# A Language-based Approach to Functionally Correct Imperative Programming

Edwin Westbrook, Aaron Stump, Ian Wehrman

Computer Science and Engineering, Washington University in Saint Louis
{ewestbro,stump,iwehrman}@cse.wustl.edu, http://cl.cse.wustl.edu

## Abstract

In this paper a language-based approach to functionally correct imperative programming is proposed. The approach is based on a programming language called RSP1, which combines dependent types, general recursion, and imperative features in a type-safe way, while preserving decidability of type checking. The methodology used is that of internal verification, where programs manipulate programmer-supplied proofs explicitly as data. The fundamental technical idea of RSP1 is to identify problematic operations as impure, and keep them out of dependent types. The resulting language is powerful enough to verify statically non-trivial properties of imperative and functional programs. The paper presents the ideas through the examples of statically verified merge sort, statically verified imperative binary search trees, and statically verified directed acyclic graphs.

***Categories and Subject Descriptors*** D.3 [*Software*]: Programming Languages

***General Terms*** Languages, Verification

***Keywords*** RSP, RSP1, Dependent Types, Program Verification

## 1. Introduction

Impressive progress in verification and analysis of imperative programs continues to be made in several research communities. In static analysis, techniques like shape analysis have been used to verify properties of the reference graph (e.g., [18, 27, 14]). Theorem proving techniques in higher-order logics have also been applied (e.g., [17]). Rapid development continues based on separation logic [26], a substructural logic that has proved convenient for stating properties of the reference graph.

In the present work, we develop an alternative, language-based approach to functionally correct imperative programming, based on the idea of *internal verification* [1, 2]. In internal verification, proofs are data in a dependently typed programming language. Functions are written to require proofs of their preconditions as additional arguments, and return proofs of their postconditions. Type checking ensures that proofs are manipulated soundly, guaranteeing partial correctness: If the program terminates and encounters no run-time errors, then the specified properties will hold. Dependent types are used for two reasons. First, the judgments-as-types principle allows specifications to be represented as types, with their proofs represented as objects of those types. Second, dependency allows a type checker to connect proofs and the data the proofs prove something about.

Dependent types are supported by a number of languages [22, 3, 16, 9, 7, 15, 6]. Including support for general recursion and imperative features while retaining desirable meta-theoretic properties like decidability of type checking is technically challenging. Twelf is the only system the authors know where this has been achieved, but the solution there depends heavily on the fact that logic programming is taken as the programming paradigm (more on this in Section 7). The technical challenges arise due to the fact that if arbitrary objects can index types, then unrestricted recursion in types can cause type checking to be undecidable (as some objects that index types might not terminate); and reads of mutable state in types are unsound, since the mutable state, and thus the types, can change over time.

The goal of this paper is to present a type-safe language, RSP1, that allows programming with proofs in the presence of unrestricted recursion and imperative features, while retaining decidable type checking. The key insight enabling this is purity: only objects which are considered pure are allowed to index types. Unrestricted recursion and imperative reads and writes are considered impure, and are banned from types. This concept could potentially be extended to other features that do not mix well with dependent types. The paper presents several examples of (imperative) programming with proofs in RSP1 (Section 3), including binary search trees where the binary search tree property is statically verified, and directed graphs which are statically verified to be acyclic. A pure functional example is also included, statically verified merge sort.

Technically, RSP1 is first-order (thus the "1" in its name): it does not allow functions or function application in its types. Lambda-abstractions are replaced by a more powerful pattern-matching facility, which is considered impure. This prevents the direct use of of Higher-Order Abstract Syntax [21] in the language, as it cannot appear in types. Despite the lack of this feature, many useful properties of algorithms and imperative data structures can be verified in the first-order setting of RSP1. Omitting lambdas from types also has the advantage that there is no need to consider $\beta$- or $\eta$-equivalence in type-checking. This greatly simplifies the proof of the decidability of type-checking, which is notoriously hard to prove in systems with $\beta$- and $\eta$-equivalence [12, 10]. This also makes it straightforward to compile RSP1 to machine code to OCaml, for which compilers to native code exist.

The rest of this document is organized as follows. Section 2 gives an overview of the language RSP1. Section 3 discusses how programming with proofs interacts with the purity and first-order restrictions, and presents the examples. Section 4 briefly describes our approach to compiling RSP1. Section 5 gives an in-depth ac-



*ICFP'05* September 26–28, 2005, Tallinn, Estonia.

| | | |
|---|---|---|
| Objects | $M ::=$ | $x \parallel c \parallel M_1 M_2 \parallel M :: T \parallel$ |
| | | let rec $D$ in $M \parallel R \parallel M.l \parallel$ |
| | | $\rho \parallel \text{null} \parallel M.c \parallel M_1.c := M_2$ |
| Types | $T ::=$ | $a \parallel T\ M \parallel \Pi^{\{r,c,a\}} x : T_1.T_2 \parallel RT$ |
| Kinds | $K ::=$ | $\text{type} \parallel \Pi^r x : T.K$ |
| Pattern Abstractions | $\rho ::=$ | $\epsilon \parallel x \backslash M_1 \backslash \Gamma \rightarrow M_2 \mid \rho$ |
| Records | $R ::=$ | $[] \parallel [l = M, x.R] \parallel [l = M, R]$ |
| Variable Definitions | $D ::=$ | $d_1 : T_1 = M_1, \ldots, d_n : T_n = M_n$ |
| Record Types | $RT ::=$ | $\{\} \parallel \{l : T, x.RT\}$ |
| Signatures | $\Sigma ::=$ | $\cdot \parallel \Sigma, a : K \parallel \Sigma, c : T$ |
| Contexts | $\Gamma ::=$ | $\cdot \parallel \Gamma, x : T$ |

**Figure 1.** RSP1 Syntax

count of the static semantics of the language. Section 6 describes the operational semantics of the language. Section 7 discussed related work. Finally, Section 8 concludes and gives directions for future work. The proofs of lemmas given in the text are given in the expanded, tech report version of this article. [30]

## 2. Language Overview

In this Section, we give an overview of RSP1 through its syntax. The first half of the section describes the constructs of the language and how they are used. We then go on to define some other important syntactic concepts of the language, such as purity, representational objects, what counts as first-order types, and patterns as they are allowed in pattern-matching.

The syntax of RSP1 is given in Figure 1. Each category of term is given an abbreviation, such as $M$ for terms or $T$ for types. Throughout this document, we will refer to the various categories of terms in the Figure by the letter(s) or symbol(s) given with them, possibly with numeric subscripts or primes, as in $M_1$ or $M'$. Other, non-numeric subscripts will be used for special restrictions of these categories, like the pure or representational objects, which are given below. We will also use $x$, $y$, and $z$ for variables, $c$ for object-level constants, $a$ for type-level constants, $l$ for record labels, and $d$ for "definition variables" (discussed below), again with possible subscripts or primes on any of these. Note that, since some of the constructs given are cannot be written in ASCII text (e.g. $\Pi^r$), there are a few instances where RSP1 code differs from this figure. These will be discussed where appropriate.

The central concepts in Figure 1 are the objects, the types, the kinds, the signatures, and the contexts, as in LF [11]. The objects level terms are the programs of RSP1. The well-formed programs of RSP1 have types, and the well-formed types have kinds. Signatures are used to type user-defined type- and term-constructors, while contexts are used to type variables. Starting at the top of this hierarchy are the standard kinds of LF: type, which classifies all the types that may be had by objects, such as record types and some user-defined types; and $\Pi^r x : T.K$, the kind of types indexed by objects of type $T$. (In LF, kind-level abstractions are normally written $\Pi x : T.K$, but we add the $r$ superscript, which stands for "representational," to distinguish from two other sorts of abstraction in RSP1.) The construct $\Pi^r x : T.K$ appears in RSP1 code as `x:T => K`.

At the type level are: type constants, $a$; type applications, $T\ M_1 \ldots M_n$, indicating a type of family $T$ indexed by objects $M_1$ through $M_n$; representational abstraction types, $\Pi^r x : T_1.T_2$, for typing functions in the LF fragment (which, as RSP1 has no lambdas, are an application of a constant to less than its required number of arguments); computational abstraction types, $\Pi^c x : T_1.T_2$, for typing pattern abstractions, the computational element of the language; attribute abstraction types, $\Pi^a$, for typing attributes (see below); and dependent record types, $\{l_1 : T_1, x_1.\{\ldots\{l_n : T_n, x_n.\{\}\}\ldots\}\}$ (the right-associating dependent record types of [25]), in which the types of later fields may be indexed by the names of earlier ones. The abstraction types, $\Pi^r x : T_1.T_2$, $\Pi^c x : T_1.T_2$, and $\Pi^a x : T_1.T_2$, are written `x:T1 => T2`, `x:T1 =c> T2`, and `x:T1 =a> T2`, respectively. Note that the type constants, type applications, and representational abstractions are exactly the type constructs of LF.

The objects, being the programs of RSP1, have more constructs. The first three constructs of the objects given in Figure 1, variables, constants, and application, are standard, though in RSP1, application is written the same for the LF fragment and for applying pattern abstractions. The next two constructs, type ascription (written $M :: T$) and let rec, are also standard (see e.g. [23]). Type ascription allows the user to ascribe a type to a term, and is useful because the type inferencing algorithm for RSP1 is not complete. The let rec construct, similar to that found in ML, allows general recursion. It is slightly nonstandard in that it requires the special $d$ variables, which are just like normal variables, except they are impure. This is because they represent recursive terms, so evaluation of them may not terminate.

