

# Programming with Dependently Typed Data Structures\*

Hongwei Xi

Computer Science Department  
Boston University

## Abstract

The mechanism for declaring datatypes in functional programming languages such as ML and Haskell is of great use in practice. This mechanism, however, often suffers from its imprecision in capturing various invariants inherent in data structures. The introduction of dependent datatypes in Dependent ML (DML) can partly remedy the situation, allowing the programmer to model data structures with significantly more accuracy. In this paper, we present several programming examples (e.g., implementations of random-access lists and red-black trees) to illustrate some practical use of dependent datatypes in capturing relatively sophisticated invariants in data structures. We claim that dependent datatypes can enable the programmer to implement algorithms in a manner that is more efficient, more robust and easier to understand.

## 1 Introduction

The mechanism that allows the programmer to declare datatypes for modeling data structures in functional programming languages such as Standard ML (Milner, Tofte, Harper, and MacQueen 1997) and Haskell (Peyton Jones et al. 1999) is of great use in practice. It can offer both convenience in programming and clarity in code. In practice, we often encounter a situation where the declared datatype for a data structure can not accurately capture certain invariants inherent in the data structure. For instance, if what we need is a data structure for pairs of integer lists with equal length, we often declare a datatype in Standard ML or Haskell that is for *all* pairs of integer lists. This inaccuracy problem is often a rich source for run-time program errors. For instance, a function that should only receive as its argument a pair of integer lists with equal length may be mistakenly applied to a pair of integer lists of unequal length. Unfortunately, such a mistake causes no type errors if pairs of integer lists of equal length are given a type that is for all pairs of integer lists, and thus can usually go unnoticed until at run-time, when debugging often becomes much more demanding than at compile-time. More important, this adversely affects our ability in building robust software.

The inaccuracy problem becomes more serious when we start to implement more sophisticated data structures such as red-black trees, binomial heaps, ordered lists, etc. There are certain invariants in these data structures that we must maintain in order to implement them correctly. For instance, a correct implementation of an insertion operation on a red-black tree should always return a red-black tree. If we can form a datatype to precisely capture the properties of being a red-black tree, then it becomes possible to detect a program error through type-checking when such an error leads to a violation of the captured properties. This is evidently a desirable feature in programming if it can be made practical.

The need for forming more accurate datatypes partially motivated the design of Dependent ML (DML) (Xi and Pfenning 1999; Xi 1998), an enrichment of ML with a restricted form of dependent types. More precisely, DML is a language schema. Given a constraint domain  $C$ ,  $DML(C)$  is the language in the

---

\*This work is partially supported by NSF grants no. CCR-0081316 and no. CCR-0092703, and a grant from the Ohio Board of Regency. A previous version of the paper was presented in the Workshop on Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99).

schema where all type index expressions are drawn from  $C$ . Roughly speaking, a type index expression is simply a term that can be used to index a type. Type-checking in  $\text{DML}(C)$  can then be reduced to constraint satisfaction in the constraint domain  $C$ . In this paper, we restrict  $C$  to some integer domain and use the name  $\text{DML}$  for this particular  $\text{DML}(C)$ .<sup>1</sup> We have currently finished a prototyped implementation of  $\text{DML}$  in Objective Caml, which allows us to experiment with programming in a dependently typed setting and in particular, to experiment with dependently typed data structures.

An alternative approach to forming more accurate datatypes is through the use nested datatypes (Bird and Meertens 1998). For instance, a nested datatype exactly representing red-black trees can also be readily formed (Hinze 1999; Kahrs 2001). However,  $\text{DML}$ -style dependent types and nested datatypes are orthogonal features. In particular, we are to present an example where a nested datatype is refined into dependent nested datatypes for capturing more properties. We find that nested datatypes are often less intuitive than  $\text{DML}$ -style dependent types when used in practice to capture certain information (e.g., size information) in data structures. Also, there are many important applications of  $\text{DML}$ -style dependent types such as array bound check elimination (Xi and Pfenning 1998) that seem difficult to handle with nested datatypes. However, type-checking for nested datatypes is significantly easier than for  $\text{DML}$ -style dependent types.

We use `typewriter` font in this paper to represent code written in the concrete syntax of  $\text{DML}$ . A significant consequence from the introduction of dependent types is the loss of the notion of principal types in  $\text{DML}$ . For instance, both of the following types can be assigned to an implementation in  $\text{DML}$  that zips two lists together.

```
'a list * 'b list -> ('a * 'b) list
{n:nat} 'a list(n) * 'b list(n) -> ('a * 'b) list(n)
```

The first type has the usually meaning, while the second one implies that for every natural number  $n$ , the function yields a list with length  $n$  when applied to a pair of lists with length  $n$ . Notice that we use  $(\tau)\text{list}(n)$  for the type of a list with length  $n$  in which every element has the type  $\tau$ . If a dependent type is to be assigned to a function in  $\text{DML}$ , it is the responsibility of the programmer to annotate the function with such a dependent type. This is probably the most significant difference between the programming styles in  $\text{ML}$  and in  $\text{DML}$ . In practice, we observe that the type annotations in a typical  $\text{DML}$  program often constitutes less than 20% of the entire code. Since dependent type annotations can often lead to more accurate reports of type errors and serve as informative program documentation, we feel that the programming style in  $\text{DML}$  is acceptable from a practical point of view. We will provide some concrete examples in support of this claim, including implementations of red-black trees and random-access lists. Some of the implementations are directly adopted from the corresponding ones in (Okasaki 1998). The implementations in  $\text{DML}$  have several advantages over the original ones. We have verified more invariants in these implementations. For instance, it is verified in the type system of  $\text{DML}$  that the function that looks up for the  $i$ th element in a random-access list can never issue a run-time exception as long as  $i$  is less than the length of the list. Also the type annotations in the implementations, which can be fully trusted since they are mechanically verified, offer some pedagogical values.

