A Coverage Checking Algorithm for LF

Carsten Schürmann[1] and Frank Pfenning[2] *

[1] Yale University
New Haven, CT, USA, email: carsten@cs.yale.edu
[2] Carnegie Mellon University
Pittsburgh, PA, USA, email: fp@cs.cmu.edu

Abstract. 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. It can be used to verify if a function defined by pattern matching covers all possible cases. This problem has a straightforward solution for the first-order, simply-typed case, but is in general undecidable in the presence of dependent types. In this paper we present a terminating algorithm for verifying coverage of higher-order, dependently typed patterns. It either succeeds or presents a set of counterexamples with free variables, some of which may not have closed instances (a question which is undecidable). Our algorithm, together with strictness and termination checking, can be used to certify the correctness of numerous proofs of properties of deductive systems encoded in a system for reasoning about LF signatures.

1 Introduction

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. This has a number of applications: in functional programming, it is used to decide if a given set of cases defining a function is exhaustive or not. In proof assistants it is used to verify if a purported proof covers all possible cases. Depending on the application, the underlying term algebra, meta-theoretic requirements, and efficiency considerations, a variety of algorithms that implement or decide properties about pattern matching emerge. In this paper we discuss one algorithm for coverage checking in the logical framework LF [9].

The choice of the underlying term algebra is essential. In traditional functional programming languages, for example, we have only simple types and possibly prefix polymorphism, and the structure of functions is not observable by pattern matching. This makes coverage checking straightforward, both in theory practice. In LF, on the other hand, we have dependent types and functions are intensional: their structure can be observed by pattern matching. This makes coverage checking undecidable since, for example, *any* set of patterns will cover all terms of an empty type and emptiness is undecidable.

* This work was supported in part by the National Science Foundation under grants CCR-0133502 and CCR-9988281.
Submitted, February 28, 2003.

Nonetheless, coverage checking has many applications in logical frameworks. Coquand's seminal paper [3] is concerned with the natural expression of proofs in a proof checker. Moreover, in certain dependent type theories, functions defined by pattern matching are strictly more expressive than schemata of primitive recursion [10]. In addition to coverage checking proof verification requires termination analysis which we do not consider here, but has previously been treated [22, 20]. Related is the work on Delphin [26] where functions may be defined by pattern matching over LF terms in their full generality. Verifying that such functions represent correct proofs requires coverage checking. In the context of the Twelf system [19], coverage checking is used to verify that a given *logic program* covers all cases for its input arguments. Besides checking the exhaustiveness of definitions, coverage checking is used to verify the correctness of meta-theoretic proofs represented as relations. Numerous case studies of this style of verification have been carried out, including cut-elimination for classical and intuitionistic logics [17], the Church-Rosser theorem for the untyped λ-calculus [15], and various translations between logical systems (see [16] for an introduction and survey). The running example in this paper is the correctness of bracket abstraction, which is a critical step in the translation from natural deduction to derivations from Hilbert's axioms.

As mentioned above, the coverage checking problem is in general undecidable. Our approach has been to design a sound approximation that always terminates and either certifies coverage or produces a set of potential counterexamples. Sometimes, these counterexamples are in effect impossible (due to dependent types). This algorithm has been fully implemented in the Twelf system and has proved enormously useful in practice to verify the correctness of meta-theoretic proofs expressed as relations. The largest project undertaken so far is Crary's implementation of foundational typed assembly language with about 30,000 lines of Twelf code [4] with more than 1000 theorems; other examples have been mentioned above. One valuable experience gained trough these experiments is that in the case of failure with spurious counterexamples it is in general possible to make a proof more explicit in such a way that it then passes the coverage checker.

The remainder of the paper is organized as follows: In Section 2 we briefly introduce the logical framework LF, and sketch the most important concepts necessary for this paper. In Section 3, we describe the coverage problem in detail present our coverage algorithm. In Section 4 we then describe its implementation in Twelf before mentioning related work in Section 5. Finally we assess results and conclude in Section 6. Because of space limitations, we were unable to integrate the illustrative example into this paper that can be found at `http://www.cs.cmu.edu/~twelf/notes/coverage.html`.

2 LF

The type-theoretic foundation for this paper is the logical framework LF [9]. In addition to the standard syntactic categories of objects, types, and kinds, we

will also use substitutions in a critical way throughout this paper so we briefly introduce them here (see also, for example, [2]).

$$
\begin{array}{lll}
\textit{Kinds} & K & ::= \text{type} \mid \Pi x{:}A.\, K \\
\textit{Atomic Types} & B & ::= a \mid B\ M \\
\textit{Types} & A & ::= B \mid \Pi x{:}A_1.\, A_2 \\
\textit{Objects} & M & ::= x \mid c \mid M_1\ M_2 \mid \lambda x{:}A.\, M \\
\textit{Signatures} & \Sigma & ::= \cdot \mid \Sigma, a : K \mid \Sigma, c : A \\
\textit{Contexts} & \Gamma & ::= \cdot \mid \Gamma, x : A \\
\textit{Substitutions} & \sigma & ::= \cdot \mid \sigma, M/x
\end{array}
$$

We write a for constant type families, x or u for object-level variables, and c for constructors. A *term* may come form any of the syntactic levels. As usual, we identify α-equivalent terms. In order to state certain definitions and propositions more concisely, we write U to stand for either an object or a type and V for either a type or a kind and h for a family-level or object-level constant. We take $\beta\eta$-conversion as the notion of definitional equality [9, 2], for which we write $U \equiv U'$ and $V \equiv V'$. Substitutions are capture-avoiding and written as $U[\sigma]$ or $V[\sigma]$ with the special form $U[M/x]$ and $V[M/x]$. Often, we write Δ for contexts that are interpreted existentially, and Γ for universal ones. When we write $\Gamma[\sigma]$, it is a shorthand for applying σ in left to right order to each variable type in Γ. Signatures, contexts, and substitutions may not declare a variable or constant more than once, and renaming of bound variables may be applied tacitly to ensure that condition. Besides equality, the main judgment is typing $\Gamma \vdash U : V$, suppressing the fixed signature Σ. We always assume our signatures, contexts and types to be valid.

