1$ and $\cdot \models (a\ N_1 \ldots N_n)[\theta] > \theta_2$. Since the empty abstract substitution approximates the empty substitution, we know by Lemma 1 that for all $k \in \text{uouts}(a)$, $M_k[\theta] = N_k[\theta]$. But this is impossible since for at least one $k \in \text{uouts}(a)$, $M_k$ and $N_k$ were non-unifiable. □

**Lemma 4.** *Let $g = \Delta \vdash A$ :* type *be a coverage goal, and $\mathcal{P}$ a mode-correct program with uniqueness information for a potentially applicable to $g$. If an mgu $\Delta' \vdash \sigma : \Delta$ exists and leads to coverage goal $\Delta' \vdash A[\sigma]$ :* type*, then every ground instance $A[\theta]$ of $A$ is equal to a ground instance of $A[\sigma]$.*

*Proof.* As in the proof of the preceding lemma, assume $\cdot \vdash \theta : \Delta$ (so that $A[\theta]$ is ground). Again we have $M_i = N_i$ for all $i \in \text{ins}(a)$ and therefore $M_i[\theta] = N_i[\theta]$. By Lemma 2, we have $\cdot \models (a\ M_1 \ldots M_n)[\theta] > \theta_1$ and $\cdot \models (a\ N_1 \ldots N_n)[\theta] > \theta_2$ for some $\theta_1$ and $\theta_2$. From Lemma 1 we now know that for all $k \in \text{uouts}(a)$, $M_k[\theta] = N_k[\theta]$. But, by assumption, $\sigma$ is a most general simultaneous unifier of $M_k \doteq N_k$ for all $k \in \text{uouts}(a)$. Hence $\theta = \sigma \circ \theta'$ for some $\theta'$ and $A[\theta] = A[\sigma \circ \theta'] = (A[\sigma])[\theta']$. □

We return to the coverage checking problem for the type family of Figure 5. As observed above, without uniqueness information for plus it cannot be seen that all cases are covered. The failed coverage goal is

$\mathtt{X_1{:}nat,\ X_3{:}nat,\ Y{:}nat,\ Y'{:}nat,\ S_1{:}plus\ X_1\ X_3\ Y,\ S_2{:}plus\ X_1\ X_3\ Y'.}$
$\quad\vdash \mathsf{preserv}\ \mathtt{S_1}\ \mathtt{S_2}\ \mathsf{le\_refl}\ \_.$

Exploiting uniqueness information for plus, we have the unification problem $Y \doteq Y'$, with mgu $Y/Y'$, yielding the new goal

$\mathtt{X_1{:}nat,\ X_3{:}nat,\ Y{:}nat,\ S_1{:}plus\ X_1\ X_3\ Y,\ S_2{:}plus\ X_1\ X_3\ Y.}$
$\quad\vdash \mathsf{preserv}\ \mathtt{S_1}\ \mathtt{S_2}\ \mathsf{le\_refl}\ \_.$

Since $S_1$ and $S_2$ have the same type, the new goal is immediately covered by the clause preserv_refl, completing the check of the original coverage goal.

## 7 Conclusion

We have described an algorithm for verifying uniqueness of specified output arguments of a relation, given specified input arguments. We have also shown how to exploit this information in coverage checking, which, together with termination checking, can guarantee the existence of output arguments when given some inputs. We can therefore also verify unique existence, by separately verifying existence and uniqueness. While our algorithms can easily be seen to terminate, they are by necessity incomplete, since both uniqueness and coverage with respect to ground terms are undecidable in our setting of LF.

The uniqueness mode checker of Section 3 has been fully implemented as described. In fact, it allows arbitrary signatures, rather than just Horn clauses at the top level, although our critical correctness proof for Lemma 1 has not yet been extended to the more general case. We expect to employ a combination of the ideas from [18] and [19] to extend the current proof. In practice, we have found the behavior of the uniqueness checker to be predictable and the error messages upon failure to be generally helpful.

We are considering three further extensions to the uniqueness mode checker, each of which is relatively straightforward from the theoretical side. The first is to generalize left-to-right subgoal selection to be instead non-deterministic. This would allow verification of uniqueness for more signatures that were not intended to be executed with Twelf's operational semantics. The second would be to check that proof terms (and not just output arguments) will be ground or ground and unique. That would enable additional goal specialization in coverage checking. The third is to integrate the idea of *factoring* [17] in which overlapping clauses are permitted as long as they can be seen to be (always!) disjoint on the result of some subgoal.