This is not a technical paper on  $\text{DML}$ . Our primary purpose is to reach a wider audience for  $\text{DML}$ , presenting some interesting examples in support of the claim that dependent types can be made both useful and practical in programming. As the use of dependent types in practice is at least rare in the literature, we believe that this paper can offer the programmer a good opportunity to see what dependent types are able to do in practice. In this paper, it is neither possible nor necessary to formally present the entirety of  $\text{DML}$ . Instead, we present a core language  $\text{ML}_0^{\Pi, \Sigma}$  of  $\text{DML}$ , formalizing both static and dynamic semantics of  $\text{ML}_0^{\Pi, \Sigma}$  and stating its type soundness (subject reduction). This serves as the foundation of  $\text{DML}$ . We then focus on presenting some concrete programming examples in  $\text{DML}$  as well as some intuitive explanation. We refer the interested reader to (Xi 1998) for the formal development of  $\text{DML}$ , though we strongly believe that this is largely unnecessary for comprehending this paper.

We organize the rest of the paper as follows. In Section 2, we formally present an explicitly typed language  $\text{ML}_0^{\Pi, \Sigma}$ , which can be regarded as a core of  $\text{DML}$ . We present both static and dynamic semantics of  $\text{ML}_0^{\Pi, \Sigma}$  and state its type soundness (subject reduction) theorem. We then introduce a prototype

---

<sup>1</sup>However, we will present an example in Section 4.4, where a constraint domain consisting of algebraic terms is used.

implementation of DML in Section 3, presenting some of its unique and significant features and explaining informally how type-checking involving dependent datatypes is performed. We present several examples in Section 4 to illustrate how various program invariants can be captured in the type system of DML. As the use of dependent types in practical programming is at least rare at present, we hope that the presented examples can provide the reader with a solid feel as to what can be expected of dependent types in practice. Lastly, we discuss some related work and conclude.

## 2 A Core Language of Dependent ML

We start with an explicitly typed language  $\text{ML}_0^{\Pi, \Sigma}$ , which can be regarded as the core of DML. The language  $\text{ML}_0^{\Pi, \Sigma}$  essentially extends the simply typed call-by-value  $\lambda$ -calculus with a form of dependent types and general recursion.

### 2.1 Syntax

The syntax for  $\text{ML}_0^{\Pi, \Sigma}$  is given in Figure 1. We fix an integer domain and restrict index expressions, namely, the expressions that can be used to index a type, to this domain. This is a sorted domain and subset sorts can be formed. For instance, we use *nat* as an abbreviation for the subset sort  $\{a : \text{int} \mid a \geq 0\}$ , which is for index expressions with natural numbers as their values. We use  $\text{if}(P, i_1, i_2)$  for an index expression that equals  $i_1$  if  $P$  holds, and  $i_2$  otherwise. We use  $\delta(\vec{i})$  for a base type indexed with a (possibly empty) sequence of index expressions  $\vec{i}$ . For instance,  $\text{bool}(0)$  and  $\text{bool}(1)$  are types for boolean values *false* and *true*, respectively; for each integer  $i$ ,  $\text{int}(i)$  is the singleton type for integer expressions with value equal to  $i$ .

We use  $\phi \models P$  for a satisfaction relation, meaning  $P$  holds under  $\phi$ , that is, the formula  $(\phi)P$ , defined below, holds in the integer domain.

$$\begin{aligned} (\cdot)\Phi &= \Phi \\ (\phi, a : \text{int})\Phi &= (\phi)\forall a : \text{int}.\Phi \\ (\phi, \{a : \gamma \mid P\})\Phi &= (\phi, a : \gamma)(P \supset \Phi) \\ (\phi, P)\Phi &= (\phi)(P \supset \Phi) \end{aligned}$$

For instance, the satisfaction relation

$$a : \text{nat}, a \neq 0 \models a - 1 \geq 0$$

holds since the following formula is true in the integer domain.

$$\forall a : \text{int}. a \geq 0 \supset (a \neq 0 \supset a - 1 \geq 0)$$

Note that the decidability of the satisfaction relation depends on the constraint domain. Although the syntax in Figure 2 allows non-linear<sup>2</sup> integer constraints, such constraints are rejected immediately in our implementation. We currently translate the problem of determining whether a given (linear) constraint is satisfiable into some integer programming problem, for which there are various existing methods.

