Imperative LF Meta-Programming

Aaron Stump

Dept. of Computer Science and Engineering Washington University in St. Louis St. Louis, Missouri, USA Web: http://cl.cse.wustl.edu/

Abstract

Logical frameworks have enjoyed wide adoption as meta-languages for describing deductive systems. While the techniques for representing object languages in logical frameworks are relatively well understood, languages and techniques for meta-programming with them are much less so. This paper presents work in progress on a programming language called Rogue-Sigma-Pi (RSP), in which general programs can be written for soundly manipulating objects represented in the Edinburgh Logical Framework (LF). The manipulation is sound in the sense that, in the absence of runtime errors, any putative LF object produced by a well-typed RSP program is guaranteed to type check in LF. An important contribution is an approach for soundly combining imperative features with higher-order abstract syntax. The focus of the paper is on demonstrating RSP through representative LF meta-programs.

Key words: Meta-Programming, Logical Frameworks, Rewriting Calculus

1 Introduction

Applications using a logical framework such as the Edinburgh Logical Framework (LF) [7] very frequently need meta-programs which produce or manipulate LF encodings of derivations. For example, proof-producing decision procedures like the CVC ("Cooperating Validity Checker") system or the Touchstone theorem prover from Necula's PCC system emit proof objects for formulas they report valid [18,10]. The Princeton and Yale Foundational Proof-Carrying Code (FPCC) projects both rely on tools that automatically generate LF derivations [21,6].

The present work contributes to the study of sound meta-programming for LF. A meta-programming language for LF is described called Rogue-Sigma-Pi (RSP). RSP extends LF in a type-safe way with convenient meta-programming constructs: pattern-matching, general recursion, a limited form of dependent records, and expression attributes (for mutable state). The combination of mutable state with *higher-order abstract syntax* (HOAS) [12] is quite delicate.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

An important contribution of the present work in progress is a (currently just conjectured) sound approach supporting this combination.

The meta-theoretic properties of RSP necessary for its intended use are type safety and conservativity with respect to LF. Due to the inclusion of dependent records (as also happens with the inclusion of pairs: see, e.g., [16]), unicity of types fails in LF and hence in RSP. The approach adopted here is to rely on bottom-up type computation, but add support for explicit ascriptions to guide the type computation to a desired type. Conservativity with respect to LF has one qualification. In RSP, run-time errors like failure of pattern matching can occur, which result in an RSP term's evaluating to `Null`. The statement of conservativity is this: any RSP value of LF type is guaranteed to be an LF object, as long as it contains no `Null`s. This form of conservativity together with type safety guarantees the property mentioned above: in the absence of run-time errors, any putative LF object produced by evaluation of an RSP term is truly an LF object.

The paper informally introduces RSP (Section 3), and then shows how the language is used in practice on several example meta-programs (Section 4). While it is not hard to formalize the basic idea of RSP's type system, the exact formalization needed to achieve the meta-theoretic results is work in progress. The paper presupposes knowledge of LF (see, e.g., [11]).

2 Related Work

LF meta-programming plays a crucial role in several application domains. In the CVC project, a cooperating validity checker was instrumented to produce proofs in a variant of LF [18]. It was originally hoped that producing proofs would help catch bugs in CVC. This was actually quite rare, since most bugs were failures of completeness, where validity proofs are of no obvious relevance. Nevertheless, there were constantly bugs in CVC's proof-producing code. Given a straightforward implementation in C++, it is easy to write code which erroneously generates malformed proofs. Such an error is caught only when a malformed proof is produced for some input formula and then run through a proof checker. Tracking down the causes of such failed proofs is extremely time consuming.