Type-checking and definitional equality on well-typed terms for LF are decidable. Every term is equal to a unique β-normal η-long form which we call *canonical form*. In the remainder of the paper we assume that all terms are in canonical form, because it simplifies the presentation significantly. In the implementation this is achieved incrementally, first by an initial conversion of input terms to η-long form and later by successive weak-head normalization as terms are traversed.

Since it is perhaps not so well-known, we will give only the typing rules for substitutions, which are used pervasively in this paper.

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

For a context $\Gamma = (x_1{:}A_1, \ldots, x_n{:}A_n)$, we define $\mathsf{id}_\Gamma = (x_1/x_1, \ldots, x_n/x_n)$ so that $\Gamma \vdash \mathsf{id}_\Gamma : \Gamma$.

Composition of substitutions is defined by $(\cdot) \circ \sigma = (\cdot)$ and $(M/x, \theta) \circ \sigma = (M[\sigma]/x, \theta \circ \sigma)$. We will only apply a substitution $\Gamma' \vdash \sigma : \Gamma$ to a term $\Gamma \vdash U : V$ or a substitution $\Gamma \vdash \theta : \Gamma''$ resulting in $\Gamma' \vdash U[\sigma] : V[\sigma]$ and $\Gamma' \vdash \theta \circ \sigma : \Gamma''$, respectively.

3 Coverage

In this section we first formally define the problem of coverage in the LF type theory in Section 3.1. This relies on *higher-order matching*, a problem whose decidability is an open question. We therefore identify an important subclass, the *strict coverage problems* (Section 3.2) which guarantee not only decidability but also uniqueness of matching substitutions. All examples we have ever encountered in practice belong to this class and we explain the reasons for this after the necessary definitions. Then we define *splitting* in Section 3.3, which is the second critical operation to be performed during coverage checking. Next we describe our basic coverage algorithm and prove it sound and terminating in Sections 3.4 and 3.5. The last component of our coverage checker is *finitary splitting*, discussed and proved correct in Section 3.6.

3.1 Definition of Coverage

A coverage goal is simply a term (object or type) with some free variables. Intuitively, a coverage goal stands for all of its closed instances. In order to emphasize the interpretation of the variables as standing for closed terms, we write Δ for such contexts and denote variables in Δ by u and v rather than x and y. The distinction between Δ and Γ can be formalized (see [21]), but this is not necessary for the present purposes.

A coverage problem is given by a goal and a set of patterns. One can think of these as the patterns of a case expression in a functional program, or the input terms in the clause heads of a logic program. In the general case, a set of patterns is just a set of terms with free variables.

Definition 1 (Immediate Coverage). *We say a coverage goal $\Delta \vdash U : V$ is* immediately covered *by a collection of patterns $\Delta_i \vdash U_i : V_i$ if there is an i and a substitution $\Delta \vdash \sigma_i : \Delta_i$ such that $\Delta \vdash U \equiv U_i[\sigma_i] : V$.*

Coverage has an infinitary definition, requiring immediate coverage of every ground instance of a goal.

Definition 2 (Coverage). *We say $\Delta \vdash U : V$ is* covered *by a collection of patterns $\Delta_i \vdash U_i : V_i$ if every ground instance $\cdot \vdash U[\tau] : V[\tau]$ for $\cdot \vdash \tau : \Delta$ is immediately covered by the collection of terms $\Delta_i \vdash U_i : V_i$.*

In this formulation the problem of coverage is very general, because the type of U and the type of the U_i's are not the same. It turns out that the algorithm is significantly easier to describe and prove correct if we restrict U and U_i to be types, and $V = V_i = \text{type}$.

Definition 3 (Type-Level Coverage). *We say a goal $\Delta \vdash A : \text{type}$ is* covered *by a collection of patterns $\Delta_i \vdash A_i : \text{type}$ if every ground instance $\cdot \vdash A[\tau] : \text{type}$ for $\cdot \vdash \tau : \Delta$ is immediately covered by $\Delta_i \vdash A_i : \text{type}$.*

The implementation in Twelf transforms any coverage problems that arise into this type-level form. This translation is straightforward and only sketched here. Given a coverage goal $\Delta \vdash M : A'$. Assume first that $A' = a'\, N_1 \ldots N_k$ for $a' : \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.\mathrm{type}$. In this case we declare a new type family $a : \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.a'\, x_1 \ldots x_n \rightarrow \mathrm{type}$. The new coverage goal is now simply $\Delta \vdash a\, N_1 \ldots N_k\, M : \mathrm{type}$. All patterns are transformed in the analogous way, using the same a to replace a'. If A' starts with some leading Π-quantifiers we carry them over from the general to the restricted form.

To summarize, without loss of generality, in the remainder of this paper we consider only coverage goals of the form $\Delta \vdash A : \mathrm{type}$ and patterns of the form $\Delta_i \vdash A_i : \mathrm{type}$.

3.2 Strict Patterns

To determine if a goal is immediately covered we have to solve a higher-order matching problem, instantiating the patterns A_i to match the goal A. Not incidentally, this is also the operation that is performed when matching a case subject against the patterns in each arm of a case branch, or when unifying the input arguments to a predicate with the clause head.[1]

In order for this pattern matching to be decidable (for the coverage algorithm) and also so that the operational semantics is well-defined (for the execution of a functional or logic program), we require the patterns to be *strict*. Strictness for a pattern $\Delta_i \vdash A_i : \mathrm{type}$ requires that each variable in Δ_i must occur in A at least once in a rigid position [11, 18].