We use  $\Pi \vec{a} : \vec{\gamma}.\tau$  and  $\Sigma \vec{a} : \vec{\gamma}.\tau$  for the usual dependent function and sum types, respectively. A type of the form  $\Pi \vec{a} : \vec{\gamma}.\tau$  is essentially equivalent to  $\Pi a_1 : \gamma_1 \dots \Pi a_n : \gamma_n.\tau$ , where we use  $\vec{a} : \vec{\gamma}$  for  $a_1 : \gamma_1, \dots, a_n : \gamma_n$ . Note that  $\gamma_k$  may contain free occurrences of  $a_j$  for  $1 \leq j < k \leq n$ . In practice, we also have types of the form  $\Sigma \vec{a} : \vec{\gamma}.\tau$ , which we omit here for simplifying the presentation. In particular, given a type constructor  $\delta$  that takes index expressions  $\vec{i}$  of sorts  $\vec{\gamma}$  to form a base type  $\delta(\vec{i})$ , we often use  $\delta$  for  $\Sigma \vec{a} : \vec{\gamma}.\delta(\vec{a})$ . For instance,  $\text{bool}$  and  $\text{int}$  stand for  $\Sigma a : \text{bool}.\text{bool}(a)$  and  $\Sigma a : \text{int}.\text{int}(a)$ , respectively, where the sort  $\text{bool} = \{a : \text{int} \mid 0 \leq a \leq 1\}$ .

<sup>2</sup>A constraint is non-linear if contains a non-linear term such as  $a_1 * a_2$  for some index variables  $a_1$  and  $a_2$ .

index constants

$$c_I ::= \dots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \dots$$

index expressions

$$i ::= a \mid c_I \mid i_1 + i_2 \mid i_1 - i_2 \mid i_1 * i_2 \mid i_1 / i_2 \mid i_1 \bmod i_2 \mid \text{if}(P, i_1, i_2)$$

index propositions

$$P ::= i_1 < i_2 \mid i_1 \leq i_2 \mid i_1 > i_2 \mid i_1 \geq i_2 \mid i_1 = i_2 \mid i_1 \neq i_2 \mid \\ P_1 \wedge P_2 \mid P_1 \vee P_2$$

index sorts

$$\gamma ::= \text{int} \mid \{a : \gamma \mid P\}$$

index variable contexts

$$\phi ::= \cdot \mid \phi, a : \gamma \mid \phi, P$$

index constraints

$$\Phi ::= P \mid P \supset \Phi \mid \forall a : \gamma. \Phi$$

types

$$\tau ::= \delta(\vec{i}) \mid \mathbf{1} \mid \tau_1 * \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \Pi \vec{a} : \vec{\gamma}. \tau \mid \Sigma a : \gamma. \tau$$

contexts

$$\Gamma ::= \cdot \mid \Gamma, x : \tau \mid \Gamma, f : \tau$$

patterns

$$p ::= x \mid \langle \rangle \mid \langle p_1, p_2 \rangle \mid c[\vec{a}](p)$$

clauses

$$ms ::= (p_1 \Rightarrow e_1 \mid \dots \mid p_n \Rightarrow e_n)$$

expressions

$$e ::= x \mid f \mid c[\vec{i}](e) \mid \text{if}(e, e_1, e_2) \mid \lambda \vec{a} : \vec{\gamma}. v \mid e[\vec{i}] \mid \\ \langle \rangle \mid \langle e_1, e_2 \rangle \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid \mathbf{lam} x : \tau. e \mid e_1(e_2) \mid \\ \mathbf{case} e \mathbf{of} ms \mid \mathbf{let} x = e_1 \mathbf{in} e_2 \mathbf{end} \mid \\ \mathbf{fix} f : \tau. v \mid \langle \rangle \mid \langle i \mid e \rangle \mid \mathbf{open} e_1 \mathbf{as} \langle a \mid x \rangle \mathbf{in} e_2$$

values

$$v ::= x \mid c[\vec{i}](v) \mid \lambda \vec{a} : \vec{\gamma}. v \mid \langle \rangle \mid \langle v_1, v_2 \rangle \mid \mathbf{lam} x : \tau. e \mid \langle i \mid v \rangle$$

Figure 1: The syntax for  $\text{ML}_0^{\Pi, \Sigma}$

```

fun ack x1 x2 =
  if x1 = 0 then x2+1
  else if x2 = 0 then ack (x1-1) 1
  else ack (x1-1) (ack x1 (x2-1))
withtype {a1:nat, a2:nat} int(a1) -> int(a2) -> [a:nat] int(a)

fix ack :  $\Pi a_1 : \text{nat} \Pi a_2 : \text{nat} . \text{int}(a_1) \rightarrow \text{int}(a_2) \rightarrow \Sigma a : \text{nat} . \text{int}(a)$ .
 $\lambda a_1 : \text{nat} . \lambda a_2 : \text{nat} . \mathbf{lam} \ x_1 : \text{int}(a_1) . \mathbf{lam} \ x_2 : \text{int}(a_2) .$ 
  if ( $= [a_1, 0] (\langle x_1, 0 \rangle)$ ),
    ( $a_2 + 1 \mid + [a_2, 1] (\langle x_2, 1 \rangle)$ ),
  if ( $= [a_2, 0] (\langle x_2, 0 \rangle)$ ),
     $ack[a_1 - 1, 1] (-[a_1, 1] (\langle x_1, 1 \rangle)) (1)$ ,
  open  $ack[a_1, a_2 - 1] (x_1) (-[a_2, 1] (\langle x_2, 1 \rangle))$ 
    as  $\langle a'_2 \mid x'_2 \rangle$  in  $ack[a_1 - 1, a'_2] (-[a_1, 1] (\langle x_1, 1 \rangle)) (x'_2)$ )