Several approaches have been proposed for writing type-safe LF meta-programs. Appel and Felty use Twelf to implement partially correct tactics and decision procedures [1]. In Twelf, sets of LF types receive a computational interpretation as logic programs [13]. LF's typing establishes that any proof produced by a successful computation is guaranteed to check. The program may still fail due to run-time errors such as non-termination or match failure. The Delphin language of Schürmann is a pure functional language for meta-programming with LF encodings [17]. Delphin carefully places pattern-matching and recursion over LF for type safe manipulation for LF encodings. Other less closely related systems include Cayenne and Alf [2,9].

3 Overview of RSP

RSP is a proper extension of LF. We adopt the following notation for LF. We write `x:A => B` for dependent product type $\Pi^x A : B.$, and `x:A -> M` for $\lambda x : A. M$. We call the latter *representational* abstractions, since they will be used in RSP solely for representation using HOAS. Another kind of abstraction will be used for computation. In RSP's operational semantics, computation occurs in the bodies of representational abstractions, although not in the bodies of computational abstractions. Insofar as a representational abstraction is simply a parameterization of its body, evaluating that body without any argument given is not unreasonable. Notationally, if `x` does not occur in `B`, we write `A => B` instead of `x:A => B`. Application is written explicitly with infix `@`, and `x @ y` is sometimes written `x(y)` when `x` is a variable or constant.

The features RSP adds to LF are briefly these. RSP has dependently typed **expression attributes**, which can be read (`X.a`) and written (`X.a := Y`). The type of an attribute is just like a dependent function type, except with `=a>` instead of `=>`. Types of attribute reads are computed as for applications. The type system restricts attributes in attribute reads and attribute writes to be just constant symbols. Using such attributes, we support recursion by writing recursive equations: a functional expression containing attribute read $a.b$, say, is written into the b attribute of a.

The computational abstractions mentioned above are dependently typed **pattern abstractions**, typed with yet another arrow, `=c>`. These are of the form `x\P\`Δ `-> M`. Here, `x` is a name for the whole input to the abstraction, `P` is the pattern the input should match, Δ is a context for pattern variables in `P`, and `M` is the body of the abstraction. If Δ is empty we write `null` for it. Pattern abstractions are applied to target expressions by matching the pattern against the target. The notion of matching used in RSP is just syntactic first-order matching. **Deterministic choice** allows pattern abstractions of the same type to be combined: we write `M|N` for deterministic choice between `M` and `N`. An application of a |-expression is evaluated by using the first abstraction (from left to right) whose pattern matches the target expression. RSP uses `Null(A)` for match failure and also for reads of uninitialized attributes, for every type `A`.

RSP's pattern abstractions are inspired by those of Pure Pattern Type Systems (P^2TS) [3], with an important difference. In P^2TS, the range type of a pattern abstraction is allowed to depend on the pattern variables. Hence, the pattern and its context must become part of the domain type of the abstraction, and pattern abstractions receive types $\Pi P : \Delta . B$, where P is the pattern, Δ is the context for pattern variables, and B is the type of the body. Abstractions can still only be connected by choice if they have exactly the same type. Since patterns are part of types, this leads to the following serious drawback of the P^2TS approach: abstractions can only be connected if they are attempting to match the same pattern. This is a severe restriction, since

$$
\textit{(c-arrow-I)}\quad \frac{\begin{array}{l}\text{pattern } \Gamma\ \Delta\ P \\ \Gamma, \Delta \vdash P :: A \qquad \Gamma, \Delta, x = P \vdash M :: B \qquad \Gamma \vdash x : A \texttt{ =c> } B :: *\end{array}}{\Gamma \vdash x \backslash P \backslash \Delta \to M \ :: \ x : A \texttt{ =c> } B}
$$

$$
\textit{(rec-I)}\quad \frac{\Gamma \vdash M : A \qquad \Gamma, y = M \vdash N :: B \qquad \Gamma \vdash x : A, B :: *}{\Gamma \vdash (x = M, y.N) :: x : A, B}
$$

Fig. 1. Selected RSP typing rules

it means programs cannot use case analysis to take different action based on the form of expressions. The present approach solves this problem by not including patterns as part of the domain types of abstractions. But this requires the range type not to depend on pattern variables. The range type can still depend, however, on the name `x` for the entire target expression to which the pattern abstraction is being applied.

