Refined Typechecking with Stardust

Jana Dunfield

Carnegie Mellon University*
Pittsburgh, PA, USA

Abstract

We present Stardust, an implementation of a type system for a subset of ML with type refinements, intersection types, and union types, enabling programmers to legibly specify certain classes of program invariants that are verified at compile time. This is the first implementation of unrestricted intersection and union types in a mainstream functional programming setting, as well as the first implementation of a system with both datasort and index refinements. The system—with the assistance of external constraint solvers—supports integer, Boolean and dimensional index refinements; we apply both *value refinements* (to check red-black tree invariants) and *invaluable refinements* (to check dimensional consistency). While typechecking with intersection and union types is intrinsically complex, our experience so far suggests that it can be practical in many instances.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms languages, verification

1. Introduction

Compile-time typechecking in statically typed languages such as ML, Haskell, and Java catches many mistakes: "well typed programs cannot 'go wrong'" (Milner 1978). However, many programs have bugs (do not behave as intended) yet do not actually "go wrong" in the operational semantics. Adding two floating point values, where one represents a length and another a mass, is nonsensical yet permitted, as is building a red-black tree in which a red node has a red child. While one can often change data representation to allow verification of such invariants in conventional type systems—by wrapping the floating point value with a "dimension tag", splitting the red-black tree datatype into two variants, adding a "phantom" type argument, and so on—programs become less efficient or, more importantly, less readable. Heavyweight approaches based on theorem proving may avoid those defects, but are not worth the effort for many programming tasks. In this paper, we follow a different approach, in which we leave the programs and their underlying data structures alone, enriching only the type expressions with specifications based on

- *datasort refinements* (also called refinement types) (Davies and Pfenning 2000; Davies 2005) and *index refinements* (so-called limited dependent types) (Xi and Pfenning 1999; Xi 1998) for atomic properties of data structures;
- *intersection types* and *union types* that combine properties by conjunction and disjunction, respectively (Davies and Pfenning 2000; Dunfield and Pfenning 2004).

Datasort and index refinements express properties of algebraic datatypes: in red-black trees, datasort refinements can distinguish empty from non-empty trees, and red nodes from black nodes; integer index refinements encode the *black height* of trees, a natural number. We also apply index refinements to base types: integers are indexed by themselves (a singleton type); floats are indexed by the associated dimensional unit (e.g. meters squared).

Our setting is a subset of core Standard ML with datasort and index refinements, intersection types, union types, and universal and existential index quantification. The type system is closely based on previous work (Dunfield and Pfenning 2003, 2004), though we incorporate a further development, *let-normal typechecking*, described only briefly here, which makes the earlier work's problematic union-elimination rule practical. This paper focuses on the type system from a user's perspective only; on the structure of the typechecker; and on examples with datasort refinements, integer index refinements, and dimension index refinements.

Intersection types have rarely been exposed to users, with the significant exception of datasort refinement systems (Freeman and Pfenning 1991; Davies and Pfenning 2000; Dunfield and Pfenning 2003, 2004). True union types (that is, *untagged* union types) are rarer still; we draw heavily on our previous work as well as our unpublished work (Dunfield 2007, Ch. 5) on let-normal typechecking.

Dimension (units of measure) checking is not a new idea, but its encoding as an index domain is elegant and goes beyond the old line of index refinement researchers that your index refinement typechecker comes with any domain you want, as long as it's integers. It also provides a nice example of what we call *invaluable refinements*, which are not based on values.

The typechecker delegates much of the work of constraint solving in the index domains; the system presently has interfaces to ICS and CVC Lite. As the power and range of such tools grows, the typechecker's power can grow with relatively little effort.

Section 2 describes our subset of Standard ML, called StardustML. Section 3 explains our property type system. Sections 4 and 6 formulate the central index domains: integers and dimensions, presenting example programs in each. Section 7 outlines the design of the typechecking system and Section 8 discusses its performance. Finally, we discuss related work and conclude.

2. The StardustML language

Except at the type level, StardustML is a subset of core (module-free) Standard ML (Milner et al. 1997). A StardustML program

* Current affiliation: McGill University, Montreal, Canada.

PLPV '07 October 5, 2007, Freiburg, Germany

```
   sort ::= id | ( sort ) | sort (* sort)*
          | { id : sort | proposition }
proposition ::= index-exp | proposition and proposition
  index-exp ::= id | index-exp (+ | − | ···) index-exp | index-aexp
 index-aexp ::= # integer-literal ( index-exp ) | integer-literal
          | ( index-exp (, index-exp)* )
       texp ::= id [( index-exp (, index-exp)* )] | ( texp )
          | texp → texp | texp * texp
          | texp & texp | texp ∨ texp
          | { proposition } texp | [ proposition ] texp
          | -all id (, id)* : sort- texp
          | -exists id (, id)* : sort- texp
```

Figure 1. Concrete syntax of index sorts, propositions, index expressions, and types in StardustML

consists of SML datatype declarations followed by a sequence of *blocks*. A block is a sequence of mutually recursive declarations (either **fun** ... **and** ... **and** ..., or just a single **fun** or **val** binding). Each block may be preceded by a type annotation of the form `(*[ ... ]*)`. This annotation form appears as a comment `(*...*)` to Standard ML compilers, allowing programs in the subset language to be compiled normally (but see Section 7.5).

StardustML does not include parametric polymorphism, modules, **ref**, user exception declarations, records, and a number of forms of syntactic sugar such as clausal function definitions. However, it does support all the interesting SML pattern forms. It also supports exceptions, though an exception datatype is pre-defined and cannot be changed by the user program.

We give the grammar for types and related constructs in Figure 1. The notations [...] and $(\dots)^*$ respectively denote zero-or-one and zero-or-more repetitions. Nonterminals are written *nonterm*. Terminals with several lexemes, such as identifiers ***id***, appear in ***bold italic***, while keywords appear in **bold**.

3. The Stardust property type system

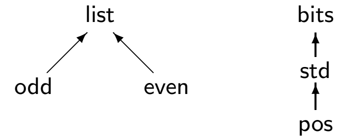
In addition to a subset of SML types, Stardust supports the full set of property types from Dunfield and Pfenning (2004), as well as guarded and asserting types (inspired by Xi (2004)).

3.1 Tridirectional typechecking

Stardust is based on tridirectional typechecking (Dunfield and Pfenning 2004), which is a form of *bidirectional typechecking*. The type of an expression is variously *synthesized* or *checked*. In this particular formulation of bidirectionality, introduction forms such as **fn** and tuples are classified as *checking forms*, while elimination forms such as $e_1(e_2)$ are classified as *synthesizing forms*. Type annotations are needed precisely where a checking form is used in a *synthesizing position*. For example, in the function application $e_1(e_2)$, the function e_1 is in a synthesizing position, while e_2 is in a checking position. Hence, the expression (**fn** $x \Rightarrow x$)y needs an annotation around the **fn** subexpression. In contrast, `map` (**fn** $x \Rightarrow x + 1$) needs no annotation because the **fn** is in a checking position. In practice, annotations seem to be rare except for function declarations, where they are mandatory.

3.2 Atomic refinements

The basic level of properties is provided by *datasort* and *index* refinements. The former, also known as refinement types, are similar to Refinement ML (Davies 2005): an ML-style (that is, algebraic and inductive) datatype is *refined* by a set of datasorts organized into an inclusion hierarchy: for example, odd- and even-length lists, or inductively defined bitstrings (1) with no leading zeroes (std), or (2) with no leading zeroes and also not the empty bitstring (pos).



Our second variety of atomic refinement, the index refinement, is very similar to DML (Xi 1998). It is a markedly limited form of dependent type, over a decidable constraint domain. As expressions have types, indices have *index sorts*. Following DML, the type t with index i is written t(i); so, indexing lists by their length, `[5, 6]` has type list(2). In addition to ML datatypes, primitive types such as integers and reals can be refined. As in DML, we index integers by their values, so 3 has type int(3); the index refinement serves as a singleton type. On the other hand, we index the ML floating-point type real by a *dimension*. Thus, real(M ^ 2) is the type of areas expressed in square meters.

Datasort and index refinements can be combined, as in the red-black tree examples below.