```

Figure 2: The Ackermann Function in DML and  $\text{ML}_0^{\Pi, \Sigma}$

We also introduce lam-variables and fix-variables in  $\text{ML}_0^{\Pi, \Sigma}$  and use  $x$  and  $f$  for them, respectively. A lambda-abstraction can only be formed over a lam-variable while recursion (via fixed point operator) must be formed over a fix-variable. A lam-variable is a value but a fix-variable is not. We use  $c$  for constructors. We assume that the type of a constructor is of the form  $\Pi \vec{a} : \vec{\gamma} . \tau_1 \rightarrow \tau$ .<sup>3</sup>

We use  $\lambda$  for abstracting over index variables, **lam** for abstracting over variables, and **fix** for forming fixed-point expressions. Note that the body after either  $\lambda$  or **fix** must be a value. We use  $\langle i \mid e \rangle$  for packing an index  $i$  with an expression  $e$  to form an expression of a dependent sum type, and **open** for unpacking an expression of a dependent sum type.

There are two forms of application. We write  $e[\vec{v}]$  for applying  $e$  to a sequence of index expressions and  $e_1(e_2)$  for applying  $e_1$  to  $e_2$ . The meaning of such forms of application is to become clear after we present the dynamic semantics of  $\text{ML}_0^{\Pi, \Sigma}$ .

Lastly, we present an example in Figure 2 for the reader to relate the concrete syntax of DML to the syntax of  $\text{ML}_0^{\Pi, \Sigma}$ . We will return to this issue later.

## 2.2 Static Semantics

We write  $\phi \vdash \tau$  [**well-formed**] to mean that  $\tau$  is a legally formed type under  $\phi$ . For instance, the following rules are for constructing types of the forms  $\text{int}(i)$  and  $\text{bool}(i)$ , respectively.

$$\frac{\phi \vdash i : \text{int}}{\phi \vdash \text{int}(i) \text{ [well-formed]}} \quad \frac{\phi \vdash i : \{a : \text{int} \mid 0 \leq a \leq 1\}}{\phi \vdash \text{bool}(i) \text{ [well-formed]}}$$

Notice that the rule for constructing  $\text{bool}(i)$  indicates that we need to show that  $0 \leq i \leq 1$  holds in order to form such a type. Therefore,  $\text{bool}(2)$  is ill-formed. We omit other standard rules for constructing well-formed types.

$$\begin{array}{ll} \text{index substitutions} & \theta_I ::= [] \mid \theta_I[a \mapsto i] \\ \text{substitutions} & \theta ::= [] \mid \theta[x \mapsto v] \mid \theta[f \mapsto e] \end{array}$$

A substitution is a finite mapping and  $[]$  represents an empty mapping. We use  $\theta_I$  for a substitution mapping index variables to index expressions and  $\text{dom}(\theta_I)$  for the domain of  $\theta_I$ . Note that  $\theta_I[a \mapsto i]$  extends  $\theta_I$ , whose domain does not contain  $a$ , with a mapping from  $a$  to  $i$ . Similar notations are used for substitutions on variables. We write  $\bullet[\theta_I]$  ( $\bullet[\theta]$ ) for the result of applying  $\theta_I$  ( $\theta$ ) to  $\bullet$ , where  $\bullet$  can be a type, a context, an expression, etc. The standard definition is omitted. In this paper, we substitute only values for lam-variables.

We use a judgment of the form  $\phi \vdash i : \gamma$  to mean that  $i$  can be assigned the sort  $\gamma$  under the index variable context  $\phi$ . We omit the standard rules for deriving such judgments. The following rules are for

<sup>3</sup>A constructor taking no argument can be treated as a constructor taking the unit  $\langle \rangle$  as its argument.

deriving judgments of the form  $\phi \vdash \theta_I : \phi'$ , which roughly means that  $\theta_I$  has the "type"  $\phi'$ .

$$\begin{array}{c} \overline{\phi \vdash [] : \cdot} \text{ (sub-i-empty)} \\ \frac{\phi \vdash \theta_I : \phi' \quad \phi \vdash i : \gamma[\theta_I]}{\phi \vdash \theta_I[a \mapsto i] : \phi', a : \gamma} \text{ (sub-i-var)} \\ \frac{\phi \vdash \theta_I : \phi' \quad \phi \models P[\theta_I]}{\phi \vdash \theta_I : \phi', P} \text{ (sub-i-prop)} \end{array}$$

Clearly, we have the following.