This approach to dependent pattern abstractions is enforced by the typing rule (c-arrow-I) in Figure 1. As mentioned above, the exact formalization of RSP is not yet complete, so this is just the essential idea: note that things like the definition of pattern used in the first premise must be formulated with great care. The third premise adds an equation to the context, which is used when checking convertibility. The fourth premise ensures that the pattern variables from Δ are not used in B.

Finally, it turns out to be a practical necessity to have some kind of pairing mechanism. This is mainly to allow pattern abstractions to perform simultaneous pattern matching on a dependently typed *bundle* of objects. Initially, RSP adopted dependent sum types, following [16]. It has become clear, however, that a limited form of **dependent record types** would lead to more readable meta-programs. This is because in some applications, it becomes necessary to manipulate rather large bundles, and it is easier to read code which refers to elements of a large bundle by name instead of by a sequence of projections. Fortunately, the uses of bundles in RSP does not seem to require some of the features which complicate record types. We do not need subtyping on record types, nor, apparently, manifest fields in record types. We adopt right-associating records as in [14]. We write $\{x : A, B\}$ for the right-associating record with leftmost field x and remaining fields B. Then `(x = M, y.N)` denotes the dependent record with leftmost field x storing element `M`, and remaining fields `N`, where `N` is allowed to use `y` as another name for `M`. This follows the approach originally proposed in [8], where each field has a label and an associated bound variable (to avoid variable capture during substitution). We often elide the binding and write just `(x = M, N)`. By using `x`

```
O : type.
IMP : (O => O => O).
FALSE : O.

Pf : (O => type).
Dn : (P : O => Pf(IMP @ (IMP @ (IMP @ P @ FALSE) @ FALSE) @ P)).
K : (P : O => Q : O => Pf(IMP @ P @ (IMP @ Q @ P))).
S : (P : O => Q : O => R : O =>
     Pf(IMP @ (IMP @ P @ (IMP @ Q @ R))
            @ (IMP @ (IMP @ P @ Q) @ (IMP @ P @ R)))).
MP : (P : O => Q : O => Pf(IMP @ P @ Q) => Pf(P) => Pf(Q)).

I : type.
EQUALS : (I => I => O).
Eqrefl : (x : I => Pf(EQUALS @ x @ x)).
Eqsymm : (x : I => y : I => Pf(EQUALS @ x @ y) =>
          Pf(EQUALS @ y @ x)).
Eqtrans : (x : I => y : I => z : I => Pf(EQUALS @ x @ y) =>
           Pf(EQUALS @ y @ z) => Pf(EQUALS @ x @ z)).
```

Fig. 2. LF signature: classical logic with equality (no quantifiers)

as an alias for `M` in `N` (which is discussed but not supported in [14]), we can often guide bottom-up type computation to some desired type for `M`. In some cases, however, we still need explicit ascriptions `M:A`. The typing rule should be essentially the (`rec-I`) rule of Figure 1.

RSP terms are parsed with the following precedences from tightest to loosest binding: attribute read and projections, application, record formation, arrows, and deterministic choice. So the first term below is fully parenthesized as the second (and evaluates to whatever value is stored for attribute `b` of expression `a`)

```
(x\a\null -> x.b | x\y\y:I -> x.c) @ a
((x\a\null -> (x.b)) | (x\y\y:I -> (x.c))) @ a.
```

4 Meta-Programming Examples

We consider two example meta-programs that manipulate LF encodings of proofs in classical first-order logic with equality. All the code has been type checked and run on sample inputs using a prototype implementation of RSP [1]. This prototype is written in Rogue, a version of the untyped Rewriting Calcu-