3.3 Combining properties

Stardust provides several means of combining properties expressed through atomic refinements. Intersection types & express conjunction: v : A & B says that v : A and v : B. Likewise, union types $\vee$ express disjunction: v : C $\vee$ D says that v : C or v : D (or possibly both). The types **-all** ***id*** (, ***id***)* : *sort- texp* and **-exists** ***id*** (, ***id***)* : *sort- texp* quantify over indices. Several of these are combined in the following 'increment' function on bitstrings. Its type annotation says that `inc` is a function that, for all natural numbers `len` and `value`, takes bitstrings in standard form (no leading zeroes) of length `len` and value `value` and returns a positive (in standard form, and nonzero) bitstring of the same length and value `value`+1, the input value, or else ($\vee$) a positive bitstring of length `len` + 1 and value `value` + 1. Moreover, it also has (&) a similar property when given a bitstring in a possibly-nonstandard form (bits).

```
(*[ val inc :
     -all len, value : nat-
        std(len, value) → (pos(len, value+1)
                         ∨ pos(len+1, value+1))
   & bits(len, value)→ (bits(len, value+1)
                         ∨ bits(len+1, value+1))   ]*)
fun inc n = case n of
     E ⇒ One E
   | Zero n ⇒ One n
   | One n ⇒ Zero (inc n)
```

In defining the type of the built-in function `*`, we use *asserting* and *guarded* types. The asserting type [P] A is like A but affirms that the index-level proposition P holds. The *guarded type* {P} A is equivalent to A provided that P holds, and useless otherwise (as a "top" type would be). Thus, we can express that *if* both arguments to `*` are nonnegative, the result is as well:

```
primitive val * :
    (-all a,b:int- int(a) * int(b) → int(a * b))
  & (-all a,b:int- {a >= 0 and b >= 0} int(a) * int(b)
              → -exists c : int- [c >= 0] int(c))
  & (-all d1,d2:dim- real(d1) * real(d2) → real(d1*d2))
```

When the type is index-refined but the index expression is omitted, as when list, int or real appears alone, the type is interpreted in one of two ways. For most types, such as list and int, an existential quantifier is added: int becomes **-exists** a : int- int(a). However, this does not work well for real: quantities of type **-exists** d : dim- real(d) are quite useless since they cannot be added, subtracted, or converted to a quantity of known dimension. Therefore, for real we define a *default index*, the special "no dimension" index `NODIM`. Data in existing code is thus interpreted as

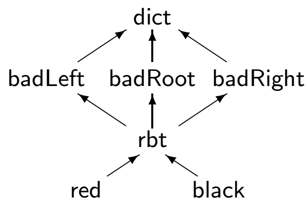


Figure 2. Subsort relation used in Figure 3

dimensionless. In our investigations so far, real is the only type for which a default index has seemed appropriate.

Precedence is as follows, from lowest to highest:

all `{···}` & $\rightarrow$ **exists** `[···]` `*` $\vee$

Thus, -**all** a : int- $A \rightarrow \mathsf{list}(a)$ & $B \rightarrow \mathsf{list}(a+1)$ is equivalent to -**all** a : int- $((A \rightarrow \mathsf{list}(a))$ & $(B \rightarrow \mathsf{list}(a+1)))$.

A program begins with datatype declarations, each of which can begin with a bracketed refinement declaration comprised of **datacon** declarations and an index sort specification, e.g. **datatype** `dict` **with** `nat`. Datasorts are specified by a "kernel" of the subsort relation—a set of pairs—of which the system takes the reflexive-transitive closure. For an example of both of these, discussed in Section 4.2, see Figures 2 and 3.

4. The integer index domain

The type system underlying Stardust is, like that of DML (Xi 1998), parametric in the index domain. Stardust supports two major index domains: integers and dimensions. The integer domain, described in this section, was implemented in DML; applying index refinements to dimension types, discussed in Section 6, is novel.

We write int for the index sort of integers. The constants are $\ldots, \tilde{}1, 0, 1, \ldots$. The functions are +, -, and *. The predicates are <, <=, =, <>, >=, >. Nonlinear expressions such as `a * b` are not allowed, making the resulting constraints decidable.

The integer sort refines the base type `int`. This is a mismatch, since `int` is a *finite* type of integers. However, SML integer arithmetic operations must raise an exception on overflow, so the results of such operations will have the values their integer indices claim they do; if $x = 5$ then $x + 1 = 6$ as claimed, while if $x =$ `maxInt` then $x + 1$ raises an exception—no harm done.

4.1 Natural numbers

The natural numbers are defined by a *subset sort*:

```
indexsort nat = {a:int | a >= 0}
```

Stardust replaces subset sorts by guarded and asserting types: -**all** a : nat- $\mathsf{int}(a) \rightarrow$ -**exists** b : nat- $\mathsf{int}(b)$ becomes -**all** a : int- $\{a \geq 0\}$ $\mathsf{int}(a) \rightarrow$ -**exists** b : int- $[b \geq 0]$ $\mathsf{int}(b)$. Other subset sorts, such as strictly positive numbers, can be defined similarly.

4.2 Example: Red-black tree insertion

Red-black trees are a good example of datasort and index refinements in combination. In this example, we fix the key type to be `int`, and we have no associated record component, so that a `dict` represents a set of integers rather than a map from integers to some other type. Since our refinements will be concerned with the *structure* of the trees rather than the integers contained—we will not try to guarantee an order invariant, for example[1]—having a set rather than a map does not detract from the example.

[1] In theory, one could guarantee order by indexing with the minimum and maximum keys; in practice, this requires splitting disjunctions, which we have not implemented. For a discussion, see Dunfield (2007, Ch. 6).

```
(* Based on an example of Rowan Davies and Frank Pfenning *)
(*[ datatype dict with nat
  datasort dict :
    badLeft < dict; badRoot < dict; badRight < dict;
    rbt < badLeft; rbt < badRoot; rbt < badRight;
             red < rbt;  black < rbt

  datacon  Empty : black(0)
  datacon  Black : -all h : nat-
       int * dict(h) * dict(h) → dict(h+1)
    &  int * rbt(h) * rbt(h) → black(h+1)
    &  int * badRoot(h) * rbt(h) → badLeft(h+1)
    &  int * rbt(h) * badRoot(h) → badRight(h+1)
  datacon  Red : -all h : nat-
       int * dict(h) * dict(h) → dict(h)
    &  int * black(h) * black(h) → red(h)
    &  int * rbt(h) * black(h) → badRoot(h)
    &  int * black(h) * rbt(h) → badRoot(h)     ]*)

datatype dict = Empty
              | Black of  int * dict * dict
              | Red of  int * dict * dict  ;

(* restore_right (Black(e,l,r)) ⟹ dict
     where (1) Black(e,l,r) is ordered,
           (2) Black(e,l,r) has black height h,
           (3) color invariant may be violated at the root of r:
                one of its children might be red.
     and dict is a re—balanced red/black tree (satisfying all invariants)
     and same black height h. *)
(*[ val restore_right :
         -all h : nat- badRight(h) → rbt(h)  ]*)
fun restore_right arg = case arg of
   Black(e, Red lt, Red (rt as (_,Red _,_))) ⇒
     Red(e, Black lt, Black rt)  (* re—color *)
 | Black(e, Red lt, Red (rt as (_,_,Red _))) ⇒
     Red(e, Black lt, Black rt)  (* re—color *)
 | Black(e, l, Red(re, Red(rle,rll,rlr), rr)) ⇒
     Black(rle, Red(e, l, rll), Red(re,rlr,rr))
 | Black(e, l, Red(re, rl, rr as Red _)) ⇒
     Black(re, Red(e, l, rl), rr)
 | dict ⇒ dict
```

Figure 3. redblack-full.rml

```
datatype dict = Empty
              | Black of  int * dict * dict
              | Red of  int * dict * dict
```

Red-black trees must satisfy three invariants: (1) For every non-empty node containing a key k, every key in its left child is less than k and every key in its right child is greater than k; (2) The children of a red node are black (color invariant); (3) Every path from the root to a leaf has the same number of black nodes, called the black height of the tree. Any tree satisfying these invariants is balanced: the height of a tree containing n non-`Empty` nodes is at most $2\log_2(n+1)$ (Cormen et al. 1990, p. 264). Invariant 2 is concerned with color, and the colors of a datasort form a small finite set, so it is a suitable candidate for a datasort refinement. Invariant 3 involves node color, but also black height, which is a natural number and therefore suitable for index refinement.

We begin with a datasort rbt of "proper" **r**ed-**b**lack **t**rees, which satisfy invariants 2 and 3. red and black are subsorts of rbt, representing proper red-black trees with a root node of the specified color.

But it is not quite enough to distinguish trees that satisfy all the invariants (rbt) from those that might not satisfy the color invariant (dict); if we know something is a dict we know nothing about *where* the color violation occurs. So we add datasorts badRoot, badLeft and badRight for possible color violations at the root (the root is red and some child is red), at the left child (the left child is red and one of *its* children is red), and at the right child. The "good" datasorts rbt, red, black are subsorts of the "bad" datasorts badRoot, etc.: the "bad" datasorts represent not that the color invariant is violated, but that it *may* be violated. See Figure 2.

To save space, we exclude the `restore_left` function (which is symmetric to `restore_right`) and the `insert` function; the full example is available at `http://type-refinements.info/stardust/plpv/redblack-full.rml`.

4.2.1 Related work

Our combination of refinements for red-black tree insertion is new, but the application of datasort and index refinements individually is not. In fact, Figure 3 is based on code from Davies' thesis (Davies 2005, pp. 277–279). Moreover, the black height refinement is not new either (Xi 1998, pp. 161–165). Xi also guarantees the color invariant, but by refining the tree datatype by the index product $\mathsf{int} * \mathsf{int} * \mathsf{int}$, representing the color, black height, and "red height" respectively. 0 in the first component means red and 1 means black, with an existential quantifier used if the color is not known. The so-called "red height" is not analogous to the black height, but instead counts the *consecutive* red nodes, and is 0 if there are none, i.e. if the color invariant is satisfied. The resulting types are awkward and substantially less legible than ours.

Other type-level programming techniques have been applied to check red-black tree invariants, relying on phantom and existential types (Kahrs 2001). In our opinion, such approaches are even more awkward than Xi's color-encoding, and require major changes to the basic tree datatype and code.