**Lemma 2.1** (*Index Substitution*) *If  $\phi \vdash \theta_I : \phi'$  holds and  $\phi, \phi' \vdash i : \gamma$  is derivable, then  $\phi \vdash i[\theta_I] : \gamma[\theta_I]$  is also derivable.*

We write  $\phi \models \tau \equiv \tau'$  for the congruent extension of  $\phi \models i = j$  from index expressions to types, which is determined by the following rules. Note that we write  $\phi \models \vec{i} = \vec{i}'$  to mean  $\phi \models i_k = i'_k$  for  $k = 1, \dots, n$ , assuming  $\vec{i} = (i_1, \dots, i_n)$  and  $\vec{i}' = (i'_1, \dots, i'_n)$ . The application of these rules generates constraints during type-checking.

$$\begin{array}{c} \frac{\delta \text{ has the kind } \vec{\gamma} \rightarrow * \quad \phi \vdash \vec{i} : \vec{\gamma} \quad \phi \vdash \vec{i}' : \vec{\gamma} \quad \phi \models \vec{i} = \vec{i}'}{\phi \models \delta(\vec{i}) \equiv \delta(\vec{i}')} \\ \frac{\phi \models \tau_1 \equiv \tau'_1 \quad \phi \models \tau_2 \equiv \tau'_2}{\phi \models \tau_1 * \tau_2 \equiv \tau'_1 * \tau'_2} \quad \frac{\phi \models \tau'_1 \equiv \tau_1 \quad \phi \models \tau_2 \equiv \tau'_2}{\phi \models \tau_1 \rightarrow \tau_2 \equiv \tau'_1 \rightarrow \tau'_2} \\ \frac{\phi, \vec{a} : \vec{\gamma} \models \tau \equiv \tau'}{\phi \models \Pi \vec{a} : \vec{\gamma}. \tau \equiv \Pi \vec{a} : \vec{\gamma}. \tau'} \quad \frac{\phi, a : \gamma \models \tau \equiv \tau'}{\phi \models \Sigma a : \gamma. \tau \equiv \Sigma a : \gamma. \tau'} \end{array}$$

We say that  $\delta$  has the kind  $\vec{\gamma} \rightarrow *$  if  $\delta$  takes index expressions  $\vec{i}$  of sorts  $\vec{\gamma}$  to form a type. There are currently no formal judgments for sort equivalence or subsorting. They are not really needed at this point, since ultimately all sorts only describe various subsets of integers. Some details can be found in (Xi 1998) as to how an expression of the type  $\Pi a : \gamma. \tau$  can be implicitly coerced into one with equivalent dynamic semantics, but of the type  $\Pi a : \gamma'. \tau$ , where  $\gamma$  and  $\gamma'$  are sorts such that  $a : \gamma' \vdash a : \gamma$ . As could be expected, we have the following proposition.

**Proposition 2.2** *Type conversion is an equivalence relation:*

1. *If  $\phi \vdash \tau$  [well-formed] holds, then  $\phi \models \tau \equiv \tau$  also holds.*
2. *If  $\phi \models \tau \equiv \tau'$  holds, then  $\phi \vdash \tau' \equiv \tau$  also holds.*
3. *If both  $\phi \models \tau \equiv \tau'$  and  $\phi \models \tau' \equiv \tau''$  hold, then  $\phi \models \tau \equiv \tau''$  also holds.*

We present the typing rules for  $\text{ML}_0^{\Pi, \Sigma}$  in Figure 3 and Figure 4. We use  $\phi; \Gamma \vdash e : \tau$  for a typing judgment. The judgment basically means that  $e$  can be assigned type  $\tau$  under  $\phi; \Gamma$ , which map free index variables and variables in  $e$  to sorts and types, respectively.

Let  $\vec{a}$  be a sequence of index variables  $a_1, \dots, a_n$  and  $\vec{i}$  be a sequence of index expressions  $i_1, \dots, i_n$ ; we write  $[\vec{a} \mapsto \vec{i}]$  for a substitution that maps  $a_k$  to  $i_k$  for  $k = 1, \dots, n$ ; given an index variable context  $\vec{a} : \vec{\gamma}$ , that is,  $a_1 : \gamma_1, \dots, a_n : \gamma_n$  for  $\vec{\gamma} = (\gamma_1, \dots, \gamma_n)$ , we write  $\phi \vdash \vec{i} : \vec{\gamma}$  to mean that  $\phi \vdash [\vec{a} \mapsto \vec{i}] : (\vec{a} : \vec{\gamma})$ . Notice that  $\phi \vdash \vec{i} : \vec{\gamma}$  does not simply mean  $\phi \vdash i_k : \gamma_k$  for  $k = 1, \dots, n$  as  $a_k$  may have a occurrence in  $\gamma_{k'}$  for  $1 \leq k < k' \leq n$ . Some of the typing rules have obvious side conditions, which are omitted. For instance, in the rule **(ty-ilam)**,  $\vec{a}$  cannot have free occurrences in  $\Gamma$ . We write  $\text{dom}(\Gamma)$  for the domain of  $\Gamma$ , that is, the set of variables declared in  $\Gamma$ . Given substitutions  $\theta_I$  and  $\theta$ , we say  $\phi; \Gamma \vdash (\theta_I; \theta) : (\phi'; \Gamma')$  holds if  $\phi \vdash \theta_I : \phi'$  and  $\text{dom}(\theta) = \text{dom}(\Gamma')$  and  $\phi; \Gamma[\theta_I] \vdash \theta(x) : \Gamma'(x)[\theta_I]$  for all  $x \in \text{dom}(\Gamma')$ .