Records in RSP1 are of two sorts: dependent records and independent, dependently typed records. The latter are the right-associating records of [25], and are written $[l_1 = M_1, [\ldots[l_n = M_n, []]\ldots]]$ in RSP1, where the $l_i$ are the record labels and the $M_i$ are the fields of the record. Dependent records allow later fields of a record to refer back to previous fields. These are written $[l_1 = M_1, x_1.[\ldots[l_n = M_n, x_n.[]]\ldots]]$, where the $x_i$ variables can be used in later fields to refer to earlier ones. These two styles can be freely intermixed. To increase readability, the user can omit the variables and all but the outermost brackets, which is syntactic sugar for dependent records with variables named the same as their fields. For example, the code

```
[l1 = c, l2 = f (d l1)]
```

makes a dependent record wherein the second `l1` (in the `l2` field) refers back to the value of the first field. Record selection, written $M.l$, is then used to extract elements of records.

The pattern abstractions, abbriviated $\rho$, are the central computational element of the language. These are written $x_1 \backslash M_1 \backslash \Gamma_1 \rightarrow M_1' | \ldots | x_n \backslash M_n \backslash \Gamma_n \rightarrow M_n' | \epsilon$. When applied to an argument, a pattern abstraction matches each $M_i$ (the patterns) against the argument, starting from the first, until one matches, i.e. until it finds some substitution for the variables in $\Gamma_i$ (the pattern variables) to make the pattern equal to the term. If, say, pattern $M_i$ matches the argument, then the whole argument is substituted for $x_i$ into $M_i'$ (the body), while the subterms of the argument that matched the pattern variables are substituted for them into body as well. The resulting term is then evaluated. For instance, the identity function on type `T` could be written `x \ y \ y:T -> x` or `x \ y \ y:T -> y` as anything matches the single pattern variable `y`. Note that we elide the empty pattern, $\epsilon$, in code. As a second example, if we create the type `nat` of natural numbers, with constructors `zero` and `succ` (so e.g. 3 would be represented by `succ succ succ zero`), then the pattern function

```
x \ zero \ -> zero | x \ succ y \ y:nat -> y
```

would be the standard predecessor function (taking the predecessor of 0 to be 0).

| | | |
|---|---|---|
| Rep. Object | $M_{\text{rep}} ::=$ | $x \parallel c \parallel M_{\text{rep}}\, M_{\text{rep}}$ |
| Pure Object | $M_{\text{pure}} ::=$ | $M_{\text{rep}} \parallel M_{\text{pure}}.l \parallel$ |
| | | $[l = M_{\text{pure}}, x.M_{\text{pure}}] \parallel$ |
| | | $[l = M_{\text{pure}}, M_{\text{pure}}] \parallel []$ |
| Pure Pattern | $M_{\text{ppat}} ::=$ | $M_{\text{rep}} \parallel [l = M_{\text{ppat}}, M_{\text{ppat}}] \parallel []$ |
| Pattern | $M_{\text{pat}} ::=$ | $M_{\text{ppat}} \parallel \text{null}$ |
| Zero-Order Rep. Type | $T_{ZRep} ::=$ | $a \parallel T_{ZRep}\, M_{\text{rep}}$ |
| Zero-Order Type | $T_Z ::=$ | $T_{ZRep} \parallel \{l : T_Z, x.T_Z\} \parallel \{\}$ |
| Rep. Type | $T_{\text{rep}} ::=$ | $T_{ZRep} \parallel \Pi^r x : T_{ZRep}.T_{\text{rep}}$ |

**Figure 2.** Other Syntactic Concepts of RSP1

If no pattern in a pattern abstraction matches, then it returns the special term null. null is considered a run-time error in RSP1, and behaves like an exception: records that contain null as a field, as well as applications that contain null as either the function or the argument, evaluate to null themselves (except that pattern abstractions can match a null argument by having null as a pattern). Thus no value, other than pattern abstractions, will have null as a proper subterm. This means that run-time errors "trickle up" to the top of a term, and can immediately be detected by checking against null.

Finally, the attribute operations, $M.c$ and $M.c := M'$, constitute the imperative feature of RSP1. An attribute is a user-defined constant of type $\Pi^a x : T_1.T_2$. These act like hashtables: if $c$ is an attribute, then $M.c$ looks up the value associated with $M$ in $c$'s hashtable and $M.c := M'$ updates this value to be $M'$. Attributes are slightly different than references, the standard imperative feature, but experience with RSP1 has shown attributes to be a useful mechanism for capturing dependencies: the fact that attributes have product types (as suggested by the $\Pi$) means that the type of $M.c$ can be indexed by $M$.

In addition to these constructs, we shall find two pieces of syntactic sugar very useful. The first, `let x = M1 in M2`, does local variable binding. This stands for for `(x \ y \ y:T -> M2) M1`, where `T` is the type of `M1`, and `y` is a fresh variable. The second, `if M1 then M2 else M3`, tests `M1` against null. This form stands for `(x \ null \ -> M3 | x \ y \ y:T -> M2) M1`, where `T` is the type of `M1` and `x` and `y` are new variables.

We shall also define some other important concepts in RSP1 in terms of syntax. These definitions are given in Figure 2. The representational objects and types are the LF fragment of the objects and types, respectively. Note the zero-order restriction on the argument type of $\Pi^r$-abstractions. The pure objects include the representational objects plus records and record selects. These will be the only objects allowed to index types. The patterns, made of records and representational objects or a single null, are the only patterns allowed in pattern abstractions. Thus a pattern abstraction cannot match on the form of another pattern abstraction. Finally, the zero-order types, which omit the abstraction types, will be the only argument types allowed for computational and attribute abstractions types.

The concept of purity, as mentioned above, is important in ensuring type soundness and decidability. The only computation allowed in types is record selects from pure records, as these will always terminate and their value is not dependent on the store. We ban attribute operations and let rec from the pure objects, since, as discussed above, these cannot be allowed to index types. We ban null, since it is not really data that we wish to prove properties about. We also ban pattern abstractions, both so we do not have to consider problems like $\beta$- and $\eta$-equivalence, and because it is unclear how to apply pattern abstractions to the variables that can appear in types. Purity will be important in the programming methodology in Section 3, since the programmer must take care not to apply dependently typed functions to impure objects. It will also be important in formalizing the metatheory of the language, as there will have to be two different substitution lemmas, one for pure and one for impure objects.

## 3. Programming with Proofs

In this section, we discuss programming with proofs in RSP1. We begin by discussing a methodology for programming with proofs in RSP1, which both solves the technical problems posed by the concept of purity and offers some conveniences not necessarily found in other methodologies. This will be illustrated by a simple list example. Then, in the subsections, we give examples of more complicated examples which illustrate the kinds of properties of data structures we can prove in RSP1. The purpose of this section will be to see both how purity affects programming with proofs, and how useful properties of data structures and code can be represented in a first-order language like RSP1.

Consider the example of lists of some arbitrary data (here given the type `data`) where it is important to prove properties about the sizes of the lists. These can be represented with the following type and term constructors:

- `nat :: type.` Natural numbers in unary, with the usual constructors `zero` and `succ`.
- `list :: n:nat => type.` Lists of `nat`s of length `n`. The term constructors are:

```
nil :: list zero;;
cons :: n:nat => data =>
        list n => list (succ n);;
```

- `plus :: nat => nat => nat => type.` The type `(plus x y z)` is intended to be inhabited iff `x` + `y` = `z`. Its term constructors, embodying the usual recursive definition of addition, are:

```
plus_zero :: x:nat => plus zero x x;;
plus_succ :: x:nat => y:nat => z:nat =>
             plus y x z =>
             plus (succ y) x (succ z);;
```

Next, consider the operation of appending two lists. We wish to write an `append` function that returns, along with the concatenation of two lists, a proof that the length of this result is the sum of the lengths of the two inputs. The code for this `append` is given in Figure 3. `append` is defined to be a recursive function. (The `rec` keyword means `append` is a top-level, recursive definition, and is equivalent to `let rec append:T = M in M`, where `T` is replaced by the type declared for `append` and `M` is replaced by the outer pattern abstraction in the Figure.) `append` takes in two records, `l1` and `l2`, each of which contain a list and its length, in the `l` and `n` fields, respectively. The length must come first, as the type of the list is indexed by it. Also, note that it is more convenient in this case to take the second list first, as we wish the inner case to discriminate against whether the first list is empty or not. `append` returns a record with a list, its length, and a proof that this length is the sum of the lengths in `l1` and `l2`. Note that this last proof is essentially the specification of `append`. This illustrates the first point of our methodology, that terms should generally be bundled in a record with the terms that index their types. This makes it easy for the user to see, in the type of the function, the dependencies between

```
rec append :: l2:{n:nat, l:list n} =c>
              l1:{n:nat, l:list n} =c>
              {n:nat, l:list n,
               deriv:plus l2.n l1.n n} =
  l2 \ dummy_var \ dummy_var:{n:nat, l:list n} ->
    ( l1 \ [n=zero, l = nil] \ ->
        [n=l2.n, l=l2.l, deriv=plus_zero l2.n]
    | l1 \ [n=succ tail1_len,
            l = cons n1 data1 list1_tail]
         \ tail1_len:nat,data1:data,list1_tail ->
       let res = append l2 [n=tail1_len,
                            l = list1_tail] in
         [n=succ res.n,
          l = cons res.n data1 res.l,
          deriv = plus_succ l2.n tail1_len
                    res.n res.deriv]);;
```

**Figure 3.** Append for lists with length

the arguments and their types, as well as the specification of the function. This could be done without dependently-typed records, but it would require declaring a new type family and constructor for each input and output bundle, which quickly becomes tedious, and separates the specification of the function over multiple locations.