4.3 Example: Red-black tree deletion

As just discussed, others have studied refinements for red-black tree insertion. Deletion is another matter. Though more complicated than insertion, deletion in an imperative style can be cooked up easily from standard sources; purely functional deletion cannot (unlike insertion, it is not treated by Okasaki (1998)). However, there are implementations, such as the *RedBlackMapFn* structure in the Standard ML of New Jersey library. We easily translated *RedBlackMapFn* into StardustML. Finding appropriate refinements and invariants was more difficult, and not made easier by several lacunae in the original code, including two bugs leading to a violated color invariant in the tree returned by the `delete` function. However, we eventually succeeded, resulting in an implementation that satisfies invariants (1)–(3) (see the previous section).

The key data structure is the *zipper*, which represents a tree with a hole, *backwards*, allowing easy traversal toward the tree's root. This corresponds to the series of rotations and color changes that may be required after deleting a node. The function `zip` takes a zipper and a tree and plugs the tree into the zipper's hole. The `TOP` constructor (Figure 5, bottom) represents a zipper consisting of just a hole; the `LEFTB` and `LEFTR` constructors represent the edge from a left child to a `Black` or `Red` node, respectively. The `RIGHTB` and `RIGHTR` constructors are symmetric. The examples in Figure 4 should clarify the constructors' meaning.

We refine zippers by a datasort and two integer indices.

4.3.1 Datasort refinement of zipper

The datasort encodes two properties: the color of the hole's parent and the color of the root of the result of $\mathtt{zip}(z, t)$. If a zipper has datasort blackZipper, the root (of the zipper, that is, the parent of the hole) is black and can therefore tolerate black trees as well as red; thus, all values $\mathtt{LEFTB}(\dots)$ and $\mathtt{RIGHTB}(\dots)$ have datasort blackZipper. If a zipper z has datasort BRzipper (for "Black Root zipper"), the resulting zipped tree $\mathtt{zip}(z, t)$ will have a black root and thus have datasort black; all zippers having the form

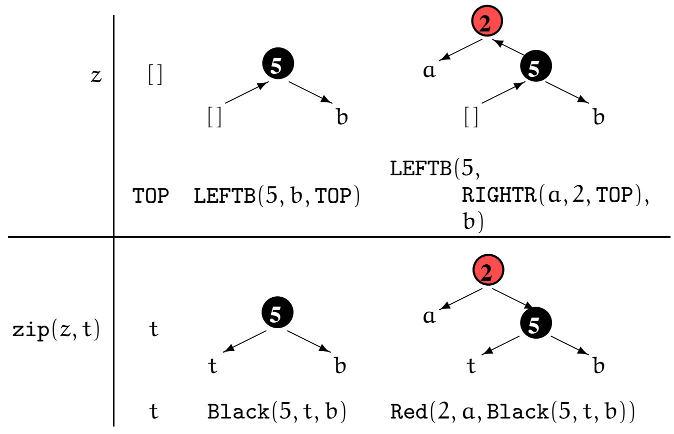
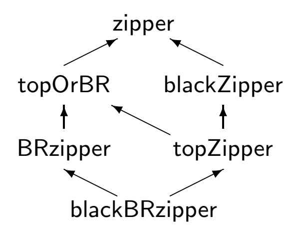


Figure 4. Examples of zippers

$\dots\mathtt{LEFTB}(\_, \_, \mathtt{TOP})\dots$ or the form $\dots\mathtt{RIGHTB}(\_, \_, \mathtt{TOP})\dots$) have datasort BRzipper.

We also have a datasort topZipper, such that $\mathtt{TOP}$: topZipper. Zipping `TOP` with a tree t yields t itself; therefore, zipping `TOP` with a black tree yields a black tree. This property of a zipper yielding a black tree when it is zipped with a black tree is captured by topZipper.

The remaining datasorts, topOrBR and blackBRzipper, can be thought of as the union and intersection, respectively, of BRzipper and topZipper:



4.3.2 Index refinement of zipper

We refine the zipper datatype by a pair of natural numbers. A zipper z has index refinement (h, hz) precisely if `zip`, when given z and a tree t of black height h (that is, t : $\mathsf{rbt}(h)$), yields a tree of black height hz. Hence, `TOP` has type $\mathsf{zipper}(h, h)$ for all h, because $\mathtt{zip}(\mathtt{TOP}, t)$ yields just t.

4.3.3 Overview of the algorithm

Our code is closely based on the SML/NJ library, which claims to implement, in a functional style, the imperative pseudocode of Cormen et al. (1990). At a very high level, without regard for the red-black invariants, deletion of key k in the tree t goes as follows:

1. Starting from the root of t, find a node with key k.
2. Join the left/right children a/b of the node containing key k:
   i. Find the minimum x of b; this x is greater than all keys in a, and less than all other keys in b.
   ii. Delete the node containing x.
   iii. Replace the node containing k with one containing x, with the same a and the new b (with x deleted) as its children.

We can distinguish cases based on the color of the node containing x, the minimum key in the subtree rooted at b.[2] First note that since x is the minimum, its node must have an empty left child.

[2] These cases do *not* correspond to the `joinRed` and `joinBlack` functions, whose names refer to the color of the node containing k.

```
(*[
 datatype dict with nat
 datasort dict :
       badLeft < dict; badRoot < dict; badRight < dict;
   rbt < badLeft;   rbt < badRoot;    rbt < badRight;
             nonempty < rbt;   black < rbt;
   red < nonempty;
   nonemptyBlack < nonempty; nonemptyBlack < black

 datacon  Empty : black(0)

 datacon  Black : -all h : nat-
        int * dict(h) * dict(h) → dict(h+1)
     &  int * rbt(h) * rbt(h) → nonemptyBlack(h+1)
     &  int * badRoot(h) * rbt(h) → badLeft(h+1)
     &  int * rbt(h) * badRoot(h) → badRight(h+1)

 datacon  Red : -all h : nat-
        int * dict(h) * dict(h) → dict(h)
     &  int * black(h) * black(h) → red(h)
     &  int * rbt(h) * black(h) → badRoot(h)
     &  int * black(h) * rbt(h) → badRoot(h)
]*)
datatype dict = Empty
             | Black of  int * dict * dict
             | Red of  int * dict * dict
(*[
 (* z : zipper(h, hz) if zip(z, t) : rbt(hz), where t : rbt(h). *)
 datatype zipper with nat * nat

 datasort zipper :
   topOrBR < zipper;  blackZipper < zipper;
   BRzipper < topOrBR;
   topZipper < topOrBR;  topZipper < blackZipper;
   blackBRzipper < topZipper; blackBRzipper < BRzipper

 datacon  TOP : -all h : nat- topZipper(h, h)

 datacon  LEFTB : -all h, hz : nat-
      int*rbt(h)*zipper(h+1,hz) → blackZipper(h,hz)
   &  int*rbt(h)*topOrBR(h+1,hz) → blackBRzipper(h,hz)

 datacon  RIGHTB : -all h, hz : nat-
      rbt(h)*int*zipper(h+1,hz) → blackZipper(h,hz)
   &  rbt(h)*int*topOrBR(h+1,hz) → blackBRzipper(h,hz)

 datacon  LEFTR : -all h, hz : nat-
      int*black(h)*blackZipper(h,hz) → zipper(h,hz)
   &  int*black(h)*blackBRzipper(h,hz) → BRzipper(h,hz)

 datacon  RIGHTR : -all h, hz : nat-
      black(h)*int*blackZipper(h,hz) → zipper(h,hz)
   &  black(h)*int*blackBRzipper(h,hz) → BRzipper(h,hz)
]*)
datatype zipper = TOP
               | LEFTB of  int * dict * zipper
               | LEFTR of  int * dict * zipper
               | RIGHTB of  dict * int * zipper
               | RIGHTR of  dict * int * zipper ;
```

Figure 5. rbdelete.rml, part 1: datatypes

- If the node containing x is red, its right child cannot be `Red` (color invariant), nor can it be `Black` (its left child is empty—black height 0—so its right child must also have black height 0, but any `Black`-rooted tree has black height at least 1), so the right child must also be empty.

  Deleting a red node does not change any black heights, so the black height invariant is preserved; the only change needed is to substitute x for k.

- If the node containing x is black, its right child cannot be `Black` (by similar reasoning to the red case). However, its right child could be $\mathtt{Red}(y, \mathtt{Empty}, \mathtt{Empty})$, in which case we can just replace x with y—keeping the node containing x black—which preserves black height. The hard case is when both children of the node containing x are empty: merely deleting that node means that its parent will have a left subtree of black height 0 and a right subtree of black height 1, which is inconsistent. This is called a *black deficit*. We can try to fix it by calling `bbZip`, which moves up the tree towards the root, performing rotations and color changes. While this process will always yield a valid subtree that satisfies the black height invariant (and of course the color invariant), it may not actually "fix the deficit": the resulting subtree may still have a black height that is one less than before. If that occurs—signalled by `bbZip` returning $(\mathtt{true}, t)$—we call `bbZip` again, continuing the rotations and color changes upward past the node that used to contain k (and now contains x). Otherwise, all the invariants have been fixed, and we need only replace k with x and call `zip`.

4.3.4 The `zip` function

The index refinement (at the start of Figure 6) is plain: a zipper that yields a tree of black height hz when zipped with a tree of black height h, when zipped with such a tree, yields a tree of black height hz. After all, we refined the `zipper` datatype with the behavior of `zip` in mind.