The following lemma plays a pivotal rôle in proving the subject reduction theorem for  $\text{ML}_0^{\Pi, \Sigma}$ .

**Lemma 2.3** *Assume that  $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau$  is derivable and  $\phi; \Gamma \vdash (\theta_I; \theta) : (\phi'; \Gamma')$  holds. Then we can derive  $\phi; \Gamma[\theta_I] \vdash e[\theta_I][\theta] : \tau[\theta_I]$ .*

*Proof* Please see (Xi 1998). ■

$$\begin{array}{c}
\frac{}{x \downarrow \tau \Rightarrow (\cdot; x : \tau)} \text{ (pat-var)} \qquad \frac{}{\langle \rangle \downarrow \mathbf{1} \Rightarrow (\cdot; \cdot)} \text{ (pat-unit)} \\
\frac{p_1 \downarrow \tau_1 \Rightarrow (\phi_1; \Gamma_1) \quad p_2 \downarrow \tau_2 \Rightarrow (\phi_2; \Gamma_2)}{\langle p_1, p_2 \rangle \downarrow \tau_1 * \tau_2 \Rightarrow (\phi_1, \phi_2; \Gamma_1, \Gamma_2)} \text{ (pat-prod)} \\
\frac{S(c) = \Pi \vec{a} : \vec{\gamma}. (\tau \rightarrow \delta(i)) \quad p \downarrow \tau \Rightarrow (\phi; \Gamma)}{c[\vec{a}](p) \downarrow \delta(j) \Rightarrow (\vec{a} : \vec{\gamma}, \phi; \Gamma)} \text{ (pat-cons)}
\end{array}$$

Figure 3: Typing rules for patterns

### 2.3 Dynamic Semantics

We present the dynamic semantics of  $\text{ML}_0^{\Pi, \Sigma}$  through the use of evaluation contexts, which are defined below.

$$\begin{array}{l}
\text{evaluation contexts } E ::= \\
\boxed{\phantom{e}} \mid \text{if}(E, e_1, e_2) \mid \text{fst}(E) \mid \text{snd}(E) \mid \langle E, e \rangle \mid \langle v, E \rangle \mid E[\vec{i}] \mid E(e) \mid v(E) \mid \\
\text{case } E \text{ of } ms \mid \text{let } x = E \text{ in } e \text{ end} \mid \langle i \mid E \rangle \mid \text{open } E \text{ as } \langle a \mid x \rangle \text{ in } e
\end{array}$$

We write  $E[e]$  for the expression resulting from replacing the hole  $\boxed{\phantom{e}}$  in  $E$  with  $e$ . Note that this replacement can *never* result in capturing free variables.

Given a pattern  $p$  and a value  $v$ , a judgment of the form  $\text{match}(v, p) \Rightarrow (\theta_I; \theta)$ , which means that matching a value  $v$  against a pattern  $p$  yields an index substitution for the index variables in  $p$  and a value substitution for the variables in  $p$ , can be derived with the application of the rules in Figure 5.

**Definition 2.4** *A redex is defined as follows.*

- $\text{if}(true, e_1, e_2)$  and  $\text{if}(false, e_1, e_2)$  are redexes, which reduce to  $e_1$  and  $e_2$ , respectively.
- $\text{fst}(\langle v_1, v_2 \rangle)$  is a redex, which reduces to  $v_1$ .
- $\text{snd}(\langle v_1, v_2 \rangle)$  is a redex, which reduces to  $v_2$ .
- $(\text{lam } x : \tau.e)(v)$  is a redex, which reduces to  $e[x := v]$ .
- $\text{let } x = v \text{ in } e \text{ end}$  is a redex, which reduces to  $e[x := v]$ .
- Let  $e$  be  $\text{fix } f : \tau.v$ . Then  $e$  is a redex, which reduces to  $v[f := e]$ .
- $\text{case } v \text{ of } (p_1 \Rightarrow e_1 \mid \dots \mid p_n \Rightarrow e_n)$  is a redex if  $\text{match}(v, p_k) \Rightarrow (\theta_I; \theta)$  is derivable for some  $1 \leq k \leq n$ , and it reduces to  $e_k[\theta_I][\theta]$ .
- $(\lambda \vec{a} : \vec{\gamma}.v)[\vec{i}]$  is a redex, which reduces to  $v[\vec{a} := \vec{i}]$ .
- $\text{open } \langle i \mid v \rangle \text{ as } \langle a \mid x \rangle \text{ in } e$  is a redex, which reduces to  $e[a := i][x := v]$ .