Definition 4 (Strictness). *We say that u* has a strict occurrence in *U if $\Delta; \Gamma \vdash_u U$ as defined by the rules depicted in Figure 1. A pattern $\Delta_i \vdash A_i : \mathrm{type}$ is strict if $\Delta_i; \cdot \vdash_u A_i$ for each variable u in Δ_i.*

Informally, an occurrence of u is strict if it is not below another variable in Δ and if that occurrence forms a higher-order pattern in the sense of Miller [14], that is, u is applied to distinct parameters as expressed by the judgment $\Gamma \vdash u\, x_1 \ldots x_n$ pat. Unlike higher-order patterns in the sense of Miller, however, other forms of occurrences of u are allowed, which is a practically highly significant generalization. All of the examples in Twelf are strict, but many higher-order examples are not patterns in the sense of Miller. Strictness is sufficient here because we are only interested in matching and not full unification.

Theorem 1. *Given a coverage goal $\Delta \vdash A : \mathrm{type}$ and a strict pattern $\Delta_i \vdash A_i : \mathrm{type}$. Then it is decidable if there is a substitution $\Delta \vdash \sigma : \Delta_i$ such that $A \equiv A_i[\sigma]$. Moreover, if such a substitution exists it is uniquely determined.*

Proof. See [24].

[1] This unification becomes matching because the input arguments of the goal are ground at run-time.

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

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

$$\frac{\Delta; \Gamma \vdash_u M_i}{\Delta; \Gamma \vdash_u c\ M_1 \ldots M_n}\ \mathsf{ls\_c} \quad (1 \le i \le n) \qquad \frac{\Delta; \Gamma \vdash_u M_i}{\Delta; \Gamma \vdash_u a\ M_1 \ldots M_n}\ \mathsf{ls\_a} \quad (1 \le i \le n)$$

$$\frac{y : A \in \Gamma \qquad \Delta; \Gamma \vdash_u M_i}{\Delta; \Gamma \vdash_u y\ M_1 \ldots M_n}\ \mathsf{ls\_var} \quad (1 \le i \le n)$$

$$\frac{\Gamma \vdash u\ x_1 \ldots x_n\ \mathrm{pat}}{\Delta; \Gamma \vdash_u u\ x_1 \ldots x_n}\ \mathsf{ls\_pat} \qquad \begin{array}{l}\text{no rule for } \Delta; \Gamma \vdash_u v\ M_1 \ldots M_n \\ \text{for } u \neq v,\ v : A \in \Delta\end{array}$$

Fig. 1. A formal system for strictness

Note that in the above theorem there is no requirement on the coverage goal A except that it be well-typed. Indeed, in practice, it will often not be a higher-order pattern, nor will it be strict. This failure of strictness is the result of the splitting operation described in the next section.

3.3 Splitting

In this section we present the second cornerstone of our coverage checking algorithm, namely *splitting*. This is a generalization of a similar operation proposed by Coquand [3]. Splitting is the answer to the question on how to proceed if the current coverage goal is *not* immediately covered by any of the patterns. In this case coverage might still hold, since we require only that all ground instances of the goal be immediately covered. Since there may be infinitely many ground instances, we instantiate the coverage goal only partially, one layer at a time.

In this situation the coverage goal may be refined into a new set of coverage goals, each of which must be covered in order for the initial coverage goal to succeed. This refinement of a coverage goal is determined by a finite complete collection of non-redundant substitutions for its free variables. Applied to the current coverage goal, each substitution generates a new coverage goal that can be checked for coverage recursively.

Implementation of refinement will be via the splitting operation on a coverage goal, which requires higher-order unification rather than just matching. It is discussed in the remainder of this section. The strategy for how to invoke this operation is the subject of the next section (3.4).

Definition 5 (Non-redundant complete set of substitutions).
Let $\Delta \vdash A : \mathrm{type}$ *be a coverage goal. We say a finite collection* $\Delta_i \vdash \tau_i : \Delta$ *is a*

non-redundant complete set of substitutions *if for every* $\cdot \vdash \tau : \Delta$ *there exists a unique* i *and a unique* $\cdot \vdash \sigma_i : \Delta_i$ *such that* $\cdot \vdash \tau = \tau_i \circ \sigma_i$.

Refining coverage goals through non-redundant complete set of substitutions is a conservative operation. Coverage of the refined set of coverage goals implies coverage of the original goal.

Theorem 2 (Conservativity of refinement). *Let* $\Delta \vdash A$: type *be a coverage goal and* $\Delta_i \vdash \tau_i : \Delta$ *a non-redundant complete collection of substitutions. All* $\Delta_i \vdash A[\tau_i]$: type *are covered by a given set of patterns if and only if* $\Delta \vdash A$: type *is covered.*

Proof. Coverage depends only on the set of ground instances of a coverage goal. But the collection of all ground instances of $\Delta_i \vdash A[\tau_i]$ is exactly the same as the set of ground instanced of $\Delta \vdash A$: type since the τ_i form a complete set. Hence coverage is preserved by refinement. □

Next we address the question of how to construct such a refinement. The method we are using is called *splitting*, and is inspired by a similar operation present in ALF [3, 1] which in turn goes back to the basic steps in Huet's algorithm for higher-order unification [11].

Among all the goals that are not immediately covered we select one goal $\Delta \vdash A$: type, and from its context Δ one declaration $u{:}A$. We refer to u as the *splitting variable*. A may be a function type, therefore, without loss of generality, it is of the following form

$$\Delta \vdash u : \Pi\Gamma.\, a\ M_1 \dots M_m.$$

For the sake of conciseness, we consolidate all successive Π-abstraction into one context Γ. This is only an abbreviation and does not properly extend LF. We also use the following abbreviations $p\ \Gamma$ which stands for $p\ x_1 \dots x_m$ if $\Gamma = x_1{:}A_1, \dots, x_m{:}A_m$ where p is a constant or a parameter. Furthermore $p\ (\Delta\ \Gamma)$ is a shorthand for $p\ (u_1\ \Gamma) \dots (u_n\ \Gamma)$ for $\Delta = u_1{:}A'_1, \dots, u_n{:}A'_n$.