The datasorts are less obvious. The first part of the intersection expresses the fact that if the parent of the zipper's hole is black (blackZipper) then replacing the hole with any valid tree (rbt) yields a valid tree. The second part says that if the parent of the hole is *not* known to be black, then only a black tree can be substituted, because the parent might be red and we cannot allow a color violation. The third part of the intersection says that, when a blackBRzipper—a zipper with a black node as the parent of the hole and that, when zipped, yields a black-rooted tree—is zipped with any tree, a black-rooted tree results. The fourth says that when either a topZipper (such as `TOP`) or a BRzipper (such as $\mathtt{RIGHTB}(a, 2, \mathtt{TOP})$) is zipped with a black tree t, a black tree results—if the zipper is `TOP`, because the result consists of just t, which is black; if the zipper is BRzipper, because the result has a black root regardless of the color of t.

4.3.5 The `bbZip` function

`bbZip` is a recursive, zipper-based version of the pseudocode "RB-DELETE-FIXUP" (Cormen et al. 1990, p. 274); the comments show how the various case arms correspond to sections of pseudocode. We therefore focus on the type annotation (Figure 6). Each part of the intersection shares index refinements; we will look at the first, which has the simplest datasorts. Given a zipper that when zipped with a tree of black height $h+1$ yields a tree of black height hz, and a tree of black height h (one less than $h+1$, i.e. , with a "black deficit"), `bbZip` returns either

- $(\mathtt{true}, t)$ where $t : \mathsf{rbt}(hz - 1)$ (that is, a *valid* tree—with no internal black height mismatches—but with a black height one less than before), or
- $(\mathtt{false}, t)$ where $t : \mathsf{rbt}(hz)$, a valid tree with the same black height hz as the original tree.

The second part of the intersection says that given a zipper that, when zipped with any tree, yields a tree with a black root, the resulting tree (whether of black height $hz - 1$ or hz) will have a black root. The third part of the intersection says that given a zipper that is either `TOP` or a BRzipper, and a black-rooted tree, the resulting tree must be black. This information is needed when we typecheck `delMin`.

4.3.6 The `delMin` function

$\mathtt{delMin}(t, z)$ returns the minimum key (an integer) in t; it also returns t with the minimum removed. It calls `bbZip` to fix internal

```
(*[ val zip : -all h, hz : nat-
         blackZipper(h, hz) * rbt(h) → rbt(hz)
       & zipper(h, hz) * black(h) → rbt(hz)
       & blackBRzipper(h, hz) * rbt(h) → black(hz)
       & topOrBR(h, hz) * black(h) → black(hz)   ]*)
fun zip arg = case arg of
   (TOP, t) ⇒ t
 | (LEFTB (x, b, z as _), a) ⇒ zip(z, Black(x, a, b))
 | (RIGHTB(a, x, z as _), b) ⇒ zip(z, Black(x, a, b))
 | (LEFTR (x, b, z), a)      ⇒ zip(z, Red(x, a, b))
 | (RIGHTR(a, x, z), b)      ⇒ zip(z, Red(x, a, b))

(* bbZip propagates a black deficit up the tree until either the top
 * is reached, or the deficit can be covered. It returns a boolean
 * that is true if there is still a deficit and the zipped tree. *)

(*[ val bbZip : -all h,hz : nat-
     zipper(h+1,hz)*rbt(h) → ((bool(true)*rbt(hz-1))
                             ∨ (bool(false)*rbt(hz)))
 & BRzipper(h+1,hz)*rbt(h) → ((bool(true)*black(hz-1))
                             ∨ (bool(false)*black(hz)))
 & topOrBR(h+1,hz)*black(h) → ((bool(true)*black(hz-1))
                              ∨ (bool(false)*black(hz)))
]*)
fun bbZip arg = case arg of
  (TOP, t) ⇒ (true, t)
| (LEFTB(x, Red(y,c,d), z), a) ⇒   (*1L—Black *)
    bbZip(LEFTR(x, c, LEFTB(y, d, z)), a)
| (LEFTB(x, Black(w,Red(y,c,d),e), z), a) ⇒
     (*3L—Black *)
   (false, zip(z, Black(y,Black(x,a,c), Black(w,d,e))))
| (LEFTR(x, Black(w,Red(y,c,d),e),z), a) ⇒   (*3L—Red*)
   (false, zip(z, Red(y, Black(x,a,c), Black(w,d,e))))
| (LEFTB(x, Black(y,c,Red(w,d,e)),z), a) ⇒
     (*4L—Black *)
   (false, zip(z, Black(y,Black(x,a,c), Black(w,d,e))))
| (LEFTR(x, Black(y,c,Red(w,d,e)),z), a) ⇒   (*4L—Red*)
   (false, zip(z, Red(y, Black(x,a,c), Black(w,d,e))))
| (LEFTR(x, Black(y,c,d),z), a) ⇒   (*2L—Red*)
   (false, zip(z, Black(x, a, Red(y,c,d))))
| (LEFTB(x, Black(y,c,d),z), a) ⇒   (*2L—Black *)
   bbZip(z, Black(x, a, Red(y,c,d)))
| (RIGHTB(Red(y,c,d),x,z), b) ⇒   (*1R—Black *)
   bbZip(RIGHTR(d, x, RIGHTB(c,y,z)), b)
| (RIGHTB(Black(y,Red(w,c,d),e),x,z), b) ⇒
     (*3R—Black *)
   (false, zip(z, Black(y,Black(w,c,d), Black(x,e,b))))
| (RIGHTR(Black(y,Red(w,c,d),e),x,z), b) ⇒   (*3R—Red*)
   (false, zip(z, Red(y, Black(w,c,d), Black(x,e,b))))
(* This 4R is correct —— unlike the buggy NJ library   *)
| (RIGHTB(Black(y,c,Red(w,d,e)),x,z), b) ⇒
     (*4R—Black *)
   (false, zip(z, Black(w,Black(y,c,d), Black(x,e,b))))
| (RIGHTR(Black(y,c,Red(w,d,e)),x,z), b) ⇒   (*4R—Red*)
   (false, zip(z, Red(w, Black(y,c,d), Black(x,e,b))))
| (RIGHTR(Black(y,c,d),x,z), b) ⇒   (*2R—Red*)
   (false, zip(z, Black(x, Red(y,c,d), b)))
| (RIGHTB(Black(y,c,d),x,z), b) ⇒   (*2R—Black *)
   bbZip(z, Black(x, Red(y,c,d), b))
```

Figure 6. rbdelete.rml, part 2: `zip`, `bbZip`

black height mismatches, but like `bbZip` it may be unable to maintain the black height of the entire tree, so like `bbZip` it returns a Boolean indicating whether there is still a black deficit. The datasorts are needed in `joinRed`, to make sure that the new children of the red node are black.

4.3.7 The functions `joinRed` and `joinBlack`

If one subtree (a or b) is empty, we simply zip up the tree (`bbZip`) with the other subtree; we can drop the first part of `bbZip`'s result—the flag indicating whether the black height has changed—because the zipper z goes all the way to the original root passed to `delete`, which has no siblings.

Otherwise, we call `delMin`, which returns a tree that may or may not have a deficit. If the returned flag is false, there is no deficit and we can zip up to the root. If the flag is true, there is a deficit, and we call `bbZip`—again, throwing away the resulting flag.

We hand-inlined `joinRed`'s call to `delMin` to make the color invariant work (if there is some refinement of `delMin` that does the job, it was not obvious to us). We removed several impossible "inlined" case arms, so this only slightly lengthened `joinRed`.

4.3.8 The `delete` function

`delete` and its helper `del` simply search for the key to delete, building a zipper, and call `joinRed` or `joinBlack`.

4.3.9 Library bugs

We found two clear bugs in the SML/NJ library; triggering either results in a tree with a red child of a red parent: that is, the color invariant is broken. These trees are still ordered, so searches succeed or fail as usual, and the failure of the color invariant does not seem to cause subsequent operations to produce disordered trees. Hence, the only calamity caused is that operations will take longer than they should. Since *RedBlackMapFn* makes the exported tree type opaque, client code cannot possibly detect the broken invariant. Thus, these bugs will not be found unless insertion and deletion are time-critical and someone is so stubborn as to actually investigate whether the operations are logarithmic. Moreover, runtime testing is not very helpful: traversing a tree to verify the invariant is linear time, so adding the tests to every operation makes those operations linear instead of logarithmic, defeating the purpose of a balanced tree. (We might dream up more clever tests that would add only constant overhead, but then we have to verify our cleverness.)

The first bug is in SML/NJ's "4R" case in `bbZip`; upon inspection, something is obviously wrong because it is not symmetric to the "4L" case. We found this bug some time before we settled on the present refinement of `zipper`: we had only a datasort refinement on `zipper`, but even that, combined with reading each case closely, sufficed to lead us to this bug.

The second bug is in `joinRed`; if `delMin` returns with its first argument true, meaning that the result has a black deficit, the original code calls `bbZip` to fix the deficit; however, the tree passed to `bbZip` includes a red node with `b'` as a child, but `b'` may be red, leading to a color violation (which is *not* somehow fixed inside `bbZip`). We found this second bug much later than the first: we had settled on the index refinement of `zipper` and a nearly-final version of the datasort refinement. Once we became suspicious that `b'` might not always be black, we looked for an input to `delete` that would trigger the bug; we found one, confirming that there was a bug and not simply a case of our refinements being too weak.