[1] This implementation does not support all the features of records yet, in particular field accesses. Versions of the examples using projections instead of field accesses have been checked and run on sample inputs.

```
1.  rank : (I =a> Int).
2.  findp : (x : I =a> {y:I, Pf(Equals @ x @ y)}).
3.  find : (base =a> x : I =c> {y:I, d:Pf(Equals @ x @ y)}).
4.
5.  uf.find := x \ y \ y : I ->
6.    Let(fx, x.findp,
7.    Ite(fx,
8.    Let(ffx, uf.find @ fx.1,
9.       x.findp := (y = ffx.1,
10.                  d = Eqtrans @ x @ fx.1 @ y @ fx.2 @ ffx.2)),
11.   Drop1(x.rank := 0, (y = x,
12.                       d = Eqrefl(x) : Pf(Equals @ x @ y))))).
```

Fig. 3. The RSP code for find

lus [19], which is essentially an untyped version of RSP. We encode our logic in a standard way as the LF signature of Figure 2. Our examples do not deal with quantifiers, so they are omitted for space reasons. Also, constructs to form first-order logic terms are omitted. The examples also make use of the following non-logical LF declarations, whose role is further explained below:

```
base : type.
uf : base.
dt : base.
```

4.1 Proof-Producing Union-Find

The first example is a proof-producing version of the well-known union-find algorithm (see, e.g., [4, Chapter 22]). Recall that this algorithm maintains disjoint sets of elements in balanced lazily path-compressed trees. The `union` operation takes two elements and merges their trees by making the root of the shallower one (as bounded by its *rank*) point to the root of the deeper one. The `find` operation takes an element `x` and returns the root of its tree. It modifies the pointers (called *find pointers*) encountered along the path from `x` to the root so that they all point directly to the root. Proof-producing `union` additionally takes in (the LF encoding of) a proof that the two given elements are equal. Proof-producing `find` additionally returns a proof that `x = r` where `x` is the input element and `r` is the root element which `find` returns. The underlying data structure is augmented so that every find pointer from a node `x` to a node `x.findp` has associated with it a proof that `x = x.findp`.

Figure 3 shows typing declarations and the code just for `find`. Lines are numbered for reference. The typing declarations declare attributes `rank` and `findp`, as well as `find` (lines 1-3). The latter is just so we can write a recursive definition, which occupies the rest of the Figure (lines 5-12). We implement find pointers using the `findp` attribute. The idea is that `x.findp` will store a dependent record of type `y:I, Pf(Equals @ x @ y)`. That is,

```
Let(x,M,N)     ≡   (x \ q \ q : type(M) -> N) @ M
Ite(M,N,N')    ≡   (Null(type(M)) -> N' |
                    q \ q2 \ q2 : type(M) -> N) @ M
Drop1(M,N)     ≡   Let(ignore, M, N)
```

Fig. 4. RSP abbreviations used in the examples

a record consisting of an individual `y` together with a proof that `x` equals `y`. Such a record is also what `uf.find` returns. We set `uf.find` to be a pattern abstraction taking in an individual `x` matching a pattern which is a single variable `y` (line 5). Since a variable matches anything, this pattern does not constrain the input to the abstraction at all (and syntactic sugar can be introduced to omit it). We first put `x`'s find pointer in temporary variable `fx` using a Let statement (line 6). We then use an Ite statement (if-then-else) to check (line 7) whether or not `x`'s find pointer is `Null` (at the appropriate type). This relies on the fact, mentioned above, that attributes without a stored value default to `Null`. Let and Ite forms (as well as Drop1, used on line 11) are abbreviations, given in Figure 4, where we write `type(M)` for the type in the current context of `M`.