We now want to determine the possible top-level structures of a term $M : \Pi\Gamma.\, a\ M_1 \dots M_m$. Because of the existence of canonical forms, it is enough to search the signature and the local context for constants that may occur in a head position in M. All we have to do is to verify that types unify, but this is far from trivial, since we are in the higher-order setting and have dependent types. We will discuss our choice of unification algorithm in more detail later; here we simply describe how to invoke it to obtain a complete and non-redundant set of substitutions.

Let Γ be the local context of variables under which we have to consider a constant application. In general, the type of a constant is $\Pi u_1{:}A_1. \dots \Pi u_n{:}A_n.\, B$ with an atomic type B. For the purpose of splitting, each u_i is intuitively interpreted as an existential variable that can be instantiated to terms valid in Γ. To account for those local dependencies, we raise those variables by Γ and turn all u_i into variables of functional type abstracting over Γ.

Definition 6 (Raising). *Let Γ be a context of local parameters, A the type of a constant c. Raising A by Γ yields a $\langle \Delta \vdash A' \rangle$, a context Δ of raised existential variables and a raised type A' (that always has the form $\Pi\Gamma.\, B$).*

$$raise\langle \Gamma \vdash A \rangle = \begin{cases} \langle \cdot \vdash \Pi\Gamma.\, A \rangle & \text{if } A \text{ is atomic} \\ \langle u : \Pi\Gamma.\, A_1, \Delta \vdash A' \rangle & \text{if } A = \Pi u{:}A_1.\, A_2 \\ & \text{and } \langle \Delta \vdash A' \rangle = raise\langle \Gamma \vdash A_2[u\ \Gamma/u] \rangle \end{cases}$$

What makes raising such a tricky operation is that the u_i may occur elsewhere in the the type, and need to be replaced by their raised versions u_i applied to Γ. The Δ that is computed during raising contains all u_i's in raised form.

Next, we describe the central definition of this section: splitting. We follow standard practice and describe unification as a first-order formula over equations $U \approx U'$. The particular unification algorithm that we use is higher-order pattern unification that postpones unresolved unification equation as constraints. The algorithm is described in detail in [5]. For our coverage algorithm, however we restrict its generality a bit: Although we allow constraints to arise during the process of unification, we require that after completion all constraints have been resolved. Otherwise we do not allow splitting over the specified variable. This is handled in our algorithm for selecting variables to split by trying another variable instead. Unfortunately, successive selections of splitting variables are not independent and it is possible that some sequences of splitting operations fail (with spurious counterexamples) while other sequences could succeed. In principle we could backtrack here, but this is currently not implemented.

Definition 7 (Splitting). *Let $\Delta \vdash A : \text{type}$ a coverage goal, and u in $\Delta = \Delta_1, u : \Pi\Gamma.\, B_u, \Delta_2$ a splitting variable. The splitting operation considers each constant c declared in the signature Σ and each local parameter y declared in Γ in turn, and determines a set of substitutions σ_c, σ_y as follows.*

1. Constants: *Let $c : \Pi\Delta_c.\, B_c \in \Sigma$, and $\langle \Delta'_c \vdash \Pi\Gamma.\, B'_c \rangle = raise\langle \Gamma \vdash \Pi\Delta_c.\, B_c \rangle$. Let $\Delta' \vdash \sigma_c : \Delta, \Delta'_c$ be the most general unifier of the higher-order unification problem*
$$\exists\Delta.\, \exists\Delta'_c.\, (\Pi\Gamma.\, B_u \approx \Pi\Gamma.\, B'_c) \wedge (u \approx \lambda\Gamma.\, c\ (\Delta'_c\ \Gamma)) \tag{1}$$
if it exists.
2. Bound Variables: *Let $y : \Pi\Delta_y.\, B_y \in \Gamma$, and $\langle \Delta'_y \vdash \Pi\Gamma.\, B'_y \rangle = raise\langle \Gamma \vdash \Pi\Delta_y.\, B_y \rangle$. Let $\Delta' \vdash \sigma_y : \Delta, \Delta'_y$ be the most general unifier of the higher-order unification problem*
$$\exists\Delta.\, \exists\Delta'_y.\, (\Pi\Gamma.\, B_u \approx \Pi\Gamma.\, B'_y) \wedge (u \approx \lambda\Gamma.\, y\ (\Delta'_y\ \Gamma)) \tag{2}$$
if it exists.

Since we collect all such most general unifiers, cases for which the unification problem fails[2] *simply do not contribute a substitution to the result of the splitting operation.*

[2] but not those whose results are indeterminate because of residual equations, which are are not permitted

The main result of this section is that splitting generates always a set of substitutions that is non-redundant and complete. Obviously, raising will play a major role in this algorithm, prompting us to prove an auxiliary lemma about raising that guarantees that any instantiation $\sigma = M_1/u_1, \ldots, M_n/u_n$ of variables in Δ with respect to Γ can be raised to the empty context as $\sigma' = (\lambda\Gamma.\, M_1)/u_1 \ldots (\lambda\Gamma.\, M_n)/u_n$. Because of space considerations, we have omitted a generalized formulation of this lemma that one would prove by induction over the structure of the context Δ.

Lemma 1 (Raising). *Let Δ be context and B an atomic type. If*

$$\mathit{raise}\langle \Gamma \vdash \Pi\Delta.\, B\rangle = \langle \Delta' \vdash \Pi\Gamma.\, B'\rangle$$

and $\Gamma \vdash \mathsf{id}_\Gamma, \sigma : \Gamma, \Delta$ then there exists a substitution σ', s.t. $\cdot \vdash \sigma' : \Delta'$ and

$$\cdot \vdash \Pi\Gamma.\, B[\sigma] \equiv (\Pi\Gamma.\, B')[\sigma'].$$

and for all corresponding u from Δ' and Δ, respectively, the following equation holds: $(u[\sigma'])\ \Gamma \equiv u[\sigma]$.