The body of the function works like a normal append function: if the second argument is the empty list (matched by the first `l1` clause, which matches `n` against `zero` and `l` against `nil`), the function simply returns the second list; otherwise, it recursively calls `append` on the tail of the list (the appearance of `append` on the third-to-last line), and prepends the first element of `l1` to the resulting list (the `cons` on the second-to-last line). The main difference is that we also construct the `deriv` field in the returned record, which is the proof that the length of the returned list is the sum of the lengths of the input lists. Since the recursive call to `append` is recursive, it is not pure. This length of this result, `res.n`, needs to index two types, the type of the `cons` expression and the type of the `plus` expression. The reason this example still type checks is because of the `let`. Inside the body of the `let`, the `res` variable is pure, as it is a normal variable. Thus it is ok that the types of some expressions are indexed by it. The type of the whole body, however, is not indexed by `res`, as the individual types of the fields of the returned record are "swallowed up" by the dependent record type. Thus, the type of the whole `let` expression need not be indexed by the impure recursive call. This is the other key to our methodology, using `let` to hide impure terms from being in types, then bundling them in dependently-typed records to hide the impurity from entering into the return type.

A final consideration is null. null can inhabit any type, so is a vacuous proof of any property. Since null "trickles up" in any term, however, there cannot be proofs that are erroneous because they contain null as a proper subterm. Also, since our methodology involves the bundling of data with its proofs in records, if any of the proofs a function returns contain null, the whole record will evaluate to null, and the function will not return any data. Instead, this is interpreted as a run-time error. So our desired property of functional correctness can be made precise: if a function that programs with proofs, according to our methodology, returns a non-null value, then the proofs it returns are guaranteed to be well-formed. Note that this motivates the use of null in our methodology. If a program wishes to prove some properties of data which could potentially fail to hold, for instance if it is input incorrectly, the program can simply return null in cases where it ascertains that the property fails.

In the following subsections, we consider the following three examples of programming with proofs in RSP1. The first is a version of mergesort which returns, in addition to the sorted output list, a proof that that list is sorted and has exactly the same elements as the input list. The third example is an implementation of imperative binary search trees where we statically verify that the binary search tree property is maintained. This example does not verify the structural property that the reference graph starting from any node is really a tree (we verify the binary search tree property even without this structural property). In the fourth example, we give an example of statically verifying a structural property of the reference graph, namely that of being acyclic. Other examples in progress but not discussed here include a proof-producing automated reasoning tool, where propositional proofs are encoded as a term-indexed datatype [13]; and mesh-manipulating algorithms from computer graphics, where a mesh is encoded as a datatype indexed by its Euler characteristic [5].

### 3.1 Merge Sort

The implementation of proof-producing merge sort in RSP1 is based on the following term-indexed datatypes:

- `nat :: type.` Natural numbers in unary, as for the `append` example.
- `list :: type.` Lists of `nat`s. We elect here not to index the type of lists by a length, for simplicity. The term constructors are `nil` and `cons`, of the usual types.
- `lte :: nat => nat => type.` Natural number less-than. This type has these term constructors:

  ```
  lte_start :: x:nat => lte x x;;
  lte_next :: x:nat => y:nat =>
              lte x y => lte x (succ y);;
  ```

- `sorted :: list => type.` The property on lists of being sorted. We rely on the following three term constructors for this type. The third one, for example, can be read as saying that for all nats `n` and `m`, and for all lists `l`; if `n` is less than `m` and `(cons m l)` is sorted, then so is `(cons n (cons m l))`.

  ```
  sorted_nil :: sorted nil;;
  sorted_cons1 :: n:nat => sorted (cons n nil);;
  sorted_cons2 :: n:nat => m:nat => l:list =>
                  lte n m =>
                  sorted (cons m l) =>
                  sorted (cons n (cons m l));;
  ```

- `occurs :: nat => list => list => type;;` The intended meaning of `(occurs n l1 l2)` is that `n` occurs in `l1`, and `l2` is the result of removing one occurrence of `x` from `l1`. We omit the (simple) term constructors here.
- `multiset_union :: list => list => list => type;;` The intended meaning of `multiset_union l l1 l2` is that the multiset of elements in `l` is equal to the multiset union of the multiset of elements in `l1` with the multiset of elements in `l2`. The term constructors for this type are:

  ```
  mu_nil :: multiset_union nil nil nil;;
  mu_cons1 :: n:nat => l:list => l1:list =>
              l1p:list => l2:list =>
              occurs n l1 l1p =>
              multiset_union l l1p l2 =>
              multiset_union (cons n l) l1 l2;;
  mu_cons2 :: n:nat => l:list => l1:list =>
              l2:list => l2p:list =>
              occurs n l2 l2p =>
              multiset_union l l1 l2p =>
              multiset_union (cons n l) l1 l2;;
  ```

With these types, we can state the types of the three critical recursive functions needed for mergesort:

```
split :: l:list =c> {a:list, b:list,
                     MU:multiset_union l a b};;
merge :: q:{l1:list, D1:sorted l1,
            l2:list, D2:sorted l2} =c>
          {l:list, D:sorted l,
           MU:multiset_union l q.l1 q.l2};;
mergesort :: l1:list =c> {l:list, D:sorted l,
              MU:multiset_union l1 l nil};;
```

The `split` function is responsible for splitting an input list `l` into two output lists `a` and `b` of roughly equal size (note that this latter property is not specified here and hence not statically checked). It additionally produces a proof that `l` is the multiset union of `a` and `b`. The `merge` function takes in lists `l1` and `l2`, together with proofs that those lists are `sorted`, and produces the merged output list `l`, together with a proof that `l` is `sorted` and the `multiset_union` of `l1` and `l2`. Finally, `mergesort` takes in a list `l1`, and returns an output list `l`, together with a proof that `l` is `sorted` and the `multiset_union` of `l1` and `nil`. This last condition is, of course, sufficient to guarantee that `l` and `l1` have exactly the same elements.

Space reasons prohibit giving all the code (87 lines) for this example, but we consider a representative piece from `merge`, shown in Figure 4. This is the case where the two input lists are both non-empty (as shown in the pattern which makes up the first line of the Figure; note that the types of the pattern variables are omitted for space reasons). The body of this case begins by calling a helper function `nat_comp` to compare the heads `n` and `m` of the two lists. If `n` is smaller, `nat_comp` returns a term of type `(lte n m)`. Otherwise, it returns `null`. Depending on which of these two cases occurs, one or the other recursive call to `merge` is made (in either the then-part or the else-part of the if-then-else). The two recursive calls to `merge` both rely on a helper lemma `sublist_sorted`, which takes in a non-empty sorted list and returns a proof that its immediate sublist is sorted. Both branches of the if-then-else then build a record with fields `l` for the merged list, `D` for the proof that `l` is sorted, and `MU` for the proof that `l` is the multiset union of the input lists. Lemmas `extend_sorted1` and `extend_sorted2` are also used. The lemma `extend_sorted1` takes in the proofs that the input lists are sorted, as well as the proof that the result of the recursive call is sorted and is the multiset union of the second list and the immediate sublist of the first list. The lemma returns a proof that consing `n` onto the list obtained from the recursive call is sorted. We note here that the implementation of mergesort relies on 250 lines of proofs of lemmas like `extend_sorted1`. A few lemmas concerning multiset union remain to be proved. These are currently just expressed as additional axioms (via declarations of additional term constructors).