5. Booleans

If we consider index predicates such as $>$ to be index functions, then a Boolean sort manifests itself immediately, as the range of such functions. The Boolean sort can also index the bool datatype. Such an indexing scheme is handy for specifying the result of certain functions. For example, we define the type of the ML function < to be **-all** a, b : int- $\mathsf{int}(a) * \mathsf{int}(b) \to \mathsf{bool}(a < b)$. As implemented, the Boolean sort has none of the usual Boolean operations such as conjunction (though that is already part of the constraint language).

```
(*[ val delMin : -all h, hz : nat-
      nonempty(h) * blackZipper(h, hz)
   → int * ((bool(false)*rbt(hz))
          ∨ (bool(true)*rbt(hz-1)))
 &    nonemptyBlack(h) * zipper(h, hz)
   → int * ((bool(false)*rbt(hz))
          ∨ (bool(true)*rbt(hz-1)))
 &    nonempty(h) * blackBRzipper(h, hz)
   → int * ((bool(false)*black(hz))
          ∨ (bool(true)*black(hz-1)))
 &    nonemptyBlack(h) * BRzipper(h, hz)
   → int * ((bool(false)*black(hz))
          ∨ (bool(true)*black(hz-1)))    ]*)
fun delMin arg = case arg of
  (Red(y, Empty, b), z) ⇒
    (y, (false
         (* i.e., no deficit, black height unchanged *) , zip(z,b)))
|  (Black(y, Empty, b), z) ⇒
    (* This is the minimum; deleting it yields a black deficit. *)
    (y, bbZip(z,b))
|  (Black(y, a, b), z) ⇒ delMin(a, LEFTB(y, b, z))
|  (Red(y, a, b), z) ⇒ delMin(a, LEFTR(y, b, z))
(*[ val joinRed : -all h,hz:nat-
       black(h)*black(h)*blackZipper(h, hz) → rbt  ]*)
fun joinRed arg = case arg of
  (Empty, Empty, z) ⇒ zip(z, Empty)
| (a, Empty, z) ⇒ #2(bbZip(z, a))
| (Empty, b, z) ⇒ #2(bbZip(z, b))
| (a, Black(x,Empty,bb), z)⇒#2(bbZip(RIGHTR(a,x,z),bb))
| (a, Black(y,aa,bb), z) ⇒
    let in case delMin(aa, LEFTB(y, bb, TOP)) of
        (x, (needB as false, b')) ⇒ zip(z, Red(x,a,b'))
      | (x, (needB as true, b')) ⇒
          #2(bbZip(RIGHTR(a,x,z), b'))
    end
(*[ val joinBlack : -all h,hz:nat-
       rbt(h)*rbt(h)*zipper(h+1, hz) → rbt  ]*)
fun joinBlack arg = case arg of
  (a, Empty, z) ⇒ #2(bbZip(z, a))
| (Empty, b, z) ⇒ #2(bbZip(z, b))
| (a, b, z) ⇒
    let in case delMin(b, TOP) of
      (x, (needB as false, b')) ⇒ zip(z, Black(x,a,b'))
    | (x, (needB as true, b')) ⇒
          #2(bbZip(RIGHTB(a,x,z), b'))
    end
(*[ val delete : -all h : nat- rbt(h) → int → rbt ]*)
fun delete t key =
  let
    (*[ val del : -all h, hz : nat-
                   rbt(h) * blackZipper(h, hz) → rbt
                 & black(h) * zipper(h, hz) → rbt  ]*)
    fun del arg = case arg of
        (Empty, z) ⇒ raise NotFound
      | (Black(key1, a, b), z) ⇒
          if key = key1 then joinBlack (a, b, z)
          else if key < key1 then del (a, LEFTB(key1, b, z))
          else del (b, RIGHTB(a, key1, z))
      | (Red(key1, a, b), z) ⇒
          if key = key1 then joinRed (a, b, z)
          else if key < key1 then del (a, LEFTR(key1, b, z))
          else del (b, RIGHTR(a, key1, z))
  in
    del(t, TOP)
  end
```

Figure 7. rbdelete.rml, part 3: `delMin`, `joinRed`, `joinBlack`, `delete`

6. Dimensions: an invaluable refinement

Dimensions are ubiquitous in physics and related disciplines. For example, the plausibility of engineering calculations can be checked by seeing whether the dimension of the result is the expected one. If one concludes that the work done by a physical process is $x \cdot (a_1 + a_2)$ where x is a distance and a_1, a_2 are masses, something is wrong. If, on the other hand, the conclusion has the form $x \cdot (n_1 + n_2)$ where n_1 and n_2 are forces, it is at least possible that the calculation is correct, work being a product of distance and force. Basic operations like addition are subject to sanity checking through dimensional analysis: one cannot add a distance to a force, and so forth. (*Dimension* refers to a quantity such as distance, mass or time; systems of *units* define base quantities for dimensions. For example, in civilized countries, the base unit of distance is the meter.)

The idea of trying to catch dimension errors in programs is old. Kennedy (1996) cites sources as early as 1978. Many dimension checking schemes were hamstrung by their lack of polymorphism: they could not universally quantify over dimension variables. For example, they could not express a suitably generic type for the square function **fn** $x \Rightarrow x * x$. Kennedy's system, extending Standard ML, is an elegant formulation providing dimension polymorphism and user-definable dimensions. However, it is a substantial extension of the underlying type system, and is complicated by doing full inference rather than bidirectional checking. For us, dimensions are, formally, just another index domain; practically, the implementation work involved was modest (less than one person-week).

We refine the primitive type real of floating point numbers with a dimension. Certain quantities, including nonzero floating-point literals, are dimensionless and are indexed by `NODIM`; however, the zero literal 0.0 has type **-all** a : dim- real(a). Constants M, S, and so forth have type real(M), real(S), etc. All these constants have the value 1.0, so $3.0 * M$ has value 3.0.

In fact, the value produced by $3.0 * M$ is equal to the values produced by 3.0, and to that produced by $3.0{*}S$, by $3.0{*}M{*}M$, and so on. Unlike the data structure refinements of Section 4, dimension refinements say virtually nothing about values! Zero is an exception to this: it appears that if $\vdash v$: **-all** a : dim- real(a) then $v = 0.0$. However, for any $\vdash v : \mathsf{real}(d)$ the set of possible values is exactly the same for every d, as well as being the same set as the simple type real. After all, there should be no tag at runtime.

But what, then, do we actually learn when a program with dimension refinements passes the typechecker? With red-black tree refinements, one could prove that any value of type red must have the form `Red(...)`, but with dimensions there are few directly corresponding properties. Instead, being well typed means that subterms of dimension type are used in a consistent way. The user must make some initial claims about dimensions (otherwise everything will be dimensionless and nothing is gained), which cannot be checked, though we can check the consistency of their consequences. For example, the user must be free to multiply by constants such as M, to assign dimensions to literals and to the results of functions like *Real.fromString*. Given free access to those constants, for any *known constant* dimensions d_1 and d_2, it is trivial to write the appropriate 'coercion', such as this one for converting M^2 to KG:

```
(*[ val m2_to_kg : real(M^2) → real(KG) ]*)
fun m2_to_kg x = (x / (M * M)) * KG
```

However, there is no 'universal cast' between arbitrary dimensions.

6.1 Definition of the index domain

The dimension sort dim has no predicates besides equality. `NODIM` stands for the multiplicative identity that indexes dimensionless quantities. The constants are M, S, KG, and any additional constants the user declares. The functions are multiplication $*$, which takes two dimensions (e.g. $\mathsf{M} * \mathsf{S}$), and '^', which takes a dimension and an integer (e.g. $\mathsf{M} \ \hat{}\ 3$). (One could also allow rational exponents; see Kennedy (1996, p. 7) for a full discussion.)

6.2 Related work on dimension types in ML

We point out certain differences between Kennedy's work on dimension types in ML and ours. Kennedy (1996, p. 66) notes that the function `power` : $\mathsf{int} \to \mathsf{real} \to \mathsf{real}$, such that `power n x` yields x^n, cannot be typed in his system because it lacks dependently typed integers. With our integer index refinements, this is easy:

```
(*[ val power : -all a:int- -all d:dim-
                  int(a) → real(d) → real(d^a) ]*)
  fun power n x =
    if n = 0 then 1.0
    else if n < 0 then 1.0 / power (~n) x
    else x * power (n-1) x
```

Similarly, in Kennedy's system, universal quantifiers over dimension variables must be prenex (on the outside), just like universal quantifiers over SML type variables. Kennedy (1996, pp. 66-67) gives the example of a higher-order function `polyadd` that applies `prod` to arguments of different dimensions (first to `NODIM` and KG, then to KG and `NODIM`); Kennedy's system cannot infer the type `polyadd` : $(\textbf{-all}\ d_1 : \mathsf{dim}\text{-}\ \textbf{-all}\ d_2 : \mathsf{dim}\text{-}\ \mathsf{real}(d_1) \to \mathsf{real}(d_2) \to \mathsf{real}(d_1 * d_2)) \to \mathsf{real}(\mathsf{KG})$ because the quantifiers are inside the arrow.

```
  fun polyadd prod = prod 2.0 KG + prod KG 3.0
```

Since we do not require universal index quantifiers to be prenex, we can typecheck `polyadd`.