In terms of implementation, we have not yet extended the coverage checker implementation in Twelf to take advantage of uniqueness information. Since specialization always reduces the complexity of the coverage goal when applicable, we propose an eager strategy, comparing inputs of type families having some unique outputs whenever possible. Since terms in the context tend to be rather small, we do not expect this to have any significant impact on overall performance.

Finally, we would like to redo the theory of foundational proof-carrying code [3, 4] taking advantage of uniqueness modes to obtain a concrete measure of the improvements in proof size in a large-scale example. We expect that most uses of explicit equality predicates and the associated proofs of functionality can be eliminated in favor of uniqueness mode checking and extended coverage checking. As a small proof of concept, we have successfully uniqueness-checked four type families in the theory, amounting to about 150 lines of Twelf code in which the use of functional arguments is pervasive. Combined with coverage checking, these checks might eliminate perhaps 250 lines of proof.

## 8 Acknowledgements

We thank the anonymous referees for several helpful suggestions.

## References

1. T. Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.
2. T. Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, Båstad, Sweden, 1992.
3. K. Crary. Toward a foundational typed assembly language. In G. Morrisett, editor, *Proceedings of the 30th Annual Symposium on Principles of Programming Languages*, pages 198–212, New Orleans, Louisiana, Jan. 2003. ACM Press.
4. K. Crary and S. Sarkar. A metalogical approach to foundational certified code. Technical Report CMU-CS-03-108, Carnegie Mellon University, Jan. 2003.
5. J. Despeyroux and P. Leleu. A modal lambda calculus with iteration and case constructs. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, pages 47–61, Kloster Irsee, Germany, Mar. 1998. Springer-Verlag LNCS 1657.
6. 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.
7. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.
8. R. Harper and F. Pfenning. On equivalence and canonical forms in the LF type theory. *Transactions on Computational Logic*, 2003. To appear. Preliminary version available as Technical Report CMU-CS-00-148.
9. M. Hofmann. *Extensional Concepts in Intensional Type Theory*. PhD thesis, Department of Computer Science, University of Edinburgh, July 1995. Available as Technical Report CST-117-95.
10. M. Hofmann and T. Streicher. The groupoid model refutes uniqueness of identity proofs. In *Proceedings of the 9th Annual Symposium on Logic in Computer Science (LICS'94)*, pages 208–212, Paris, France, 1994. IEEE Computer Society Press.
11. C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. Available as Technical Report ECS-LFCS-00-419.

12. F. Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, Mar. 2000.
13. F. Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001.
14. F. Pfenning and C. Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, pages 179–193, Kloster Irsee, Germany, Mar. 1998. Springer-Verlag LNCS 1657.
15. F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
16. B. Pientka. Termination and reduction checking for higher-order logic programs. In *First International Joint Conference on Automated Reasoning (IJCAR)*, pages 401–415, Siena, Italy, 2001. Springer Verlag, LNCS 2083.
17. A. Poswolsky and C. Schürmann. Factoring pure logic programs. Draft manuscript, Nov. 2003.
18. E. Rohwedder and F. Pfenning. Mode and termination checking for higher-order logic programs. In H. R. Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, Apr. 1996. Springer-Verlag LNCS 1058.
19. C. Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Aug. 2000. Available as Technical Report CMU-CS-00-146.
20. C. Schürmann. Recursion for higher-order encodings. In L. Fribourg, editor, *Proceedings of the Conference on Computer Science Logic (CSL 2001)*, pages 585–599, Paris, France, August 2001. Springer Verlag LNCS 2142.
21. C. Schürmann. A type-theoretic approach to induction with higher-order encodings. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence and Reasoning(LPAR 2001)*, pages 266–281, Havana, Cuba, 2001. Springer Verlag LNAI 2250.
22. C. Schürmann. Delphin – toward functional programming with logical frameworks. Technical Report TR #1272, Yale University, Department of Computer Science, 2004.
23. C. Schürmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266:1–57, 2001.
24. C. Schürmann and F. Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, Sept. 2003. Springer-Verlag LNCS 2758.
25. G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the Eighth International Conference on Functional Programming*, pages 249–262, Uppsala, Sweden, Aug. 2003. ACM Press.