### 3.2 Imperative Binary Search Trees

We consider implementing imperative binary search trees in such a way that the binary search tree property is ensured statically by RSP1's type checking. The binary search trees are imperative in the sense that the left and right subtrees of a particular tree are reached by following mutable pointers from the node at the top of the tree. Trees can, of course, be implemented as an inductive datatype in a language with user-declared datatypes (like RSP1 or ML). But imperative trees have the advantage that subtrees can be modified in place, without requiring the entire tree to be rebuilt (as would be the case with a datatype of trees). For simplicity, the data in our binary search tree will just be natural numbers in unary (as defined above).

```
| q \ [l1 = cons n l1, D1 = D1,
       l2 = cons m l2, D2 = D2]
    \ D1,l1,m,n,l2,D2 ->
  let C = nat_comp n m in
    if C then
      let R = merge [l1 = l1,
                     D1 = sublist_sorted
                            [n = n,
                             l = l1, D = D1],
                     l2 = q.l2, D2 = D2]
      in
        [l = cons n R.l,
         D = extend_sorted1 n m l2 D2 C
               [l1 = l1, D1 = D1,
                l = R.l, D = R.D, MU = R.MU],
         MU = mu_cons1 n R.l (cons n l1) l1
                q.l2 (occurs_start n l1) R.MU]
    else
      let R = merge [l1 = q.l1, D1 = D1,
                     l2 = l2,
                     D2 = sublist_sorted
                            [n = m,
                             l = l2, D = D2]]
      in
        [l = cons m R.l,
         D = extend_sorted2 n l1 m D1
               (nat_comp m n)
               [l2 = l2, D2 = D2,
                l = R.l, D = R.D, MU = R.MU],
         MU = mu_cons2 m R.l q.l1 (cons m l2)
                l2 (occurs_start m l2) R.MU]
```

**Figure 4.** Recursive case of `merge`

The binary search tree property we would like to verify statically is that every piece of data stored in the left subtree of a tree whose top node stores data `d` must be less than or equal to `d`; and every piece of data stored in the right subtree must be greater than or equal to `d`. Note that allowing data equal to `d` to appear in either subtree makes the development simpler. Note also that we will not actually try to enforce the structural property of being a tree, as opposed to a proper graph (although see Section 8). To express our binary search tree property as an RSP1 type, we cannot rely on being able to speak directly about the reference graph, as is often done in static analysis (e.g., [18, 27, 14]). RSP1's types may not contain attribute reads or any other impure expressions, and hence cannot refer directly to the reference graph. The approach we follow instead is to express local invariants which imply the binary search tree property. The local invariants are statically enforced. The fact that they imply the binary search tree property is not (in any obvious way) expressible in RSP1. Hence, we cannot prove in RSP1 that the local invariants indeed imply the global property, and must argue that outside the system.

The basic plan is to build our binary search tree out of nodes, connected by `bst_left` and `bst_right` attributes. We associate (in a way explained shortly) two numbers `l` and `u` with each node `n`. These are intended to be a lower bound and upper bound, respectively, on all the data stored in the subtree rooted at `n`. Then we enforce the following local invariants on the pointers from node `n` to another node `n'`, storing data `d'` and having associated lower and upper bounds `l'` and `u'`:

- If `n'` is the left child of `n`, then `l` $\leq$ `l'` and `u'` $\leq$ `d`. That is, the left subtree's lower bound must be the same or tighter than the current tree's lower bound, and the left subtree's upper bound

must be less than or equal to the data at the top of the current tree.

- If n' is the right child of n, then u' $\leq$ u and d $\leq$ l'. That is, the right subtree's upper bound must be the same or tighter, and the right subtree's lower bound must be greater than or equal to the data at the top of the current tree.

We take the following term-indexed datatype for the type of binary search tree nodes:

```
node :: l:nat => d:nat => u:nat => type;;
```

The type (node l d u) is the type for nodes with associated lower bound l and upper bound u, and data d stored in the node. We include d as an index to the type so we can refer to it when we express the local invariants. To construct nodes, we use the following term constructor, which requires proofs that l $\leq$ d $\leq$ u, as well as a unique id (to ensure the graph is not cyclic, although as mentioned, we do not statically check that property):

```
mknode :: l:nat => d:nat => u:nat => id:nat =>
          lte l d => lte d u => node l d u;;
```

Now we may express the local invariants with the following attribute declarations:

```
bst_left :: parent:{l:nat, d:nat,
                    u:nat, n:node l d u} =a>
            {l:nat, d:nat, u:nat, n:node l d u,
             p1:lte parent.l l,
             p2:lte u parent.d};;

bst_right :: parent:{l:nat, d:nat,
                     u:nat, n:node l d u} =a>
             {l:nat, d:nat, u:nat, n:node l d u,
              p1:lte u parent.u,
              p2:lte parent.d l};;
```

These declarations state that bst_left and bst_right are attributes of dependent records containing the indices l,d, and u, as well as the node itself. We cannot make them attributes just of nodes, due to the presence of the indices. The declarations state that proofs of the local invariants discussed above are included as members of the records stored in the attributes. This means that whenever an attribute is written, the proofs of the local invariants must be supplied. And similarly, those proofs are available whenever an attribute is read. By the soundness of our encoding of judgments of natural number less-than as the term-indexed datatype lte, the existence of these proofs for every edge in the reference graph shows that the local invariants always hold.

To show that the local invariants imply the binary search tree property (which we must do outside RSP1), it suffices to show that the putative lower and upper bounds on the data reachable from each node really hold. The argument cannot proceed by induction on the structure of trees, since, as mentioned above, we are not enforcing the structural property of being a tree. Nothing prevents the pointers from being incorrectly set to create cycles or reconvergent paths. Nevertheless, the binary search tree property still holds. We prove that for every length $k$, for every node n, and for every node n' reachable by a simple path of length $k$ from n, the data stored at n' is within the bounds associated with n. The proof is by induction on $k$. The data stored at n itself is within the bounds, since mknode requires proofs of those containments. For the inductive case, suppose n' is reachable from n with a simple path of length $k+1$. This must be by following either bst_left or bst_right to reach a node n''. The node n' is thus reachable from n'' using a simple path of length $k$. Then by the induction hypothesis, we know the data stored at n' is within the bounds associated with n''. But by the enforcement of the local invariants and transitivity of $\leq$, this implies that the data is within the bounds associated with n.

Based on our data structure, we can implement insertion into a binary search tree:

```
rec bst_insert
  :: x:nat =c> q:{l:nat, d:nat,
                  u:nat, n:node l d u}
          =c> bst_insert_ret_t q.l q.d q.u x
  = ...
```

The return type of bst_insert uses a new term-indexed datatype, introduced to return information about how the insertion proceeded. The information is needed to construct suitable proofs when recursive calls to bst_insert return. The term constructors for bst_insert_ret_t correspond to three possible scenarios that could occur when doing the insertion:

1. l $\leq$ x $\leq$ u, and the input node to the recursive call remains the root of the updated tree.
2. x $\leq$ l, and the input node is no longer the root of the updated tree (since the lower bound must now be x). The node which has everything the same as the input node except that the lower bound is x is the new root.
3. u $\leq$ x, and the input node is again no longer the root of the updated tree (since the upper bound must now be x). The node which has everything the same as the input node except that the upper bound is x is the new root.

When bst_insert makes a recursive call, it uses the information returned as follows. If the input node is no longer the root of the updated tree, the bst_left or bst_right (as appropriate) attribute of the node currently being processed must be reset to point to the node which is the new root. Then the current call to bst_insert must itself return the appropriate instance of bst_insert_ret_t. This instance is readily determined. For example, if the recursive call was made in order to insert x into the right subtree of the current node, and if that recursive call returned an instance corresponding to case 1 or case 2 above, then the current call returns an instance corresponding to case 1: in either case, the data x is still within the current node's bounds.

Note that we are not checking, and cannot in any obvious way check, that bst_insert actually inserts the data into the tree. This might be considered something like a *liveness* property: the reference graph is actually modified in a certain way. But we are enforcing what might be considered a *safety* property: the reference graph is guaranteed always to have a certain property, however it may be modified.

### 3.3 Statically Enforcing a Structural Property

The preceding example showed how to enforce statically a non-structural property of the reference graph in RSP1. Here, we give a simple example where a structural property is statically enforced. The property is that a reference graph determined by two attributes, dag_left and dag_right, is acyclic. As in the preceding Section, we must devise local invariants that imply this global property of the reference graph. We rely on the simple fact that a finite directed graph is acyclic (a *dag*) if its edge relation is contained in some well-founded ordering. For the implementation in RSP1, we take natural number less-than as our well-founded ordering. More complex (computable) orderings could be supported in a similar way. We will associate with each of our dag nodes a natural number. The well-founded ordering on dag nodes is then the ordering of those nodes by the associated natural number. In RSP1, as for the example in the preceding Section, we index the type of dag nodes by the associated natural number. We will enforce statically the local invariant that all dag nodes reachable in one or more

```
type nat =  Null_nat | Zero | Succ of nat
type list = Null_list | Nil | Cons of nat * list
type lte =  Null_lte | Lte_start of nat
          | Lte_next of nat * nat * lte
type sorted = Null_sorted | Sorted_nil
            | Sorted_cons1 of nat
            | Sorted_cons2 of nat * nat * list
                                  * lte * sorted
type mu = Null_mu | Mu_nil
        | Mu_cons1 of nat * list * list * list
                          * list * occurs * mu
        | Mu_cons2 of nat * list * list * list
                          * list * occurs * mu

let record =
    let _num = Succ Zero in
    let _data = Cons _num Nil in
    let _order = Lte_start _num in
        [num = _num; data = _data; order = _order]
in record.data;;
```

**Figure 5.** Intermediate compiled representation of types and records in OCaml

steps from a given dag node have a smaller associated number. Hence, each dag node's number will be a strict upper bound on the numbers associated with dag nodes reachable in one or more steps. For concreteness, we implement dags where each dag node stores a natural number (unrelated to the bound associated with the dag node). The term-indexed datatype we need, with its term constructor, is:

```
dag :: b:nat => type;;
mkdag :: b:nat => data:nat => dag b;;
```

We then specify our local invariant in these attribute declarations, which use a term-indexed datatype `lt` for strict natural-number less-than (we omit its simple declarations):

```
dag_left :: parent:{b:nat, d:dag b} =a>
            {b:nat, d:dag b, p:lt b parent.b};;
dag_right :: parent:{b:nat, d:dag b} =a>
             {b:nat, d:dag b, p:lt b parent.b};;
```

Using these definitions, it is straightforward to implement conversion from (functional) trees to dags with maximal sharing. We declare an inductive datatype of trees in the usual way, and then implement:

```
rec dagify :: x:tree =c> {b:nat, d:dag b} = ...
```

As in the preceding Section, we do not here statically check the liveness property that the `dag` returned is suitably related to the input `tree`. But we do enforce the safety property that the reference graph starting from any node created by this method is contained within the natural number less-than relation (and hence really a dag, although that implication must again be verified outside the system).