Another small example from Kennedy (1996, p. 11) implements the Newton-Raphson method.

```
(*[ val newton : -all d1,d2:dim-
(* f, a function *)            (real(d1)→real(d2))
(* f', its derivative *)     * (real(d1)→real((d1^~1)*d2))
(* x, the initial guess *)   * real(d1)
(* xacc, relative accuracy *) * real → real(d1)     ]*)
  fun newton (f, f', x, xacc) =
    let val dx = f x / f' x
        val x' = x - dx
    in if abs dx / x' < xacc then x'
       else newton (f, f', x', xacc)
    end
```

Kennedy also presents results about dimension polymorphism in the vein of parametricity (Reynolds 1983). We do not know if similar results hold for our system.

6.3 Units of the same dimension

Some of the most catastrophic dimension bugs are not strictly attributable to confusion of dimensions, but to confusion of *units*. In 1984, the space shuttle Discovery erroneously flew upside down because a system was given input in feet, when it expected input in nautical miles (Kennedy 1996, p. 12). And in 1999, NASA's $125 million Mars Climate Orbiter was lost and presumed destroyed after navigational errors resulting in part from confusion between pound-forces and newtons (Euler et al. 2001, p. 7).

Stardust has no specific support for multiple units of the same dimension. However, one can simply consider the units to be distinct dimensions, though at the cost of explicit conversions between units.

6.4 Implementation of dimensions

Neither ICS nor CVC Lite directly support dimensions, so Stardust reduces a constraint on dimensions to a conjunction of constraints on the exponents: $\mathsf{M}^a = (\mathsf{M} * \mathsf{S})^b$, which is equivalent to $(\mathsf{M}^a) * (\mathsf{S}^0) = (\mathsf{M}^b) * (\mathsf{S}^b)$, reduces to $(a = b) \wedge (0 = b)$. Without existentials, that would be the end of the story, since every index expression of dimension sort can be reduced to a normal form in which each base dimension or (*universally* quantified) dimension variable appears once and in some particular order (Kennedy 1996, pp. 16–17). Then equality is just the conjunction of equalities of exponents. However, existentials require that we actually solve for dimension variables, but this is quite easy. Given a normal form equation $i_1 = i_2$ containing a factor $\widehat{a}$ (with nonzero exponent), we first rearrange the equation into the form $\mathtt{NODIM} = (i_1^{-1}) * i_2$ and distribute the -1 over the factors in i_1, yielding an equation $\mathtt{NODIM} = \widehat{a}^k * j_1^{k_1} * \cdots * j_n^{k_n}$, where $k \neq 0$. Multiplying both sides by $\widehat{a}^{-k}$ yields $\widehat{a}^{-k} = j_1^{k_1} * \cdots * j_n^{k_n}$; raising both sides to the power $1/(-k)$ gives the solved form $\widehat{a} = \ldots$.

6.5 Related work on invaluable refinements

Our term "invaluable refinement" is new, but similar notions have come up in other contexts. The *qualified types* of Foster (2002) encompass a variety of flow-sensitive, invaluable properties: Zero or more qualifiers, under a partial order (reminiscent of datasort refinements), may appear with a type. Foster's qualified type annotations are of two forms: $\mathtt{annot}(e, Q)$, which adds the qualifier Q to the type inferred for e (a kind of cast), and $\mathtt{check}(e, Q)$, which directs the system to check that Q is among the qualifiers of the type inferred for e. Thus, as with dimensions in our system, qualified types are based on annotations provided by the user and cannot be checked at runtime. In our system, we suspect that either a datasort refinement or an index refinement with a domain of finite sets of constants (the qualifiers) would suffice to model qualified types, with some kind of cast—some well-named identity function—acting as $\mathtt{annot}(e, Q)$, and type annotation $(e : A)$ with the appropriate refinement acting as $\mathtt{check}(e, Q)$.

Garden-variety Hindley-Milner typing also supports invaluable refinements. A *phantom type* (Finne et al. 1999; Leijen and Meijer 1999) is an ordinary datatype with a "phantom" polymorphic type parameter that is not tied to the values of that type, at least not in the obvious way that α is tied to $\alpha\ \mathsf{list}$. Phantom types can even mimic integer index refinements by encoding integers through dummy types, as Blume (2001) does in his "No Longer Foreign Function Interface" for Standard ML of New Jersey. From the user's point of view, integer index refinements seem more natural.

Phantom types can also be used as value refinements, but the typechecker's ability to reason based on inversion is limited. For invaluable refinements there are few interesting inversion principles, but when phantom types are used to encode a value-based property this is a serious shortcoming, especially since exhaustiveness of pattern matching cannot be shown. Hence, researchers have designed "first-class" phantom types, under various names, e.g. Cheney and Hinze (2003); Fluet and Pucella (2006); Xi et al. (2003); Peyton Jones et al. (2006). This approach lacks one virtue of phantom types: that one can use a standard compiler.

Phantom types (whether first- or second-class) can be seen as tantamount to index refinements in which the index objects are types. These systems lack intersection types, so they cannot transparently express conjunctions of refinement properties. More fundamentally, when the index objects are types, index equality is type equivalence—which, as equational theories go, is rather impoverished. It is no coincidence that a standard example of phantom types is an interpreter for a tiny typed language, where (in our terminology) terms in the interpreted language are indexed by types. The encoding from the problem domain is trivial, because the source language's types are a superset of the interpreted language's types. When that is not the case, such encodings become nontrivial.

In order to (again, in our terminology) obtain richer index domains than their current type expressions, phantom type systems have been enriched with elements of traditional dependent typing (Sheard 2004). Unlike ours, these systems allow users to write their own proofs of properties in undecidable domains. From a user's perspective, this approach seems more complex than ours.

Ephemeral refinements (Mandelbaum et al. 2003) may be a form of invaluable refinement as well: the refinements are about 'the state of the world', which is not a manipulable value in SML and similar type systems. If we consider ephemeral refinements involving mutable storage, a monadic formulation of ephemeral refinements would reify the state into a value and the ephemeral/invaluable refinement of the state into a value refinement. Think of Haskell's state monad with a refinement about the array's contents: the contents of the array are part of the world encapsulated by the monad. However, given an ephemeral refinement that encodes information that cannot be directly inspected, such as (some property of) the bytes written to standard output, there is nothing to reify; unless the program is modified to store that information, there is no value to refine. Thus, both value and invaluable refinements should be useful when effects are encapsulated monadically.

Finally, Refinement ML (Davies 2005) does not support invaluable refinements. In that system, the inhabitants of the datasorts are specified through regular tree grammars in which the symbols are the datatype's constructors; the only way to define datasorts that are not perfectly synonymous is to specify that they are inhabited by different sets of values. (Mere laziness kept us from following the same strategy: we did not want to bother transforming regular tree grammar-based specifications into constructor types!)

7. The design of Stardust

Stardust consists of a parser, a few preprocessing phases, a translator from the source language to let-normal form, and a typechecker that includes interfaces to external constraint solvers, to which we delegate much of the work of integer constraint solving.

The type system presents several implementation challenges. The first is that certain rules pretend that we can somehow guess how to instantiate index variables, for example, when eliminating a universal quantifier `-`**all** a `: sort-` . The usual approach, which we follow, is to postpone instantiation by generating constraints with existential variables. However, for efficient typechecking, constraint solving must be online. Otherwise, if we check `f(x)` where `f` : A `&` B, we may choose `f` : A, continue typechecking to the end of the block, find that the constraint is false, backtrack and choose `f` : B, etc. If $A = \mathsf{list}(0) \to A'$ and `x` : $\mathsf{list}(1)$, we should know immediately that trying `f` : A is wrong, since $0 = 1$ is invalid. Thus, we give the additional constraint to the solver on the fly, and when it reports that $0 = 1$ is invalid we can proceed immediately to consider `f` : B. For n such choices, if the program is ill typed and all choices (would) ultimately fail, this takes us from typechecking the block 2^n times to only n times.

A fundamental challenge is making typechecking with intersection and union types fast enough. To check an expression against A `&` B, we check it against A and then B, doubling typechecking time. To check an expression against $A \vee B$, we check it against A and, if that fails, against B, doubling typechecking time in the worst case. Dually, if we have a known expression (such as a variable) of type A `&` B, we first assume that it has type A; if typechecking subsequently fails, we assume it has type B. Finally, if we have a known expression of type $A \vee B$, it could have either type and so we must typecheck first under the assumption that it has type A, and then under type B. Thus, in the worst case, typechecking is exponential in the number of intersections and unions appearing in the program.

Intersections and unions affect error reporting as well: ideally we might like reports of the form "checking against A `&` B failed: could not check against B when $x : C_1$ (where $x : C_1 \vee C_2$)", in addition to a program location. Still, bidirectionality gives us some advantage over typecheckers based on unification. As Pierce and Turner (1998) and Davies (2005) have observed, in a bidirectional system, the location reported is more likely to be the real cause of the error, which is not always the case when unification alone is used.

7.1 Interface to an ideal constraint solver

We would like a constraint solver that supports the following for all index domains of interest:

1. A notion of *solver context* (represented by Ω) that encapsulates assumptions;
2. An ASSERT operation taking a context Ω and proposition P, yielding one of three answers:
   (a) Valid if P is already valid under the current assumptions Ω;
   (b) $\mathsf{Invalid}$ if P is unsatisfiable, that is, leads to an inconsistent set of assumptions;
   (c) $\mathsf{Contingent}(\Omega')$ if P is neither valid under the current assumptions Ω nor inconsistent when added to Ω; yields a new context $\Omega' = \Omega, P$.
3. A VALID operation taking a context Ω and proposition P, and returning one of two possible answers:
   (a) Valid if P is valid under the current assumptions;
   (b) $\mathsf{Invalid}$ otherwise.