Finally, we state and prove the main theorem of this section that informally states that no cases are lost due to splitting.

Theorem 3 (Splitting is non-redundant and complete). *The set of substitutions generated by splitting is non-redundant and complete.*

Proof. Non-redundancy: Trivial since for each $c \in \Sigma$ and $y \in \Gamma$, $\sigma_c(u)$ as well as $\sigma_y(u)$ have distinct head.
Completeness: Let $\Delta \vdash A : \text{type}$ and $u : \Pi\Gamma.\, B_u \in \Delta$ a splitting variable and $\cdot \vdash \sigma : \Delta$. Let $\cdot \vdash \sigma(u) = M : (\Pi\Gamma.\, B_u)[\sigma]$ where M has canonical form $\cdot \vdash \lambda\Gamma[\sigma].\, p\ M_1 \ldots M_n$. There are a constant and parameter case to consider for p.

Case: $p = c : \Pi\Delta_c.\, B_c \in \Sigma$. We construct a substitution $\tau = \mathsf{id}_\Gamma, M_1/u_1, \ldots M_n/u_n$ with $\Gamma[\sigma] \vdash \tau : \Gamma[\sigma], \Delta_c$. Let $\langle \Delta'_c \vdash \Pi\Gamma[\sigma].\, B'_c\rangle = \mathrm{raise}\langle \Gamma[\sigma] \vdash \Pi\Delta_c.\, B_c\rangle$. By Corollary 1 there exists a substitution τ' such that $\cdot \vdash \tau' : \Gamma'_c$ and

$$\cdot \vdash \Pi\Gamma[\sigma].\, B_c[\tau] \equiv (\Pi\Gamma[\sigma].\, B'_c)[\tau'].$$

Therefore, by concatenating σ and τ' we obtain a new substitution η, that satisfies $\cdot \vdash \eta : \Delta, \Delta'_c$. By uniqueness of types for LF, the following types are equivalent:

$$\cdot \vdash (\Pi\Gamma.\, B_u)[\eta] \equiv (\Pi\Gamma.\, B'_c)[\eta] : \text{type}.$$

Furthermore, also from Corollary 1, we can infer that for all $u_i \in \Delta'_c$,

$$u_i[\tau']\ (\Gamma[\sigma]) \equiv u_i[\tau] \equiv M_i$$

and hence

$$(\lambda\Gamma.\, c\ (\Delta'_c\ \Gamma))[\eta] \equiv \lambda\Gamma[\sigma].\, (c\ (\Delta'_c[\tau']\ (\Gamma[\sigma]))) \equiv \lambda\Gamma[\sigma].\, (c\ M_1 \ldots M_n) \equiv u[\eta].$$

Consequently, η is a unifier for Equation (1). Recall, that by construction σ_c is most general. Therefore, there exists a $\cdot \vdash \sigma' : \Delta'$, such that $\eta = \sigma_c \circ \sigma'$. By restriction to Δ, we obtain that there exists an σ'' such that $\sigma = \sigma_c \circ \sigma''$.

Case: Almost identical to the one above, except that η will be a unifier for Equation (2). □

3.4 The Coverage Algorithm

Recall that a coverage goal $\Delta \vdash A : \text{type}$ is immediately covered by a collection of terms $\Delta_i \vdash A_i : \text{type}$ if there is an i and $\Delta \vdash \sigma_i : \Delta_i$ such that $\Delta \vdash A \equiv A_i[\sigma_i] : \text{type}$.

Immediate coverage is central to the naive, non-deterministic coverage algorithm which we discuss next. We assume we have a set of coverage goals, all of which must be covered for the algorithm to succeed. In the first step, this is initialized with the goal $\Delta \vdash A : \text{type}$. We pick one of the coverage goals and determine, via strict higher-order matching, if it is immediately covered by any covering type A_i. If so we remove it from the set and continue. If not, we non-deterministically select a variable in the coverage goal and split it into multiple goals, which replace it in the collection of coverage goals.

This coverage algorithm is naive because it may not terminate, even if the goal is covered. Even if types are non-empty and coverage holds, splitting the wrong variable can lead to non-termination.

The procedure we propose in this section always terminates and either indicates that coverage holds, or outputs a set of potential counterexamples. Some of these may fail to be actual counterexamples, because we me may not be able to instantiate the remaining variables to a ground term that is not covered. If the counterexample is ground, however, it is guaranteed to be an actual counterexample. We analyze the possible forms of counterexamples in more detail at the beginning of Section 3.6.

The basic idea is to record *why* immediate coverage fails and not just *if* it does. Assume we are given a coverage goal $\Delta \vdash A : \text{type}$ and a pattern $\Delta' \vdash A' : \text{type}'$. Instead of just applying our matching algorithm, we then construct a conjunction of equations E and the symbols $\top$ (success) or $\bot$ (failure) such that $\langle \Delta, \Delta'; \cdot \vdash A < A' \rangle \Longrightarrow E$. This is accomplished by using the rules for the judgment

$$\langle \Delta; \Gamma \vdash U < U' \rangle \Longrightarrow E$$

defined in Figure 2. We should read this judgment as: *Match U against pattern U' in the parameter context Γ to obtain the residual equations E.* Δ is the disjoint union of the (existential) variables in U and U', of which only those in U may be instantiated during matching. Initially, the context Γ is always empty, and both U and U' are types. However, internally we require the context Γ of shared local parameters.