Consider then the first case of the if-then-else statement (lines 8-10). We make a recursive call to `uf.find` on the first component of `fx`, which is the individual pointed to by `x`'s find pointer, and put the result in temporary `ffx` (line 8). Then we set `x`'s find pointer to be a new record, consisting of `ffx.1` (line 9), which by induction is the top element of the chain of find pointers from `fx.1`; and the appropriate transitivity proof (line 10). Note the careful use of y as the third argument to `Eqtrans`. This ensures that the type computed bottom-up for the record (i.e., the one being stored in `x.findp`) is `y:I, d:Pf(Equals @ x @ y)`, as required by the stated return type for `uf.find`. The "else" branch of the Ite expression (lines 11-12) sets `x`'s rank to 0 (for the benefit of `uf.union`), and then returns a record consisting of `x` and a reflexivity proof. Bottom-up type computation for `Eqrefl(x)` will compute the type `Pf(Equals @ x @ x)`. In order for the two branches of the Ite expression to have the same type, an ascription must be used (line 12) so that the reflexivity proof will be viewed as having type `Pf(Equals @ x @ y)`. Since `y` is an alias for `x` at this point, this ascription is legal.

4.2 Imperative Deduction Theorem

The union-find example constructs proofs but never applies a pattern abstraction to a proof to analyze it. In this second example, we consider a meta-program that does analyze proofs using pattern abstractions. For the Hilbert-style formulation of classical logic we have adopted (Figure 2), it is standard to prove the so-called Deduction Theorem by induction on the structure of proofs (cf. [20, Chapter 2]) with case analysis:

```
dedthm : (base =a> A : O =c> B : O =c>
         (Pf(A) => Pf(B)) =c> Pf(IMP @ A @ B)).

dedthm_h : (base =a> bridge : (u:O => Pf(u)) =>
            bundle : {A : O, B : O, d : Pf(B)} =c>
            Pf(IMP @ bundle.A @ bundle.B)).

dedthm_cached : (bundle : {A : O, B : O, d : Pf(B)} =a>
                 Pf(IMP @ bundle.A @ bundle.B)).
```

Fig. 5. Declarations for the Deduction Theorem

```
dt.dedthm := A:O \ null -> B:O \ null ->
 D : (Pf(A) => Pf(B)) \ null ->
   (bridge : (u:O => Pf(u)) ->
     dt.dedthm_h @ bridge @
        (q = A, p = B, d = D @ bridge(A) : Pf(p))
        : Pf(IMP @ A @ B))
   @ Null(u:O => Pf(u))
```

Fig. 6. Deduction Theorem, outer routine

Theorem 1 (Deduction Theorem) *If formula B is derivable possibly using assumption u of formula A, then the formula "A implies B" is derivable without assumption u.*

The inductive proof corresponds exactly to a certain recursive program, where case analysis is implemented by pattern matching. Such a program is a standard example for meta-programming in Twelf [11]. The Twelf program implementing the Deduction Theorem uses HOAS to represent the hypothetical judgment that `B` is derivable from assumption `A` as a representational (i.e., λ-) abstraction. Careful use of higher-order pattern unification enables computation to proceed beneath such abstractions.

We develop here an implementation of the Deduction Theorem in RSP. Since RSP supports imperative programming using attributes, we will actually be able to write an imperative version of this function, which caches intermediate results. Caching intermediate results is a simple but highly effective optimization in automated reasoning systems. In the case of the Deduction Theorem, we will cache intermediate proofs using an attribute `dedthm_cached`. Since the type of the cached proof, `IMP @ A @ B`, depends on both formulas `A` and `B`, we have to store cached results in the `dedthm_cached` attribute of dependent records of type `A:O, B:O, d:Pf(B)`. This explains the declared type for `dedthm_cached` in Figure 5.