Implicit in this specification is that the contexts Ω are persistent: if $\mathrm{ASSERT}(\Omega_1, P)$ yields $\mathsf{Contingent}(\Omega_2)$, the "earlier" context Ω_1 should remain unchanged. This is a key property, given all the backtracking the typechecker does. Where a constraint solver does not have this property, it can be simulated, though at some cost; see Section 8.1. Likewise, where the constraint solver does not support an index domain, propositions in that domain must be reduced to propositions in a supported domain.

7.2 Constraint-based typechecking

The typechecker has a notion of *state* that is independent of the particular constraint solver used. It includes index assumptions such as the index sorting $a{:}\mathsf{int}$ and the proposition $a > 0$; an accumulated constraint that needs to be valid to make the program well typed; a substitution containing solutions for existentially quantified variables; and a representation of the external constraint solver's state.

Our constraint solvers do not support existential variables at all (ICS) or support them incompletely (CVC Lite), which significantly affects the design. The typechecker itself manages existentials in the integer domain, and lies to the constraint solver by telling it that existential variables are universal. Therefore, when adding a constraint we cannot immediately check its validity, since the constraint may include existential variables that the solver thinks are universal: we cannot directly check $\widehat{a}{:}\mathsf{int} \models \widehat{a} = 0$ (meaning $\exists a.\ a = 0$), only $a{:}\mathsf{int} \models a = 0$ (meaning $\forall a.\ a = 0$) with a universally quantified. Clearly, the first relation should hold and the second should not. Fortunately, we can still "fail early" (recalling `f` : A `&` B from the example earlier). Instead of checking validity, we assert the new constraint, adding it to the assumptions. If the resulting assumptions are inconsistent (as with $0 = 1$, or—less trivially—$a = a + 1$), no instantiation of existential variables can make the constraint valid, so we can correctly fail, and backtrack as needed.

Of course, we must check validity of the constraint at some point! Otherwise, given a constraint $b = 0$, we would conclude $b{:}\mathsf{int} \models b = 0$ since $b = 0$ is a consistent assumption. Therefore, in addition to asserting $b = 0$, we add it to a constraint built up in a manner similar to off-line constraint solving. Eventually, the typechecker tries to solve for existentials (applying a simplistic and probably incomplete rewriting algorithm) and asks the solver whether the built-up constraint is valid.

7.3 Interface to ICS

Stardust includes an interface to ICS (de Moura et al. 2004) as an external constraint solver. ICS has cooperating decision procedures for fragments of rational arithmetic and several theories; the typechecker presently uses only the arithmetic theory. While there is a notion of "current context" in the ICS interface (for example, ICS's ASSERT operation takes only a proposition and implicitly uses the current context as the Ω), previously constructed contexts can be saved and restored quickly, yielding an interface extremely close to the idealized one presented above. This is no coincidence: we designed our system with ICS in mind. We do not use ICS as a library; instead, it runs as a separate process and we communicate through Unix pipes.

7.4 Interface to CVC Lite

Stardust also includes an interface to CVC Lite (Barrett and Berezin 2004), the successor to CVC, the Cooperating Validity Checker (Stump et al. 2002), which in turn succeeded SVC, the Stanford Validity Checker (Barrett et al. 1996). CVC Lite has cooperating decision procedures for fragments of integer and rational arithmetic, Boolean propositions (including conjunction, disjunction, negation, and implication), and other theories; we presently use only the integer and Boolean theories. It has limited support for quantifiers, both universal and existential (free variables are, as in ICS, considered universal); a response of Invalid may be given even when an existential solution exists. We have not explored whether that limited support is enough for Stardust; if as powerful as our home-grown existentials, we might get a simpler design.

Unlike ICS, CVC Lite does not support persistent contexts. We discuss the impact of this in Section 8.

CVC Lite has recently become CVC3. We hope to add support for CVC3, which should allow us to easily implement an index domain where the objects are inductive datatypes.

7.5 No refinement restriction

Davies' Refinement ML (Davies 2005) has a *refinement restriction* on intersection types: an intersection A & B is well formed only if A and B are refinements of the same simple type. For example, even & odd is permitted if even and odd both refine list; likewise, (even $\rightarrow$ odd) & (odd $\rightarrow$ even) is permitted, since each component of the intersection refines list $\rightarrow$ list. On the other hand, list & (list $\rightarrow$ list) and int & string do not satisfy the refinement restriction; in the first, lists and functions are incompatible, while int and string are distinct base types. Because of the refinement restriction, typechecking in Refinement ML is conservative over Standard ML: every program that is well typed in Refinement ML is also well typed in Standard ML.

In contrast, Stardust does not enforce a refinement restriction on intersections and unions. Stardust also does not check code that it knows (through the type system) to be dead. Thus, it is *not* conservative in the sense that Refinement ML is.

7.6 Let-normal translation

Stardust translates programs into a let-normal form before typechecking them, enabling a more efficient typechecking algorithm than the one arising directly from the tridirectional system (Dunfield and Pfenning 2004). Our translation is unusual in that all synthesizing forms, including variables, are let-bound; this helps to guarantee that no programs that would be well typed if left untranslated (i.e. , well typed in the tridirectional system) become ill-typed when translated. Because we have that guarantee, the let-normal translation is completely transparent to the user. The details of the transformation and the proof that no well-typed programs become ill-typed (and vice versa) after translation are beyond the scope of this paper; see Dunfield (2007, Ch. 5).

8. Speed of typechecking

In this section, we give the time needed to typecheck several example programs, and discuss some of the factors affecting performance.

Input program	Wall-clock time in seconds: ICS	CVC Lite (library)	CVC Lite (standalone)
redblack-full	1.9	8.2	9.2
redblack-full-bug1	1.6	6.8	8.1
redblack	< 1	< 1	< 1
rbdelete	*	37.7	31.6
bits	*	9.5	4.0
bits-un	33.5	298.5	241.4

Table 1. Time required for typechecking

The times indicated are under Standard ML of New Jersey version 110.59 on a 4-CPU Intel Xeon (3 GHz) and 2 GB RAM. The constraint solvers were ICS version 2.0 (November 2003) and CVC Lite version 20070121 (January 2007). An asterisk (*) indicates programs for which the constraint solver gives a wrong answer, or the system otherwise fails. 'redblack-full' is the program in Figure 3; 'redblack-full-bug1' is that program with a bug introduced; 'redblack' is the same program with index refinements removed, using only datasort refinements. 'bits' contains several functions on bitstrings. 'bits-un' is similar, but uses union types more extensively. The very long typechecking time is due in part to having to check certain expressions against each component of a 4-way union; those expressions themselves are of a 4-way union type.

All of the dimension examples in this paper typecheck in less than one second, which appears to be typical for code that does not use intersection and union types.

8.1 Impact of solver interfaces

Stardust communicates with ICS through Unix pipes. This is not very efficient: experiments suggest that the overhead of sending one command and receiving one response is 20–40% for ICS.

We can also communicate with CVC Lite through Unix pipes, but we have also implemented a direct interface to a shared library through CVC Lite's C-level API and the SML/NJ NLFFI. As we expected, this speeds typechecking in most cases.

For CVC Lite, another source of inefficiency is CVC Lite's inability to rapidly switch back to previously visited contexts. Unlike ICS, in which contexts are persistent and can be recalled instantaneously, CVC Lite can roll back only to ancestors of its current context. This requires us to "replay" assertions; typically, 20–50% of transactions with CVC Lite are replay assertions. This suggests that for our purposes, persistent context in a constraint solver is useful but not absolutely essential.

8.2 Conservation of speed

We believe that Stardust conserves typechecking speed, in the sense that checking a program—more usefully, a block—that does not use property types should take polynomial time (as with monomorphic SML programs). This is subject to the caveat that property types appear in the types of many primitive functions; any block that uses * actually uses intersection types. This (unproven) claim rests on our belief that the underlying type system has a subformula property (Prawitz 1965, p. 53): formulas (here, type expressions)—and, therefore, connectives like &—appear in parts of a derivation only if they appear as subformulas of the goal (where types in annotations are goals).

8.3 Scaling up

Typechecking is modular, in the specific sense that each block of mutually recursive function declarations can be checked independently of each other block. For example, given a program with two mutually recursive functions *f1*,*f2* followed by a function *g*, i.e. **fun** *f1* . . . **and** *f2* . . . **fun** *g*, if checking *g* fails, it cannot be blamed on a choice made while checking *f1* and *f2*.

Thus, while property types can make checking a particular block very slow, adding a second block of the same complexity will only double typechecking time. This "block independence" should mean that once we have acceptable efficiency for typical programs of a few hundred lines, only linear speedup will be required to be acceptably efficient on larger programs. Moreover, we should be able to get that speedup through an easy form of distributed computation: If we send each block to a different processor for typechecking, the communication cost will be low, since the input is small and the output is tiny: typechecking either succeeds, or fails with some error information, for that block.

Davies' work (Davies 2005) suggests that there would be no major barriers to adding ML modules to Stardust. This would allow users to give refined types in module signatures, providing important documentation; it also does not add to the volume of annotations, since signatures must be written out anyway.

9. Related work

The type system that underlies Stardust is based on prior work (Dunfield and Pfenning 2004; Dunfield 2007), which includes intersections, unions, index refinements, and datasort refinements.