## 4. Compilation

We translate well-typed RSP1 programs into Objective Caml (OCaml) and leverage the OCaml compiler to generate native executables. OCaml was chosen as an intermediate language primarily because many of its syntactic and operational aspects closely mirror those of RSP1. Furthermore, OCaml's strong type system allows the intermediate representation to use much of the original type information to ensure correctness of the translation and the compiler itself.

OCaml's type system does not support dependent types. So, RSP1 types are compiled into OCaml types in which the indexing terms are elided (see Fig. 5). Most features in RSP1 translate with little adjustment to their natural analogues in OCaml. Computational functions are represented by the OCaml `function` construct. The `let` and `let rec` constructs are identical. RSP1 patterns are similar to those in OCaml, but, unlike in RSP1, all variables in OCaml patterns are treated as pattern variables. The translation from RSP1 patterns must therefore include a pattern guard (a `when` clause) to constrain the values of pattern variables with corresponding local variables already in the context. Terms are created by data constructor application in a manner that is essentially identical in both languages.

Other RSP constructs are more flexible than their OCaml counterparts or simply do not map directly to any high-level OCaml feature. For example, RSP attribute declarations are compiled into a pair of functions that manage reads and writes to a hash table for the attribute. Records in RSP allow intra-record prior field lookups, unlike in OCaml. In the intermediate representation, field contents are compiled into a series of `let`-defined OCaml expressions culminating in an OCaml record that gathers the definitions and allows for later access in the typical fashion (see Fig. 5).

The constant $\mathrm{null}$ may adopt any type in an RSP program, but since the OCaml type system disallows polymorphic constants, a special `Null` constructor is added to the signature for each data type. More complex types are "$\eta$-expanded" into null objects at compile-time — null records become records of the appropriate type with null fields and null functions become functions that return a null object of the appropriate type regardless of the arguments it is applied to. Testing for null in an if-then-else RSP1 expression is translated into a test for the appropriate null object in OCaml. [1]

## 5. Static Semantics

To formalize the static semantics of RSP1, we must first define the valid signatures and contexts. Rules for these judgments are given in Figure 6. Note that any constant in a valid signature must either have representational type or be an attribute. This is the only place we shall mention these judgments explicitly: all typing rules in the sequel implicitly require that all signatures and contexts involved are valid. We also assume that all variables in a context are always distinct; it will always be possible to assure this with alpha-conversion.

$$\frac{}{\vdash \cdot\ \mathsf{sig}} \qquad \frac{\vdash \Sigma\ \mathsf{sig} \quad \Sigma;\cdot \vdash K : \mathrm{kind}}{\vdash \Sigma, a : K\ \mathsf{sig}}$$

$$\frac{\vdash \Sigma\ \mathsf{sig} \quad \Sigma;\cdot \vdash T_{\mathrm{rep}} : \mathrm{type}}{\vdash \Sigma, c : T_{\mathrm{rep}}\ \mathsf{sig}}$$

$$\frac{\vdash \Sigma\ \mathsf{sig} \quad \Sigma;\cdot \vdash \Pi^a x : T_1.T_2 : \mathrm{type}}{\vdash \Sigma, c : \Pi^a x : T_1.T_2\ \mathsf{sig}}$$

$$\frac{\vdash \Sigma\ \mathsf{sig}}{\Sigma \vdash \cdot\ \mathsf{ctxt}} \qquad \frac{\Sigma \vdash \Gamma\ \mathsf{ctxt} \quad \Sigma;\Gamma \vdash T : \mathrm{type}}{\Sigma \vdash \Gamma, x : T\ \mathsf{ctxt}}$$

**Figure 6.** Valid RSP1 Signatures and Contexts

The rules for typing types and kinds are given in Figure 7. As mentioned above, only $\Pi$-abstractions with zero-order argument types are allowed. Also, $\Pi^r$-abstractions require representational

[1] Compiled RSP does not yet support the correct behavior of evaluating a record with a single null to a null record.

types for argument and result types, to keep them inside the representational fragment. Finally, note that, as promised above, only types indexed by pure objects will be considered well-formed.

$$\frac{}{\Sigma;\Gamma \vdash \text{type} : \text{kind}}\ \textsf{t-type-kind}$$

$$\frac{\Sigma;\Gamma \vdash T_{ZRep} : \text{type} \quad \Sigma;\Gamma, x : T_{ZRep} \vdash K : \text{kind}}{\Sigma;\Gamma \vdash \Pi^r x : T_{ZRep}.K : \text{kind}}\ \textsf{t-pi-kind}$$

$$\frac{a : K \in \Sigma}{\Sigma;\Gamma \vdash a : K}\ \textsf{t-const-type}$$

$$\frac{}{\Sigma;\Gamma \vdash \{\} : \text{type}}\ \textsf{t-empty-rec-type}$$

$$\frac{\Sigma;\Gamma \vdash T_{ZRep} : \text{type} \quad \Sigma;\Gamma, x : T_{ZRep} \vdash T_{\text{rep}} : \text{type}}{\Sigma;\Gamma \vdash \Pi^r x : T_{ZRep}.T_{\text{rep}} : \text{type}}\ \textsf{t-r-pi-type}$$

$$\frac{\Sigma;\Gamma \vdash T_Z : \text{type} \quad \Sigma;\Gamma, x : T_Z \vdash T : \text{type}}{\Sigma;\Gamma \vdash \Pi^{c,a} x : T_Z.T : \text{type}}\ \textsf{t-\{c,a\}-pi-type}$$

$$\frac{\Sigma;\Gamma \vdash T : \Pi^r x : T_{ZRep}.K \quad \Sigma;\Gamma \vdash M_{\text{pure}} : T_{ZRep}}{\Sigma;\Gamma \vdash T M_{\text{pure}} : [M_{\text{pure}}/x]K}\ \textsf{t-type-app}$$

$$\frac{\Sigma;\Gamma \vdash T : \text{type} \quad \Sigma;\Gamma, x : T \vdash RT : \text{type}}{\Sigma;\Gamma \vdash \{l : T, x.RT\} : \text{type}}\ \textsf{t-rec-type}$$

**Figure 7.** RSP1 Type- and Kind-Level Typing

In order to define object-level typing, we need two more judgments. The first is type equivalence, written $\vdash T_1 = T_2$. The rules for type equivalence are given in Figure 8. Most of these correspond directly to structural equivalence of the two types. Note that record types are considered equivalent up to $\alpha$-conversion on the variables they bind. The interesting case is for type application, which requires the argument objects to be equivalent. This is a separate judgment, written $\vdash M_1 = M_2$, and has only one rule, eq-obj. This rule requires the two objects to evaluate to the same object: $\Rightarrow$ is the single-step evaluation relation, given in Section 6, and $\Rightarrow^*$ is the reflexive-transitive closure of this relation. This evaluation is also in the empty store, as object equivalence should not depend on values in the store. For more on the evaluation relation, see Section 6. Note that only pure objects can be considered equivalent, which will be sufficient for our purposes, as only pure objects can enter into types.

The use of evaluation to the same normal form as the basis for object equivalence is justified, because this coincides with the customary, more declaratively defined equivalence relation. We could define object equivalence as the set of equational consequences of the equational versions of the rules e-rec-sel1 and e-rec-sel2 (Figure 14). But if we orient those equations as in the Figure, the resulting rewrite system is terminating and clearly locally confluent. Hence, it is convergent by Newman's Lemma, and equivalent to the equational theory by Birkhoff's Theorem [4]. Then any strategy for applying the oriented equations, including the call-by-value strategy of our evaluation relation, is sound and complete for the equational theory. Note that our notion of equivalence does not depend on typing. This is one instance where the first-order nature of RSP1 greatly simplifies the type theory. In standard LF with $\beta$- and $\eta$-equivalence, confluence does not hold for ill-typed terms. Thus

$$\frac{\vdash \cdot; M_{\text{pure}-1} \Rightarrow^* \cdot; M \quad \cdot; M_{\text{pure}-2} \Rightarrow^* \cdot; M}{\vdash M_{\text{pure}-1} = M_{\text{pure}-2}}\ \textsf{eq-obj}$$

$$\frac{}{\vdash a = a}\ \textsf{eq-const} \qquad \frac{}{\vdash \{\} = \{\}}\ \textsf{eq-empty-rec}$$

$$\frac{\vdash T_1 = T_2 \quad \vdash M_1 = M_2}{\vdash T_1 M_1 = T_2 M_2}\ \textsf{eq-app}$$

$$\frac{\vdash T_{11} = T_{21} \quad \vdash T_{12} = T_{22}}{\vdash \Pi^{\{r,c,a\}} T_{11}.T_{12} = \Pi^{\{r,c,a\}} T_{21}.T_{22}}\ \textsf{eq-\{r,c,a\}-pi}$$

$$\frac{\vdash T_1 = T_2 \quad \vdash RT_1 = RT_2}{\vdash \{l : T_1, x.RT_1\} = \{l : T_2, x.RT_2\}}\ \textsf{eq-rec-type}$$

**Figure 8.** RSP1 Equivalence

proving confluence requires notions of typing, and is much more complicated.