Just as in Twelf, it will be necessary to compute under representational abstractions. RSP is able to achieve this using just first-order matching. We borrow a technique of Fegaras and Sheard, developed for implementing cata-

```
1. dt.dedthm_h := bridge : (u:O => Pf(u)) ->
2.  bundle : {A:O, B : O, d:Pf(B)} \ null ->
3.  Ite(bundle.dedthm_cached, bundle.dedthm_cached,
4.  bundle.dedthm_cached := ((A : O ->
5.  (B \ A \ null -> F \ bridge @ B \ null -> IMP_REFL |
6.
7.  B:O \ null ->
8.  (F \ MP @ P @ B @ d1 @ d2
9.     \ (P : O, d1 : Pf(IMP @ P @ B), d2 : Pf(P)) ->
10.    MP @ (IMP @ A @ P) @ (IMP @ A @ B)
11.       @ (MP @ (IMP @ A @ (IMP(P) @ B))
12.             @ (IMP @ (IMP @ A @ P) @ (IMP @ A @ B))
13.             @ (S @ A @ P @ B)
14.             @ (dt.dedthm_h @ bridge @
15.                  (x \ A, y \ (IMP @ P @ B), d1 : Pf(y))))
16.       @ (dt.dedthm_h @ bridge @ (x \ A, y \ P, d2 : Pf(y))) |
17.     D : Pf(B) \ null -> MP @ B @ (IMP @ A @ B) @
18.                          (K @ B @ A) @ D))) @
19.   bundle.A @ bundle.B @ bundle.d)).
```

Fig. 7. Deduction Theorem, main routine

morphisms over datatypes with embedded functions, to program with HOAS in RSP [5]. The function `dt.dedthm` of Figure 6 takes in the function from `Pf(A)` to `Pf(B)` representing the hypothetical judgment. It calls this function on a placeholder term `bridge(A)`, thus replacing (representations of) uses of the assumption `A` in the proof with the placeholder. The helper routine `dt.dedthm_h` of Figure 7 then operates on objects of type `Pf(B)` which may contain occurrences of the placeholder. Enountering the placeholder signals that the assumption is being used, and the appropriate action may be taken. One nice twist here is that unlike in [5], the placeholder does not need to be built in (either to our LF signature or to RSP). We simply introduce it using a representational abstraction (Figure 6, line 3). Since we compute in the bodies of representational abstractions, we will then call the helper (line 4) with the placeholder deployed in the term (line 5). Finally, the placeholder is eliminated after the helper is done computing by applying the representational abstraction to `Null` at the appropriate type (line 7). Bugs in our implementation might lead to occurrences of `Null` appearing in the resulting proof, but this is consistent with our statement of conservativity with respect to LF.

The main routine of the Deduction Theorem (Figure 7) has few surprises. The code begins by checking to see if there is a cached result, and uses it if so (line 3). Otherwise (lines 4-19), it sets the cached result to the appropriate proof, computed by applying cases to the parts of the input bundle. Recursive calls are needed (line 14 and line 16) just when the input proof is an application of `MP`. For typographic reasons the proof `IMP_REFL` for one of the cases (in line

```
MP @ (IMP @ A @ (IMP @ B @ B)) @ (IMP @ A @ B)
   @ (MP @ (IMP @ A @ (IMP(IMP @ B @ B) @ B))
         @ (IMP @ (IMP @ A @ (IMP @ B @ B)) @ (IMP @ A @ B))
         @ (S @ A @ (IMP @ B @ B) @ B)
         @ (K @ A @ (IMP @ B @ B))) @ (K @ A @ B)
```

Fig. 8. IMP_REFL (reflexivity of implication, where A = B)

5) appears in Figure 8. Note that this proof is in terms of A and B, but it is supposed to prove `IMP @ A @ A`. This is indeed what it proves, because at the point in Figure 7 where `IMP_REFL` is used (line 5), it is known that A and B are identical. This is because in line 5, the pattern abstraction matches iff the argument given for `B` matches pattern `A`. Bottom-up type computation for the proof of Figure 8 will, however, compute type `IMP @ A @ B`, which is just what we need to match the return type of `dt.dedthm_h`.

5 Mutable State and HOAS

The above examples combine mutable state and HOAS. Without some restrictions, this would quickly lead to failure of type preservation. For example, consider the term

```
x:I -> (a.b := x)
```

Since this is a representational abstraction, evaluation will occur in the body, causing variable `x` to be stored in attribute `b` of expression `a`. Reading this attribute subsequently returns an open term, which is hardly typable!