We use  $r$  for a redex and write  $r \hookrightarrow e$  if  $r$  reduces to  $e$ . If  $e_1 = E[r]$ ,  $e_2 = E[e]$  and  $r \hookrightarrow e$ , we also write  $e_1 \hookrightarrow e_2$  and say  $e_1$  reduces to  $e_2$  in one step.

Let  $\hookrightarrow^*$  be the reflexive and transitive closure of  $\hookrightarrow$ . We say  $e_1$  reduces to  $e_2$  (in many steps) if  $e_1 \hookrightarrow^* e_2$ . We are now ready to establish the the following subject reduction theorem for  $\text{ML}_0^{\Pi, \Sigma}$ .

**Theorem 2.5** (Subject Reduction) *Assume  $\cdot; \vdash e : \tau$  is derivable in  $\text{ML}_0^{\Pi, \Sigma}$ , that is,  $e$  is a well-typed closed expression in  $\text{ML}_0^{\Pi, \Sigma}$ . If  $e \hookrightarrow e'$ , then  $\cdot; \vdash e' : \tau$  is also derivable in  $\text{ML}_0^{\Pi, \Sigma}$ .*

*Proof* Please see (Xi 2002) for a thorough proof of the theorem. ■

$$\begin{array}{c}
\frac{\phi; \Gamma \vdash e : \tau_1 \quad \phi \models \tau_1 \equiv \tau_2}{\phi; \Gamma \vdash e : \tau_2} \text{ (ty-eq)} \\
\frac{\Sigma(c) = \Pi \vec{a} : \vec{\gamma}. \tau_1 \rightarrow \tau_2 \quad \phi \vdash \vec{i} : \vec{\gamma} \quad \phi; \Gamma \vdash e : \tau_1[\vec{a} := \vec{i}]}{\phi; \Gamma \vdash c : \tau_2[\vec{a} := \vec{i}]} \text{ (ty-constructor)} \\
\frac{\Gamma(x) = \tau}{\phi; \Gamma \vdash x : \tau} \text{ (ty-lam-var)} \quad \frac{\Gamma(f) = \tau}{\phi; \Gamma \vdash f : \tau} \text{ (ty-fix-var)} \\
\frac{\phi, \vec{a} : \vec{\gamma}; \Gamma \vdash v : \tau}{\phi; \Gamma \vdash \lambda \vec{a} : \vec{\gamma}. v : \Pi \vec{a} : \vec{\gamma}. \tau} \text{ (ty-ilam)} \\
\frac{\phi; \Gamma \vdash e : \Pi \vec{a} : \vec{\gamma}. \tau \quad \phi \vdash \vec{i} : \vec{\gamma}}{\phi; \Gamma \vdash e[\vec{i}] : \tau[\vec{a} := \vec{i}]} \text{ (ty-iapp)} \\
\frac{\phi; \Gamma \vdash e : \text{bool}(i) \quad \phi, i = 1; \Gamma \vdash e_1 : \tau \quad \phi, i = 0; \Gamma \vdash e_2 : \tau}{\phi; \Gamma \vdash \text{if}(e, e_1, e_2) : \tau} \text{ (ty-if)} \\
\frac{}{\phi; \Gamma \vdash \langle \rangle : \mathbf{1}} \text{ (ty-unit)} \quad \frac{\phi; \Gamma \vdash e_1 : \tau_1 \quad \phi; \Gamma \vdash e_2 : \tau_2}{\phi; \Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 * \tau_2} \text{ (ty-tuple)} \\
\frac{\phi; \Gamma \vdash e : \tau_1 * \tau_2}{\phi; \Gamma \vdash \mathbf{fst}(e) : \tau_1} \text{ (ty-fst)} \quad \frac{\phi; \Gamma \vdash e : \tau_1 * \tau_2}{\phi; \Gamma \vdash \mathbf{snd}(e) : \tau_2} \text{ (ty-snd)} \\
\frac{\phi; \Gamma, x : \tau_1 \vdash e : \tau_2}{\phi; \Gamma \vdash \mathbf{lam} x : \tau_1. e : \tau_1 \rightarrow \tau_2} \text{ (ty-lam)} \\
\frac{\phi; \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \phi; \Gamma \vdash e_2 : \tau_1}{\phi; \Gamma \vdash e_1(e_2) : \tau_2} \text{ (ty-app)} \\
\frac{p \downarrow \tau_1 \Rightarrow (\phi_1, \Gamma_1) \quad \phi, \phi_1; \Gamma, \Gamma_1 \vdash e : \tau_2}{\phi; \Gamma \vdash p \Rightarrow e : \tau_1 \Rightarrow \tau_2} \text{ (ty-clause)} \\
\frac{\phi; \Gamma \vdash p_k \Rightarrow e_k : \tau_1 \Rightarrow \tau_2 \text{ for } k = 1, \dots, n}{\phi; \Gamma \vdash (p_1 \Rightarrow e_1 \mid \dots \mid p_n \Rightarrow e_n) : \tau_1 \Rightarrow \tau_2} \text{ (ty-clauses)} \\
\frac{\phi; \Gamma \vdash e : \tau_1 \quad \phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\phi; \Gamma \vdash \mathbf{case} e \text{ of } ms : \tau_2} \text{ (ty-case)} \\
\frac{\phi; \Gamma \vdash e_1 : \tau_1 \quad \phi; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi; \Gamma \vdash \mathbf{let} x = e_1 \text{ in } e_2 \text{ end} : \tau_2} \text{ (ty-let)} \\
\frac{\phi; \Gamma, f : \tau \vdash v : \tau}{\phi; \Gamma \vdash \mathbf{fix} f : \tau. v : \tau} \text{ (ty-fix)} \\
\frac{\phi; \Gamma \vdash e_1 : \Sigma a : \gamma. \tau_1 \quad \phi, a : \gamma; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi; \Gamma \vdash \mathbf{open} e_1 \text{ as } \langle a \mid x \rangle \text{ in } e_2 : \tau_2} \text{ (ty-open)} \\
\frac{\phi \vdash i : \gamma \quad \phi; \Gamma \vdash e : \tau[a := i]}{\phi; \Gamma \vdash \langle i \mid e \rangle : \Sigma a : \gamma. \tau} \text{ (ty-pack)}
\end{array}$$