Intersection types are fairly old (Coppo et al. 1981); type inference is undecidable (Amadio and Curien 1998). Reynolds (1996), who was the first to use intersection types in a practical programming language, proved that typechecking is PSPACE-hard. Intersection types (sometimes with union types too) have also been used to infer control flow properties, e.g. Palsberg and Pavlopoulou (2001) and for compositional type inference, e.g. Bakewell et al. (2005).

Freeman and Pfenning (1991) introduced datasort refinements combined with intersection types, showed that full type inference was decidable under the refinement restriction, and developed an inference algorithm based on techniques from abstract interpretation. Interaction with effects in a call-by-value language was first addressed conclusively by Davies and Pfenning (2000), who restricted intersection introduction to values, pointed out the unsoundness of distributivity, and proposed a practical bidirectional checking algorithm. Davies' datasort refinement checker (Davies 2005) supports all of Standard ML. Pierce (1991) gave examples of programming with intersections and unions in a pure λ-calculus, relying on syntactic markers which are not needed in our system.

Xi (1998) formulated Dependent ML, a bidirectional type system with index refinements for a variant of ML and implemented it as an extension to Caml Light. He showed a number of applications (using the integer constraint domain), including array bounds check elimination. To cope with some issues arising from existential index quantification, Xi's approach transformed programs into a let-normal form before typechecking them; however, typechecking is then incomplete, in the sense that some programs that typecheck in their original form do not typecheck after translation. We attack similar issues with existentials in our work in a broadly similar way, through translation to our own peculiar variant of let-normal form. However, our let-normal typechecking *is* complete (as well as sound) (Dunfield 2007, Ch. 5).

The ancestor of index refinement is the notion of dependent type developed by Martin-Löf and used in various theorem proving systems. The types $\Pi x{:}A.\ B$ and $\Sigma x{:}A.\ B$ roughly correspond to the universal and existential quantifiers over indices; however, instead of drawing x from a restricted index domain, dependent types draw x from terms of type A. This is powerful but (in any language in which some programs do not terminate) undecidable.

A number of systems have tried to tame dependent types. In Cayenne (Augustsson 1998), typechecking "times out" after a given number of steps. In Epigram (McBride and McKinna 2004), all well-typed programs terminate, so type equivalence is decidable. The dependent indices are elements of *inductive families* of constructors; the example of natural numbers with *zero* and *succ* constructors is probably the canonical one. In the system of Chen and Xi (2005), as in Epigram, users can write explicit proofs of type equivalences; unlike Epigram, the language itself is not restricted—decidability comes by restricting the terms that can inhabit indices. A similar system is described by Licata and Harper (2005), who give a detailed comparison of these and related type systems.

We see two major advantages of our approach over these systems. The first is that our system needs no guidance beyond type annotations. The second is the legibility and clarity of the types themselves. We believe that the types in our system are easier to understand than in these more traditional dependent type systems. It could be argued that both flavors of system add to the number of 'levels' a user must think about—ours adds index refinements (and datasorts, but let us not muddy the comparison), while theirs add dependent typing and kind-level programming. However, the level we add seems to be *lower* than the types in conventional type systems, rather than higher.

Our approach also differs significantly from extended static checking (Leino 2001), which, like our system, uses annotations to express properties and processes the invariants at compile time, without the user writing explicit proofs. However, a report from ESC must be interpreted quite differently from a report from Stardust. In the extended static checking framework, a favorable report simply means that no problem was *found* in the program; it does not guarantee that the properties actually hold. In Stardust, a report that the program typechecks means that the properties really do hold (subject to the usual caveats about bugs in Stardust itself, in the compiler, etc.). On the other hand, that limitation is a key reason that ESC can express many properties Stardust cannot.

10. Conclusion

We have presented the first implementation of a system combining intersections, unions, and type refinements, in which the expressible properties, while limited by decidability concerns, are legible and straightforward. We have formulated and implemented index domains of integers, Booleans, and dimensions. While typechecking speed is adequate for most of our examples, it is not fully satisfactory; more work is needed to allow extensive use of intersection and union types.

As the set of supported domains grows, an already present problem grows with it: the scalability of the refinements themselves. Different invariants will be important in different parts of a program. It is perfectly reasonable to index lists by length; it is also perfectly reasonable to index them by their contents, or by some property of a particular element. Our current approach requires that one either cram all manner of indices into a tuple, and index by that, or create new datatypes for each new property, each with its own refinement. The first technique is brazenly anti-modular; the second leads to code duplication and tedium. Thus, designing truly modular refinements is an important goal for future work.

We intend to explore additional index domains including bit vectors, inductive families, functional arrays (vectors), fragments of set theory, and regular languages, all of which have practical decision procedures and are therefore compatible with our approach.

We are in the process of adding parametric polymorphism to the type system and implementation, and are investigating extensions to call-by-name and call-by-need semantics.

Acknowledgments. Frank Pfenning's advice was invaluable; support from the US National Science Foundation under grant CCR-0204248 was valuable; the anonymous reviewers caught several mistakes. Portions of Stardust are based on "Restricted ML" (Tolmach and Oliva 1998).

References

Roberto Amadio and Pierre-Louis Curien. *Domains and lambda calculi*, volume 46 of *Cambridge Tracts in Theoretical Comp. Sci.*, chapter 3. Cambridge Univ. Press, 1998.

Lennart Augustsson. Cayenne—a language with dependent types. In *ICFP*, pages 239–250, 1998.

Adam Bakewell, Sébastien Carlier, A. J. Kfoury, and J. B. Wells. Inferring intersection typings that are equivalent to call-by-name and call-by-value evaluations. Technical report, Church Project, Boston University, 2005.

Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the Cooperating Validity Checker. In *Int'l Conf. Computer Aided Verification (CAV '04)*, pages 515–518, 2004.

Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity checking for combinations of theories with equality. In *Int'l Conf. Formal Methods in Computer-Aided Design (FMCAD '96)*, pages 187–201, 1996.

Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Elect. Notes in Theoretical Comp. Sci.*, 59 (1), 2001.

Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP*, pages 66–77, 2005.

James Cheney and Ralf Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift f. math. Logik und Grundlagen d. Math.*, 27:45–58, 1981.

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

Rowan Davies. *Practical refinement-type checking*. PhD thesis, Carnegie Mellon University, 2005. CMU-CS-05-110.

Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ICFP*, pages 198–208, 2000.

Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, and Natarajan Shankar. The ICS decision procedures for embedded deduction. In *Int'l Joint Conf. Automated Reasoning (IJCAR '04)*, pages 218–222, 2004.

Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007. CMU-CS-07-129. To appear; latest version at `http://www.cs.cmu.edu/~joshuad/thesis/`.

Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Found. Software Science and Computation Structures (FOSSACS '03)*, pages 250–266, 2003.

Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *POPL*, pages 281–292, January 2004.

Edward A. Euler, Steven D. Jolly, and H. H. 'Lad' Curtis. The failures of the Mars Climate Orbiter and Mars Polar Lander: a perspective from the people involved. In *AAS Guidance and Control Conf.*, 2001. `http://brain.cs.uiuc.edu/integration/AAS01_MCO_MPL_final.pdf`.

Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In *ICFP*, pages 114–125, 1999.

Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. ArXiv postprint, `http://arxiv.org/abs/cs.PL/0403034`, January 2006.

Jeffrey Scott Foster. *Type qualifiers: lightweight specifications to improve software quality*. PhD thesis, University of California, Berkeley, 2002.

Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI*, pages 268–277. ACM Press, 1991.

Stefan Kahrs. Red-black trees with types. *J. Functional Programming*, 11(4):425–432, 2001.

Andrew Kennedy. *Programming languages and dimensions*. PhD thesis, University of Cambridge, 1996.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *USENIX Conf. Domain-Specific Languages (DSL '99)*, pages 109–122, 1999.

K. Rustan M. Leino. Extended Static Checking: A ten-year perspective. In *Dagstuhl Anniversary Conf.*, pages 157–175, 2001.

Daniel R. Licata and Robert Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University, 2005.

Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *ICFP*, pages 213–226, 2003.

Conor McBride and James McKinna. The view from the left. *J. Functional Programming*, 14(1):69–111, 2004.

Robin Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17(3):348–375, 1978.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge, 1998.

Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Functional Programming*, 11(3):263–317, 2001.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP*, 2006.

Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

Benjamin C. Pierce and David N. Turner. Local type inference. In *POPL*, pages 252–265, 1998. Full version in *ACM Trans. Prog. Lang. Sys.*, 22(1):1–44, 2000.

Dag Prawitz. *Natural Deduction*. Almqvist & Wiksells, 1965.

John C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier, 1983. `http://www.cs.cmu.edu/afs/cs/user/jcr/ftp/typesabpara.pdf`.

John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.

Tim Sheard. Languages of the future. *SIGPLAN Notices*, 39(12): 119–132, 2004. OOPSLA '04.

Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In *Int'l Conf. Computer Aided Verification (CAV '02)*, pages 500–504, 2002.

Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *J. Functional Programming*, 8(4):367–412, 1998.

Hongwei Xi. Applied Type System (extended abstract). In *TYPES 2003*, LNCS, pages 394–408. Springer, 2004.

Hongwei Xi. *Dependent types in practical programming*. PhD thesis, Carnegie Mellon University, 1998.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *POPL*, pages 224–235, 2003.

Abbrev.: *ICFP* = Int'l Conf. Functional Programming; *PLDI* = Programming Language Design and Implementation; *POPL* = Principles of Programming Languages.