The second judgment we require is record type selection, the rules for which are given in Figure 9. This judgment, written $\vdash \textsf{RTSel}_l\ M\ RT\ T$, is meant to indicate that selecting label $l$ from object $M$ with type $RT$ will result in an object of type $T$. If $l$ is the first label in $RT$, then the required type is the first type in $RT$, as suggested by the rule rtsel-base. If, however, $l$ is not the first label in $RT$, then, since $RT$ is a dependent record type, the first element of $M$ must be substituted for the first variable in $RT$ into the second and later field types in $RT$. Since we require only pure objects to appear in types, then either $M$ should be pure, or it should not be substituted into the record type. Thus we get the pure and impure rules in the Figure. To ensure coherence of the two rules, we use the question of whether the variable $x$ is free in $RT$ to distinguish which rule is applicable.

$$\frac{}{\vdash \textsf{RTSel}_l\ M\ \{l : T, x.RT\}\ T}\ \textsf{rtsel-base}$$

$$\frac{x \in FV(RT) \qquad l_1 \neq l_2 \quad \vdash \textsf{RTSel}_{l_1}\ M_{\text{pure}}\ ([M_{\text{pure}}/x]RT)\ T_1}{\vdash \textsf{RTSel}_{l_1}\ M_{\text{pure}}\ \{l_2 : T_2, x.RT\}\ T_1}\ \textsf{rtsel-pure}$$

$$\frac{\vdash \textsf{RTSel}_{l_1}\ M\ RT\ T_1 \quad x \notin FV(RT) \quad l_1 \neq l_2}{\vdash \textsf{RTSel}_{l_1}\ M\ \{l_2 : T_2, x.RT\}\ T_1}\ \textsf{rtsel-impure}$$

**Figure 9.** RSP1 Record Type Selection

The object-level typing rules for RSP1 are given in Figure 10. These include standard rules for constants, variables, ascriptions, and conversions, and a rule typing $\text{null}$ at any type. Pattern abstractions are only well-typed when all patterns follow the syntactic restrictions of Figure 2. Note that the requirement that $\epsilon$ only be typed by valid $\Pi^c$-types ensures inductively that the $\Pi^c$-type of any pattern abstraction is a valid type in just the context $\Gamma$. In particular, this ensures that none of the variables in $\Gamma'$, for any part of a pattern abstraction, appear in the resulting type of the abstraction. Following these are pure and impure rules for applications and attribute reads, similar to the pure and impure rules for RTSel discussed above. The Figure gives rules for dependent and independent records, the latter being equivalent to that given in [25] and the former being a straightforward adaptation to dependent records. Record selects are typed using RTSel, which, as discussed above, substitutes record selects of earlier labels of $M$ into the types of later labels. Finally, the Figure gives straightforward rules for attribute writes, empty records, and $\text{let rec}$.

$$\frac{x : T \in \Gamma}{\Sigma; \Gamma \vdash x : T}\ \textsf{t-var} \qquad \frac{c : T \in \Sigma}{\Sigma; \Gamma \vdash c : T}\ \textsf{t-const}$$

$$\frac{\Sigma; \Gamma \vdash T : \text{type}}{\Sigma; \Gamma \vdash \text{null} : T}\ \textsf{t-null} \qquad \frac{\Sigma; \Gamma \vdash M : T}{\Sigma; \Gamma \vdash (M :: T) : T}\ \textsf{t-asc}$$

$$\frac{\Sigma; \Gamma \vdash M : T \quad \Sigma; \Gamma \vdash T' : \text{type} \quad \vdash T = T'}{\Sigma; \Gamma \vdash M : T'}\ \textsf{t-conv}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash \rho : \Pi^c x : T_Z.T \\ \Sigma; \Gamma, \Gamma' \vdash [M_{pat}/x]M : [M_{pat}/x]T \\ \Sigma; \Gamma, \Gamma' \vdash M_{\text{pat}} : T_Z \\ FV(M_{\text{pat}}) = \text{Vars}(\Gamma')\end{array}}{\Sigma; \Gamma \vdash x \backslash M_{\text{pat}} \backslash \Gamma' \to M \mid \rho : \Pi^c x : T_Z.T}\ \textsf{t-rho}$$

$$\frac{\Sigma; \Gamma \vdash \Pi^c x : T_z.T : \text{type}}{\Sigma; \Gamma \vdash \epsilon : \Pi^c x : T_z.T}\ \textsf{t-epsilon}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash M : \Pi^{r,c} x : T_1.T_2 \\ \Sigma; \Gamma \vdash M_{\text{pure}} : T_1 \quad x \in FV(T_2)\end{array}}{\Sigma; \Gamma \vdash M\ M_{\text{pure}} : [M_{\text{pure}}/x]T_2}\ \textsf{t-\{r,c\}-pure-app}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash M_1 : \Pi^{r,c} x : T_1.T_2 \\ \Sigma; \Gamma \vdash M_2 : T_1 \quad x \notin FV(T_2)\end{array}}{\Sigma; \Gamma \vdash M_1\ M_2 : T_2}\ \textsf{t-\{r,c\}-impure-app}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash c : \Pi^a x : T_Z.T \\ \Sigma; \Gamma \vdash M_{\text{pure}} : T_Z \quad x \in FV(T_2)\end{array}}{\Sigma; \Gamma \vdash M_{\text{pure}}.c : [M_{\text{pure}}/x]T}\ \textsf{t-attr-read-pure}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash c : \Pi^a x : T_Z.T \\ \Sigma; \Gamma \vdash M : T_Z \quad x \notin FV(T)\end{array}}{\Sigma; \Gamma \vdash M.c : T}\ \textsf{t-attr-read-impure}$$

$$\frac{\Sigma; \Gamma \vdash M_1.c : T \quad \Sigma; \Gamma \vdash M_2 : T}{\Sigma; \Gamma \vdash M_1.c := M_2 : T}\ \textsf{t-attr-write}$$

$$\frac{}{\Sigma; \Gamma \vdash [] : \{\}}\ \textsf{t-empty-rec}$$

$$\frac{\Sigma; \Gamma \vdash M : RT \quad \vdash \mathsf{RTSel}_l\ M\ RT\ T}{\Sigma; \Gamma \vdash M.l : T}\ \textsf{t-rec-select}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash [M/x]R : [M/y]RT \\ \Sigma; \Gamma \vdash M : T \quad \Sigma; \Gamma, y : T \vdash RT : \text{type}\end{array}}{\Sigma; \Gamma \vdash [l = M, x.R] : \{l : T, y.RT\}}\ \textsf{t-record}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash R : [M/y]RT \\ \Sigma; \Gamma \vdash M : T \quad \Sigma; \Gamma, y : T \vdash RT : \text{type}\end{array}}{\Sigma; \Gamma \vdash [l = M, R] : \{l : T, y.RT\}}\ \textsf{t-indep-record}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma, d_1 : T_1, \ldots, d_n : T_n \vdash M : T \\ \Sigma; \Gamma, d_1 : T_1, \ldots, d_n : T_n \vdash M_i : T_i\end{array}}{\begin{array}{l}\Sigma; \Gamma \vdash \text{let rec } d_1 : T_1 = M_1, \ldots, \\ \qquad d_n : T_n = M_n \text{ in } M : T\end{array}}\ \textsf{t-let-rec}$$

**Figure 10.** RSP1 Object-Level Typing

Many standard structural properties of dependent type systems hold for RSP1, such as weakening and validity. Substitution, however, is nonstandard, because impure objects cannot enter into types. In RSP1, we in fact have two Substitution Lemmas, for pure and impure objects:

LEMMA 5.1. **(Pure Substitution)**
*If* $\Sigma; \Gamma \vdash M : T$, $x \notin FV(\Gamma)$, $M$ *is pure, and* $\Sigma; \Gamma, x : T \vdash M' : T'$, *then* $\Sigma; \Gamma \vdash [M/x]M' : [M/x]T'$.

LEMMA 5.2. **(Impure Substitution)**
*If* $\Sigma; \Gamma \vdash M : T$, $x \notin FV(\Gamma)$, $\mathcal{D}$ *is a derivation of* $\Sigma; \Gamma, x : T \vdash M' : T'$, *and* $x$ *is not free in any type in* $\mathcal{D}$, *then* $\Sigma; \Gamma \vdash [M/x]M' : T'$.

The pure form of substitution is similar to the pure typing rules, in that $M$ can only be substituted into a type if it is pure. The impure form requires a much stronger condition: not only can $x$ not occur in $T'$ if $M$ is impure, but it cannot occur free in *any* type in the deduction $\mathcal{D}$. This is because the argument type of an abstraction typing could contain $x$, even though $T'$ does not, since these argument types "disappear" below the line of their respective elimination rules (i.e. the typing rules for applications and attribute reads).

Type and object equivalence are also decidable, the first following from the second, and the second following from the fact that the evaluation relation is deterministic, so easily Church-Rosser, and and terminating on the pure objects.

LEMMA 5.3. **(Decidability of Equivalence)**
*For any* $T_1$, $T_2$ *and* $M_1$, $M_2$, *it is decidable whether* $\vdash T_1 = T_2$ *and whether* $\vdash M_1 = M_2$.