We can think of the algorithm as a rigid decomposition, which corresponds to the `simplify` function in Huet's algorithm for higher-order unification. If all residual equations can be solved (and there is no $\bot$), then matching is successful. Otherwise, we have to interpret the equations to determine candidates for splitting that will make progress (as defined below).

$$\frac{\langle \Delta; \Gamma, x{:}A \vdash U < U' \rangle \Longrightarrow E}{\langle \Delta; \Gamma \vdash \lambda x{:}A.\, U < \lambda x{:}A.\, U' \rangle \Longrightarrow E}$$

$$\frac{\langle \Delta; \Gamma \vdash A < A' \rangle \Longrightarrow E_1 \qquad \langle \Delta; \Gamma, x{:}A' \vdash B < B' \rangle \Longrightarrow E_2}{\langle \Delta; \Gamma \vdash \Pi x{:}A.\, B < \Pi x{:}A'.\, B' \rangle \Longrightarrow E_1 \wedge E_2}$$

$$\frac{}{\langle \Delta; \Gamma \vdash \Pi x{:}A.\, B < a \ldots \rangle \Longrightarrow \bot} \qquad \frac{}{\langle \Delta; \Gamma \vdash a \ldots < \Pi x{:}A.\, B \rangle \Longrightarrow \bot}$$

$$\frac{\langle \Delta; \Gamma \vdash U_i < U_i' \rangle \Longrightarrow E_i \qquad \text{for } 1 \leq i \leq n}{\langle \Delta; \Gamma \vdash c\, U_1 \ldots U_n < c\, U_1' \ldots U_n' \rangle \Longrightarrow E_1 \wedge \ldots \wedge E_n}$$

$$\frac{c \neq c'}{\langle \Delta; \Gamma \vdash c \ldots < c' \ldots \rangle \Longrightarrow \bot} \qquad \frac{\Gamma(x) = A \quad \langle \Delta; \Gamma \vdash U_i < U_i' \rangle \Longrightarrow E_i \qquad \text{for } 1 \leq i \leq n}{\langle \Delta; \Gamma \vdash x\, U_1 \ldots U_n < x\, U_1' \ldots U_n' \rangle \Longrightarrow E_1 \wedge \ldots \wedge E_n}$$

$$\frac{\Gamma(x) = A, \Gamma(y) = A', x \neq y}{\langle \Delta; \Gamma \vdash x \ldots < y \ldots \rangle \Longrightarrow \bot} \qquad \frac{\Gamma(x) = A, \Sigma(c) = A'}{\langle \Delta; \Gamma \vdash x \ldots < c \ldots \rangle \Longrightarrow \bot}$$

$$\frac{\Delta(u) = A, \Gamma(x) = A'}{\langle \Delta; \Gamma \vdash u\, U_1 \ldots U_n < x\, U_1' \ldots U_m' \rangle \Longrightarrow \langle \Gamma \vdash u\, U_1 \ldots U_n \approx x\, U_1' \ldots U_m' \rangle}$$

$$\frac{\Delta(u) = A, \Sigma(c) = A'}{\langle \Delta; \Gamma \vdash u\, U_1 \ldots U_n < c\, U_1' \ldots U_m' \rangle \Longrightarrow \langle \Gamma \vdash u\, U_1 \ldots U_n \approx c\, U_1' \ldots U_m' \rangle}$$

$$\frac{}{\langle \Delta; \Gamma \vdash U < u'\, U_1' \ldots U_m' \rangle \Longrightarrow \langle \Gamma \vdash U \approx u'\, U_1' \ldots U_m' \rangle}$$

Fig. 2. Rigid Matching Algorithm

Note that during rigid matching, no variable assignment takes place: where the two terms disagree, we record an equation. But if matching is not possible, we might either record an equation or return $\bot$.

In order the state the lemmas in the generality required for an inductive proof, we say that for $\Delta; \Gamma \vdash U : V$ and $\Delta'; \Gamma \vdash U' : V'$ that U' *covers* U if there is a substitution $\Delta, \Gamma \vdash \sigma, \mathsf{id}_\Gamma : \Delta', \Gamma$ such that $\Delta, \Gamma \vdash U \equiv U'[\sigma, \mathsf{id}_\Gamma] : V$.

Lemma 2. *If $\Delta, \Delta'; \Gamma \vdash U < U' \Longrightarrow E$ where E contains $\bot$, then U' does not immediately cover U or any instance of U.*

Proof. By induction on the given derivation.

Because U' cannot immediately cover any instance of U, we do not generate any candidate variables for splitting in Δ' in this case.

Lemma 3. *If $\Delta, \Delta'; \Gamma \vdash U < U' \Longrightarrow E$ where E does not contain $\bot$, but contains equations of the form $u \ldots \approx c \ldots$ or $u \ldots \approx x \ldots$. Then U' does not immediately cover U (but U' could possibly cover some instance of U).*

Proof. By induction on the given derivation. In the base cases, x and c are rigid and therefore cannot be instantiated to u.

In this case, any variable u occurring in an equation of the given form is added to the set of candidate variables for splitting, since it is possible that splitting might make progress.

Lemma 4. *If $\Delta, \Delta'; \Gamma \vdash U < U' \Longrightarrow E$, where E does not contain $\bot$ or equations for the form $u \ldots \approx c \ldots$ or $u \ldots \approx x \ldots$. Then any substitution $\Delta \vdash \sigma : \Delta'$ such that for each residual equation $\langle \Gamma_i \vdash U_i \approx U_i' \rangle$ in E we have $\Gamma_i \vdash U_i \equiv U_i'[\sigma, \mathsf{id}_{\Gamma_i}]$ is a valid match and shows that U' covers U.*

Proof. Again, by induction on the given derivation. The base cases are evident. The tricky part in the inductive argument is that the two matched terms do not necessarily have the same type or kind (even though the do initially) because we postpone non-rigid equations. However, as in the case of higher-order dependently typed unification [6], it is enough to maintain well-typedness modulo postponed equations if we eventually solve them from left-to-right.

This means that if we have no candidates from the first two kinds of equations, we call a strict higher-order matching algorithm [24] on the residual equations. If this succeeds then A' covers A. Otherwise, A' does not cover A and we suggest no candidate variables for splitting because it would be difficult to guarantee termination.