Our solution to this problem is based on the following very simple observation. Suppose that instead of the above term we had something like

```
x:I -> (x.b := x)
```

Then there would be no unsoundness, because outside the scope of this binding for `x`, we cannot reference `x`. Hence, we cannot attempt to read its `b` attribute.

We generalize this observation as follows. In an attribute write expression `x.a := y`, if the set of free variables FV(`x`) is a superset of FV(`y`), then we know that it cannot happen that a variable of `y` goes out of scope while `x.a` could still be evaluated. The above examples all are safe in this regard. For instance, take the attribute write in `uf.find` (Figure 3, lines 9-10). Assume by induction that terms `t` cannot evaluate to terms `t'` such that FV(`t'`) $\supsetneq$ FV(`t`). Then no term in the body of `uf.find` can evaluate to a term containing more than the free variables of `x`. Hence, the attribute write is safe. A similar observation applies to the attribute write in `dt.dedthm_h` (Figure 7, line 4). A suitable analysis can enforce this approach; in some cases, it appears some annotations may need to be supplied relating the free variables sets of different arguments to a function. This is the case with the `union` function of union-find, for example, whose code we omit for space reasons.

6 Conclusion and Future Work

This paper has presented work in progress on imperative LF meta-programming in Rogue-Sigma-Pi (RSP). RSP overlays LF with standard programming constructs, including syntactic pattern matching and unrestricted recursion. Imperative programming is supported through dependently typed attributes, which are very convenient for numerous examples, including those of proof-producing union-find and the imperative Deduction Theorem which were considered here. It is well-known that great care is required to combine programming constructs with LF. Imperative features pose special problems, particularly due to the interaction with HOAS. A conservative solution was proposed: we can store values with free variables in attributes as long as we know that the attribute read expressions become inaccessible at least as soon as the values do. The main future work is proving the meta-theoretic properties of type safety and conservativity with respect to LF for a formalization of the system.

References

[1] A. Appel and A. Felty. Dependent Types Ensure Partial Correctness of Theorem Provers. *Journal of Functional Programming*, 2002. to appear.

[2] Lennart Augustsson. Cayenne – a language with dependent types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250. ACM Press, 1998.

[3] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure patterns type systems. In *Principles of Programming Languages*. ACM, 2003.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[5] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294. ACM Press, 1996.

[6] N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A Syntactic Approach to Foundational Proof Carrying-Code. In *IEEE Symposium on Logic in Computer Science*, 2002.

[7] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[8] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st Symposium on Principles of Programming Languages*, pages 123–137, 1994.

[9] Lena Magnusson. *The Implementation of ALF—a Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers University of Technology and Göteborg University, 1995.

[10] G. Necula and P. Lee. Proof Generation in the Touchstone Theorem Prover. In David McAllester, editor, *17th International Conference on Automated Deduction*, 2000.

[11] F. Pfenning. *Logical Frameworks*, chapter 21. Volume 2 of Robinson and Voronkov [15], 2001.

[12] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation*, 1988.

[13] F. Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[14] Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002.

[15] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning.* Elsevier and MIT Press, 2001.

[16] J. Sarnat. LF-Sigma: The Metatheory of LF with Sigma types. Technical Report 1268, Yale CS department, 2004.

[17] C. Schürmann. Recursion for higher-order encodings. In *Proceedings of Computer Science Logic*, number 2142 in LNCS, pages 585–599, 2001.

[18] A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.

[19] A. Stump, R. Besand, J. Brodman, J. Hseu, and B. Kinnersley. From Rogue to MicroRogue. In *International Workshop on Rewriting Logic and Applications*, 2004.

[20] A. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge University Press, 2nd edition, 2000.

[21] D. Wu, A. Appel, and A. Stump. Foundational Proof Checkers with Small Witnesses. In D. Miller, editor, *5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2003.