Given the decidability of equivalence, it is straightforward to implement a sound, but not complete, type inferencing procedure, using local type inference [24]. The problem with completeness lies in the t-record and t-indep-record rules of Figure 10: the substitutions below the line make it impossible to know, from just structural information, what instances of $M$ in $R$ should be replaced by $x$ in $RT$. The basic idea of the local type inference algorithm we use is that when typing applications, the type of the functional term is synthesized, and its domain type is used to guide type checking of the argument. This means that in the common case where records are passed as arguments to recursively defined functions, we check that the supplied record can indeed have the domain type of the function.

Our procedure, which incorporates a few further ideas, works well in practice, and the code that has been written in RSP1 so far rarely needs to make use of ascriptions. Further details of the local type inference algorithm are beyond the scope of this paper. Note that the algorithm currently does not compute omitted types for bound variables; those must still be supplied by the programmer. We conjecture that a complete type inference algorithm for RSP1 should be achievable, since the number of distinct types a dependent record can have in RSP1 is finite. Indeed, this observation shows that the non-determinism of the present typing rules is bounded, and hence type checking is decidable. In more general settings, there can be infinitely many incomparable types (an example is given in [28]).

## 6. Operational Semantics

To define the operation semantics of RSP1, we first define the values and the stores, given in Figure 11. The values, as in all languages, represent the possible final results of a computation. In RSP1, these include records, pattern abstractions, representational objects, and null, though note that, as mentioned above, null cannot

| | | |
|---|---|---|
| Record Values | $RV$ ::= | $[] \parallel [l = V_{\text{pure}}, RV]$ |
| Representational Values | $V_{\text{rep}}$ ::= | $c \parallel V_{\text{rep}}\, V_{\text{rep}}$ |
| Pure Values | $V_{\text{pure}}$ ::= | $V_{\text{rep}} \parallel RV$ |
| Values | $V$ ::= | $V_{\text{pure}} \parallel \rho \parallel \text{null}$ |
| Stores | $\mu$ ::= | $\cdot \parallel \mu, V_1.c \mapsto V_2$ |

**Figure 11.** RSP1 Operational Syntax

be a proper subterm of a value. The stores are necessary because of the attributes. They associate attribute expressions with their values. Whenever an attribute is read, it is retrieved from the current store, and whenever it is written, the current store is updated.

Stores also need to be well-typed. The rules for typing stores are given in Figure 12. The interesting rule, opt-store-add, simply ensures that the referenced value, $V_2$, has the same type as the corresponding attribute read expression, $V_1.c$, has. Note that we do not need a pure and an impure version of this rule, because values are always pure.

$$\frac{}{\Sigma \vdash \cdot}\ \text{opt-store-empty}$$

$$\frac{\begin{array}{ll}\Sigma \vdash \mu & c : \Pi^a x{:}T_z.T \in \Sigma \\ \Sigma; \cdot \vdash V_1 : T_z & \Sigma; \cdot \vdash V_2 : [V_1/x]T\end{array}}{\Sigma \vdash \mu, V_1.c \mapsto V_2}\ \text{opt-store-add}$$

**Figure 12.** RSP1 Operational Typing

The operational semantics of RSP1 are given in Figures 13 and 14 in terms of a small-step semantics. The small-step evaluation judgment, $\mu; M \Rightarrow \mu'; M'$, describes the evaluation of an object $M$ in the context of $\mu$, the current store, which, as discussed above, gives the current values of the attributes.

The rules in Figure 13 describe the congruences, and are thus straightforward. An interesting case is e-rec-congr3, which evaluates dependent records to independent records. Also note that these rules define a deterministic evaluation order.

The rules in Figure 14 describe the action of each of the object-level constructs of RSP1. Most of these are as expected: record selects retrieve the value for the particular label, ascriptions are removed, attribute reads retrieve the necessary value from the store, attribute writes update the store, and let rec operates as usual. null is returned whenever no other value is appropriate, including when null or $\epsilon$ are applied to a value, when a record select acts on null, or when an attribute is read that does not have a corresponding value.

Applying a non-empty pattern abstraction requires testing whether the argument matches the outer pattern. This is the meaning of the **match** function; $\mathbf{match}(V_1, \Gamma, V_2)$ is the substitution for the variables in $\Gamma$ that, when applied to $V_1$, obtains $V_2$. If such a match exists for a pattern and an argument, $[V_2/x, \sigma]$ is applied to the body, where $\sigma$ is the given substitution. Otherwise, if no such substitution exists, written $\mathbf{match}(V_1, \Gamma, V_2)\uparrow$, then the outer pattern is stripped from the pattern abstraction, so that the next pattern can be tried. Note that, as a special case, only the pattern null can match null, i.e. a pattern variable cannot match null. This is because null is impure, but the output type of a pattern abstraction might depend on its argument, and substituting null in as the argument would put it into a type. If we think of null as an exception, this means the pattern null is really a catch statement.

Our pure and impure rules, along with our first-order syntactic restrictions, ensure that our static semantics is sound with respect to our operational semantics. This is proved via the standard Preservation and Progress Lemmas, proofs of which are deferred to the Appendix.

$$\frac{\mu; M_1 \Rightarrow \mu'; M_1'}{\mu; M_1 M_2 \Rightarrow \mu'; M_1' M_2}\ \text{e-app-congr1}$$

$$\frac{\mu; M_2 \Rightarrow \mu'; M_2'}{\mu; V_1 M_2 \Rightarrow \mu'; V_1 M_2'}\ \text{e-app-congr2}$$

$$\frac{\mu; M \Rightarrow \mu'; M'}{\mu; M.l \Rightarrow \mu'; M'.l}\ \text{e-rec-sel-congr}$$

$$\frac{\mu; M \Rightarrow \mu'; M'}{\mu; M.c \Rightarrow \mu'; M'.c}\ \text{e-attr-read-congr}$$

$$\frac{\mu; M_1 \Rightarrow \mu'; M_1'}{\mu; M_1.c := M_2 \Rightarrow \mu'; M_1'.c := M_2}\ \text{e-attr-write-congr1}$$

$$\frac{\mu; M \Rightarrow \mu'; M'}{\mu; V.c := M \Rightarrow \mu'; V.c := M}\ \text{e-attr-write-congr2}$$

$$\frac{\mu; M \Rightarrow \mu'; M'}{\mu; [l = M, x.R] \Rightarrow \mu'; [l = M', x.R]}\ \text{e-rec-congr1}$$

$$\frac{\mu; M \Rightarrow \mu'; M'}{\mu; [l = M, R] \Rightarrow \mu'; [l = M', R]}\ \text{e-rec-congr2}$$

$$\frac{}{\mu; [l = V, x.R] \Rightarrow \mu; [l = V, [V/x]R]}\ \text{e-rec-congr3}$$

$$\frac{\mu; R \Rightarrow \mu'; R'}{\mu; [l = V, R] \Rightarrow \mu'; [l = V, R']}\ \text{e-rec-congr4}$$

$$\frac{}{\mu; [l = \text{null}, RV] \Rightarrow \mu; \text{null}}\ \text{e-rec-null1}$$

$$\frac{}{\mu; [l = V_{\text{pure}}, \text{null}] \Rightarrow \mu; \text{null}}\ \text{e-rec-null2}$$

**Figure 13.** RSP Operational Semantics Part 1 (Congruence)

LEMMA 6.1. **(Preservation)**
*If $\Sigma \vdash \mu$, $\Sigma; \Gamma \vdash M : T$, and $\mu; M \Rightarrow \mu'; M'$, then $\Sigma \vdash \mu'$ and $\Sigma; \Gamma \vdash M' : T$.*

LEMMA 6.2. **(Progress)**
*If $\Sigma \vdash \mu$ and $\Sigma; \cdot \vdash M : T$, then either $M$ is a value, or $\mu; M \Rightarrow \mu'; M'$, for some $\mu'$ and $M'$.*

THEOREM 6.1. **(Type Safety)**
*If $\Sigma; \Gamma \vdash M : T$, $\Sigma; \Gamma \vdash \mu$, and $\mu; M \Rightarrow^* \mu'; M'$, then $M'$ is either a value or can evaluate another step.*

## 7. Related Work

There has been much research in dependently typed languages and in verifying programs with them. For discussion of the latter, see, for example, [1, 2]. As for the former, many of these (for instance, [16, 9, 15, 8, 19]) are strongly normalizing. This is an important property in showing them correct. None of these languages support any sort of effects or mutable state, which is not surprising, as many of them are intended more as proof assistants than as programming languages.