When considering a particular coverage goal $\Delta \vdash A : \mathrm{type}$, we apply the above algorithm with each pattern. If one of them immediately covers, we are done. If not, we take the union of all the suggested candidates and pick one non-deterministically. The current implementation picks the rightmost candidate in Δ, because internal dependencies might further constrain variables to its left during the splitting step. If splitting fails because higher-order unification with the algorithm in [5] can not determine a complete and non-redundant set of substitutions, then we try another candidate, and so on. If there are no remaining splitting candidate, we add the coverage goal to the set of potential counterexamples and pick another goal.

3.5 Termination

The overall structure of the algorithm is such that the splitting step replaces a coverage goal by several others. In order to show termination with respect to a simple multi-set ordering, we must show that each of the subgoals that replace a given goal are smaller according some well-founded measure.

We calculate this measure as follows. Given a coverage goal $\Delta \vdash A : \mathrm{type}$ apply rigid matching against each pattern. Eliminate those equations that contain $\bot$. Among the remaining ones, consider only equations $u\, U_1 \ldots U_n \approx h'\, U_1' \ldots U_m'$ where $h = x$ or $h = c$. Note that all candidates for splitting appear on the left-hand side of such an equation. Take the sum of the sizes of the right-hand sides as measured by the number of bound variable and constant occurrences.

When we apply splitting to any candidate variables in Δ, that is, one of the variables u that appears on the left-hand side of an equation as given above, then this measure decreases.

Lemma 5. *Given a coverage goal $\Delta \vdash A$: type and a fixed set of patterns proposed to cover it. If we split the coverage goal along a variable u suggested by rigid matching, each of the resulting subgoals has a smaller measure than the original goal.*

Proof. u occurs on the left-hand side of at least one residual equation $\langle \Gamma \vdash u\, U_1 \ldots U_n \approx h'\, U'_1 \ldots U'_m \rangle$. After splitting, this residual equation may disappear altogether (say, because the case has become impossible). However, if rigid matching reaches again this subterm in the subgoal, it will now have the form $\Gamma \vdash h\, U^*_1 \ldots U^*_k \approx h'\, U'_1 \ldots U'_m$ for some $h = x$ (a local parameter in Γ) or $h = c$ (a constant). If $h \neq h'$ then the this equation drops out altogether, since it generates $\bot$ instead. If $h = h'$, then $k = m$ and the algorithm recurses by comparing each U^*_j with U'_j for $1 \leq j \leq m$. But this eliminates at least one constant or variable occurrence (namely h), thereby decreasing our measure. □

Theorem 4. *Coverage checking terminates after a finite number of steps, yielding either an indication of coverage or a finite set of potential counterexamples.*

Proof. Immediate by the previous lemma by a multi-set ordering on the set of coverage goals.

3.6 Finitary splitting

The failure-directed algorithm described above works well in most practical cases, within or outside the pattern fragment. There are two remaining difficulties: one are remaining constraints during splitting as discussed in Section 3.3, the other is that occasionally the generated counterexamples fail to be actual counterexamples. The latter is a common occurrence. In large part this is because meta-theoretic proofs represented as dependently typed functions or relations often have a number of cases that are impossible. Instead of explicitly proving that the cases are impossible, one usually just lists the cases that can arise if it is syntactically obvious that the others can not arise.

What are the types of spurious counterexamples that may be produced by the algorithm? The most obvious one is a coverage goal that is incompatible with all patterns, but has no ground instances. We explain below how to handle some of these case. A less obvious problem is that matching the residual equations fail because of a spurious dependency that cannot be an actual dependency because of subordination considerations. We treat this case by applying strengthening [24] to eliminate these spurious dependencies throughout the algorithm. Finally, it is possible that two distinct variables of the coverage goal fail to match, yet they must be identical because the type has only one element. Finitary splitting will often catch these cases and correctly report coverage.

In order to handle as many spurious counterexamples as possible, we extend the algorithm described above as follows. Once the algorithm terminates with a set of proposed counterexamples to coverage, we examine each such counterexample to see if we can determine if it is impossible, that is, if it quantifies over an empty type. More concretely, let $\Delta \vdash A : \text{type}$ be a counterexample, that is, coverage goal that is not covered and does not produce any splitting candidates. We now attempt to split each variable $u{:}A$ in Δ in turn, leading to a new set of coverage goals $\Delta_i \vdash U_i : V_i$ for $0 \leq i < n$. If $n = 0$ we know that the case is impossible.

If $n > 0$ we could, in principle, continue the algorithm recursively to see if each of the subgoals $\Delta_i \vdash U_i : V_i$ are impossible. However, in general this would not terminate (and cannot, because inhabitation is undecidable). Instead, we only continue to split further if all of the new variables $u_k : A_k$ in Δ_i have a type that is strictly subordinate to the type A [27, 24]. Otherwise, we fail and report the immediate supergoal as a potential counterexample.

Theorem 5. *Finitary splitting terminates, either with an indication that the given coverage goal has no ground instances, or failure.*

Proof. There are only a finite number of variables in a given coverage goals. During each step of splitting we either stop or obtain subgoals where a variable $u : A$ has been replaced by several variables $u_i : A_i$ each of which has a type strictly lower in the subordination hierarchy. Since this hierarchy is well-founded, finitary splitting will terminate.

This process can be very expensive. Fortunately, we have not found it to be a bottleneck in practice, because finitary splitting is applied only to remaining counterexamples. Usually, there are not many, and usually it is immediate to see that they are indeed possible because most types are actually inhabited. We do not presently try to verify if the types are actually inhabited (that is, start a theorem prover), although it may be useful for debugging purposes to distinguish between definite and potential counterexamples. However, in a future extension this could be done at the user's direction if he or she cannot easily detect the source of the failure of coverage.

4 Implementation

The coverage checking algorithm is implemented as part of the current Twelf [19] distribution, available from the Twelf webpage at `http://www.twelf.org/`. From the user's perspective, it can be employed in two different ways.

First, Twelf ascribes an operational meaning to LF signatures, that can be executed with a logic programming interpreter. Verifying coverage via the coverage checker means that execution will always be able to make progress and can not fail assuming the program is well-moded, that is, the role of arguments for input and output are properly respected. That it also terminates is an entirely different issue enforced by a termination and reduction checker [22, 20]. In

this relational form, the coverage algorithm distinguishes between input coverage (the argument position that will be matched when a logical program is called) and the output coverage (the argument position that will be matched after a subgoal has been successfully evaluated). Although the interaction between the two is well understood up to programs of order 2, output coverage is a difficult operation to implement for higher-order logic programs of order 3 and greater.

Second, the internal data structures of Twelf are taken advantage of by the functional programming language Delphin that supports function definition by cases over arbitrary LF terms. Its type theory is based on $\mathcal{T}_\omega^+$ [25]. Although a suitable Delphin parser is still under construction, a specialized converter allows Twelf logic programs to be translated and run natively in Delphin. The differentiating features of $\mathcal{T}_\omega^+$ to type theories used in other functional programming languages are dependently typed data, pattern matching against functions, a world system that controls the dynamic extension of a datatype by new constructors at run-time. Delphin programs do not distinguish between input and output coverage since they are functional programs, which renders it an attractive target platform for coverage checking of Twelf logic programs. And indeed, we have managed to overcome the limitations of the Twelf coverage checker due to order restriction by translating Twelf logic programs into Delphin functional programs, and subsequently applying the Delphin coverage checker.

5 Related Work

Coquand has considered the problem of coverage for a type theory in the style of Martin-Löf [3]. He defines coverage and splitting in much the same way we do here, except that no matching against the structure of λ-expressions is allowed. He also suggests a non-deterministic semi-decision procedure for coverage by guessing the correct sequence of variable splits. In an implementation this split can be achieved interactively.

Most closely related to ours is the work by McBride [13]. He refines Coquand's idea by suggesting an algorithm for successive splitting that is quite similar to ours in the first-order case. He also identifies the problem of empty types and suggest to recognize "obviously" empty types, which is a simpler variant of finitary splitting. Our main contribution with respect to McBride's work is that we allow matching against the structure of higher-order terms which poses significant additional challenges.

Another related development is the theory of partial inductive definitions [8], especially in its finitary form [7] and the related notion of definitional reflection [23]. This calculus contains a rule schema that, re-interpreted in our context, would allow any (finite) complete set of unifiers between a coverage goal $\Delta \vdash A : \text{type}$ and the heads of the clauses defining A. Because of the additional condition of so-called a-sufficiency for the substitutions this was never fully automated. Also, it appears that a simple, finite complete set of unifiers was computed as in the splitting step, but that the system could not check whether an arbitrary given set of premises could be obtained as a finite complete set of unifiers.

In the Coq system [12] functions defined by patterns can be compiled to functions defined by standard primitive recursive elimination forms. Because of the requirement to compile such functions back into pure Coq and the lack of matching against functional expressions, the algorithm is rather straightforward compared to our coverage checker and does not handle variable dependencies, non-linearity, or empty types. It does, however, treat polymorphism which we have not considered.

6 Conclusion

We have presented a solution to the coverage checking problem for LF, generalizing and extending previous approaches. The central technical developments are strict patterns (which significantly generalize higher-order patterns in the sense of Miller), strict higher-order matching, splitting in the presence of full higher-order unification, and a two-phase control structure to guarantee termination of the algorithm.

Our coverage algorithm is sound and terminating, but it is necessarily incomplete. Applied to a given set of patterns, it either reports "yes", or it generates a set of potential counterexamples, that often contain the vital information about why coverage has failed. Because coverage is undecidable in the case of LF, the algorithm sometimes generates spurious counterexamples, that can sometimes be removed with a highly specialized albeit incomplete algorithm called finitary splitting and has proven tremendously useful in practice.

All algorithms and techniques described in this paper are implemented in the Twelf system, Version 1.4 (December 2002). Many examples of coverage are available in the example directories of the Twelf distribution. The current implementation is somewhat more general than what we describe here since it also accounts for regular worlds [24]. We plan to extend the rigorous treatment given here to this larger class of coverage problems in a future paper.

References

1. C. Coquand. A proof of normalization for simply typed lambda calculus written in ALF. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 85–92, Båstad, Sweden, 1992.
2. 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.
3. T. Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, Båstad, Sweden, 1992.
4. K. Crary. Toward a foundational typed assembly language. In G. Morrisett, editor, *Proceedings of the 30th Annual Symposium on Principles of Programming Languages*, New Orleans, Louisiana, Jan. 2003. ACM Press. To appear.
5. 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.

6. C. M. Elliott. *Extensions and Applications of Higher-Order Unification*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.
7. L.-H. Eriksson. *Finitary Partial Inductive Definitions and General Logic*. PhD thesis, Department of Computer and System Sciences, Royal Institute of Technology, Stockholm, 1993.
8. L. Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 87(1):115–142, Sept. 1991.
9. 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.
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. G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
12. INRIA. *The Coq Proof Assistant*, version 7.4 edition, Feb. 2003. Reference Manual.
13. C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. Available as Technical Report ECS-LFCS-00-419.
14. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
15. F. Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *Journal of Automated Reasoning*, 1993. To appear. A preliminary version is available as Carnegie Mellon Technical Report CMU-CS-92-186, September 1992.
16. F. Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter XXI. Elsevier Science and MIT Press, 1999. In preparation.
17. F. Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, Mar. 2000.
18. 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.
19. 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.
20. 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.
21. B. Pientka and F. Pfenning. Optimizing higher-order pattern unification. Submitted, Jan. 2003.
22. 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.
23. P. Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 222–232, Montreal, Canada, June 1993.

24. 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.
25. 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.
26. C. Schürmann, R. Fontana, and Y. Liao. Delphin: Functional programming with deductive systems. Draft.
27. R. Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, Sept. 1999. Available as Technical Report CMU-CS-99-167.