The Cayenne language [3], on the other hand, is not strongly normalizing. Unfortunately, this percolates up to its types, causing

$$\frac{}{\mu; \mathrm{null}\ V \Rightarrow \mu; \mathrm{null}}\ \text{e-null-app}$$

$$\frac{}{\mu; \epsilon\ V \Rightarrow \mu; \mathrm{null}}\ \text{e-epsilon-app}$$

$$\frac{}{\mu; M_{\mathrm{rep}}\ \mathrm{null} \Rightarrow \mu; \mathrm{null}}\ \text{e-rep-null-app}$$

$$\frac{\mathbf{match}(V_1, \Gamma, V_2) = \sigma}{\mu; (x \backslash V_1 \backslash \Gamma \rightarrow M \mid \rho) V_2 \Rightarrow \mu; [V_2/x, \sigma]M}\ \text{e-rho-app1}$$

$$\frac{\mathbf{match}(V_1, \Gamma, V_2) \uparrow}{\mu; (x \backslash V_1 \backslash \Gamma \rightarrow M \mid \rho) V_2 \Rightarrow \mu; \rho V_2}\ \text{e-rho-app2}$$

$$\frac{}{\mu; M :: T \Rightarrow \mu; M}\ \text{e-ascription}$$

$$\frac{}{\mu; \mathrm{null}.l \Rightarrow \mu; \mathrm{null}}\ \text{e-rec-sel-null}$$

$$\frac{}{\mu; [l = V, RV].l \Rightarrow \mu; V}\ \text{e-rec-sel1}$$

$$\frac{l_1 \neq l_2}{\mu; [l_1 = V, RV].l_2 \Rightarrow \mu; RV.l_2}\ \text{e-rec-sel2}$$

$$\frac{V_1.c \mapsto V_2 \in \mu}{\mu; V_1.c \Rightarrow \mu; V_2}\ \text{e-attr-read}$$

$$\frac{V_1.c \notin Dom(\mu)}{\mu; V_1.c \Rightarrow \mu; \mathrm{null}}\ \text{e-attr-read-null}$$

$$\frac{}{\mu; V_1.c := V_2 \Rightarrow \mu[V_1.c \mapsto V_2]; V_2}\ \text{e-attr-write}$$

$$\frac{D \equiv x_1 : T_1 = M_1, \ldots, x_n : T_n = M_n}{\begin{array}{l}\mu; \mathrm{let\ rec}\ D\ \mathrm{in}\ M \Rightarrow \\ \mu; [\ldots, \mathrm{let\ rec}\ D\ \mathrm{in}\ M_i/x_i, \ldots]M\end{array}}\ \text{e-let-rec}$$

**Figure 14.** RSP Operational Semantics Part 2 (without Congruence)

type-checking to be undecidable. Cayenne is the only other language we know of that has a construct similar to null, namely $\bot$. $\bot$ inhabits all types, just like null, but, unlike null, it does not "trickle up" in terms. Thus, in Cayenne, a proof can be incorrect by containing $\bot$ at any arbitrary point in the term, which can be difficult to check.

A different approach to dependent types is Dependent ML [31]. Dependent ML restricts the terms allowed to index types to constraint domains: the paper uses arithmetic over the integers as an example. The type system of Dependent ML can then express constraints over these domains, such as one integer being greater than another, and can solve for properties of these constraints, which end up proving properties of the code. This is different from the goals of RSP1, which involve letting the user prove arbitrary properties of the data she manipulates.

Yet another approach we consider here is Twelf [22], a logic programming language built on LF. Twelf supports unrestricted recursion, but has a decidable type-checking problem. This is possible without the notion of purity because logic programs cannot be explicitly called in types. Twelf also supports a form of mutable state, through dynamically added clauses. The main difference between RSP1 and Twelf is thus the difference of paradigm: functional programming with reference-like features versus logic programming with dynamic clauses.

A final approach that is similar to RSP1 is ATS [6]. ATS is a pattern-matching language for programming with proofs which supports unrestricted recursion but has a decidable type-checking problem. This is achieved by separating the terms into the "proof terms" which encode proofs and the "dynamic terms" which allow for more powerful computation such as recursion. ATS also supports pointers, and can reason about them using "stateful views" (see [32]). The main difference from RSP1 is that ATS is a much more elaborate type system, with linear and intuitionistic types, the two systems for static and dynamic terms, datatypes, dataviews, etc. Our approach to RSP1 has been to find more of a "sweet spot" between expressiveness and simplicity of the type system.

## 8. Conclusion and Future Work

We have seen a language, RSP1, which combines dependent types with imperative and computational features. Using this language, we can implement examples where properties of data and even local properties of the reference graph can be enforced statically. This makes it relatively straightforward to implement examples like statically verified merge sort and binary search trees where the binary search tree property is statically verified. The type theory for this language has two important features. First, it is first-order, meaning that it does not contain lambda-expressions and does not allow abstraction types in argument positions. The latter is achieved with syntactic restrictions on the forms of the argument types of abstraction types. Second, only pure objects, which contain no use of the computational or imperative features of the language, are allowed into types. This is achieved with pure and impure versions of rules that need to substitute objects into types. These developments greatly simplify the type theory; the first-order restriction means there is no $\eta$-reduction, allowing for a deterministic evaluation relation. When combined with the restriction of only allowing pure objects into types, this makes the proof of the decidability of equivalence checking straightforward. Only allowing pure objects into types also ensures soundness of the type system in the face of imperative features.

For future work, we would like to establish a sound and complete type inferencing procedure for RSP1. This would both remove ascriptions from the language and establish the decidability of type-checking. Such an inference procedure would need to handle the special case of a pattern abstraction with record patterns applied to a record. This would probably require step similar to first-order unification in the procedure.

Also, we intend to add coverage checking and simple termination checking [29] and support for proof irrelevance [20]. This would allow the use of RSP1 pattern abstractions as proofs of metatheoretic properties of an object logic, which would not need to be executed at run-time. They do not need to be executed if we can determine that they would always succeed (using coverage and termination checking) and that their results are never analyzed by any code other than more proof-irrelevant code. This piece of future work is practically quite important, to avoid having to execute lemmas at run-time.

Finally, we would like to do more examples of verifying safety properties of imperative data structures. For example, it should be possible to enforce, through local invariants, the safety property that the reference graph from any node is a tree. One way to do this would be to associate an id with each node (in addition to the node's data). Then the same idea as for the local invariants of the binary search tree can be used, except that we require the id at a node to be strictly greater than all ids reachable by going left and strictly less than all ids reachable by going right. Proof irrelevance is likely to be important here, since inserting a new node into the tree would generally require updating the ids at many (if not all)

nodes. Naturally, we would need to slice away such code after type checking to get an efficient implementation.

**Acknowledgments:** Thanks to Joel Brandt, Robert Klapper, and Li-Yang Tan for many discussions about RSP1. Thanks also to Erran Li for discussion of the binary search tree and dag examples.

## References

[1] T. Altenkirch. Integrated verification in Type Theory. Lecture notes for a course at ESSLLI 96, Prague, 1996. available from the author's website.

[2] A. Appel and A. Felty. Dependent Types Ensure Partial Correctness of Theorem Provers. *Journal of Functional Programming*, 14(1):3–19, Jan. 2004.

[3] L. 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.

[4] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[5] J. Brandt. What a Mesh: Dependent Data Types for Correct Mesh Manipulation Algorithms. Master's thesis, Washington University in Saint Louis, 2005. In preparation.

[6] C. Chen and H. Xi. Combining Programming with Theorem Proving. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, Tallinn, Estonia, September 2005.

[7] R. Constable and the PRL group. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

[8] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[9] G. Dowek, A. Felty, H. Herbelin, and G. Huet. The Coq proof assistant user's guide. Technical Report 154, Inria-Rocquencourt, France, 1993.

[10] H. Goguen. A syntactic approach to eta equality in type theory. In *Principles of Programming Languages (POPL)*, pages 75–84, 2005.

[11] 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.

[12] R. Harper and F. Pfenning. On equivalence and canonical forms in the LF type theory. *Transactions on Computational Logic*, 6:61–101, Jan. 2005.

[13] R. Klapper and A. Stump. Validated Proof-Producing Decision Procedures. In C. Tinelli and S. Ranise, editors, *2nd International Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2004.

[14] N. Klarlund and M. Schwartzbach. Graph types. In *Principles of Programming Languages*, pages 196–205. ACM Press, 1993.

[15] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, Edinburgh LFCS, 1992.

[16] C. McBride and J. McKinna. The View from the Left. *Journal of Functional Programming*, 14(1), 2004.

[17] F. Mehta and T. Nipkow. Proving Pointer Programs in Higher-Order Logic. In F. Baader, editor, *19th International Conference on Automated Deduction*, volume 2741 of *LNCS*, pages 121–135. Springer-Verlag, 2003.

[18] A. Møller and M. Schwartzbach. The pointer assertion logic engine. In M. Soffa, editor, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[19] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[20] F. Pfenning. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In J. Halpern, editor, *Proceedings of the Sixteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2001*. IEEE Computer Society Press, 2001.

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

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

[23] B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[24] B. C. Pierce and D. N. Turner. Local type inference. In *25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–265, 1998.

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

[26] J. Reynolds. Separation Logic: a Logic for Shared Mutable Data Structures. In *IEEE Symposium on Logic in Computer Science*, 2002.

[27] S. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.

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

[29] 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*, volume 2758 of *LNCS*, pages 120–135. Springer-Verlag, 2003.

[30] E. Westbrook, A. Stump, and I. Wehrman. A Language-based Approach to Functionally Correct Imperative Programming. Technical Report FIXME, Washington University in Saint Louis, July 2005.

[31] H. Xi. Facilitating Program Verification with Dependent Types. In *Proceedings of the International Conference on Software Engineering and Formal Methods*, pages 72–81, 2003.

[32] D. Zhu and H. Xi. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, pages 83–97, Long Beach, CA, January 2005. Springer-Verlag LNCS vol. 3350.