Figure 4: Typing Rules for  $\text{ML}_0^{\Pi, \Sigma}$

$$\begin{array}{c}
\frac{}{\mathbf{match}(v, x) \Rightarrow (\[], [x \mapsto v])} \text{ (match-var)} \\
\frac{}{\mathbf{match}(\langle \rangle, \langle \rangle) \Rightarrow (\[], \[])} \text{ (match-unit)} \\
\frac{\mathbf{match}(v_1, p_1) \Rightarrow (\theta_1^1, \theta_1) \quad \mathbf{match}(v_2, p_2) \Rightarrow (\theta_2^2, \theta_2)}{\mathbf{match}(\langle v_1, v_2 \rangle, \langle p_1, p_2 \rangle) \Rightarrow (\theta_1^1 \cup \theta_2^2, \theta_1 \cup \theta_2)} \text{ (match-prod)} \\
\frac{\mathbf{match}(v, p) \Rightarrow (\theta_I; \theta)}{\mathbf{match}(c[\vec{v}](v), c[\vec{a}](p)) \Rightarrow ([\vec{a} \mapsto \vec{v}] \cup \theta_I; \theta)} \text{ (match-cons)}
\end{array}$$

Figure 5: The pattern matching rules for  $\text{ML}_0^{\Pi, \Sigma}(C)$

## 2.4 Erasure

We can simply transform  $\text{ML}_0^{\Pi, \Sigma}$  into a language  $\text{ML}_0$  by erasing all syntax related to index expressions in  $\text{ML}_0^{\Pi, \Sigma}$ . Then  $\text{ML}_0$  basically extends simply typed  $\lambda$ -calculus with recursion. Let  $|e|$  be the erasure of expression  $e$ .<sup>4</sup>

Given a well-typed closed expression  $e$  in  $\text{ML}_0^{\Pi, \Sigma}$ , it can be shown that if  $|e|$  reduces to  $e_0$  in  $\text{ML}_0$  then  $e$  reduces to some  $e_1$  such that  $|e_1| = e_0$ . For this, we need the observation that the erasure of a value in  $\text{ML}_0^{\Pi, \Sigma}$  is a value in  $\text{ML}_0$ , which can be readily verified. This is precisely the reason why we impose the requirement that the body of each  $\lambda$  be a value. Otherwise, the erasure of  $\lambda a : \gamma. e$ , which is  $|e|$ , may not necessarily be a value in  $\text{ML}_0$ . Therefore, the evaluation of a program in  $\text{ML}_0^{\Pi, \Sigma}$  can be done through the evaluation of its erasure in  $\text{ML}_0$ . The significance of erasure is that it allows a program in DML to be treated as one in ML.

## 2.5 Extensions

We can readily extend  $\text{ML}_0^{\Pi, \Sigma}$  with second-order polymorphism, which encompasses the let-polymorphism supported in DML.<sup>5</sup>

type variables	$\alpha$
types	$\tau ::= \dots \mid \alpha \mid \forall \alpha. \tau$
expressions	$e ::= \dots \mid \Lambda \alpha. v \mid e[\tau]$
values	$v ::= \dots \mid \Lambda \alpha. v$

A type judgment is now of the form  $\vec{\alpha}; \phi; \Gamma \vdash e : \tau$ , where  $\vec{\alpha}$  is for a (possibly empty) sequence of type variables  $\alpha_1, \dots, \alpha_n$ ; all free type variables in  $\Gamma$  and  $\tau$  are declared in  $\vec{\alpha}$ . It should be clear how to modify each previous typing rule to accommodate type variables. We present the typing rules for handling type abstraction and application as follows.

$$\begin{array}{c}
\frac{\vec{\alpha}, \alpha; \phi; \Gamma \vdash v : \tau}{\vec{\alpha}; \phi; \Gamma \vdash \Lambda \alpha. v : \forall \alpha. \tau} \text{ (ty-tlam)} \\
\frac{\vec{\alpha}; \phi \vdash \tau \text{ [well-formed]} \quad \vec{\alpha}; \phi; \Gamma \vdash e : \forall \alpha. \tau}{\vec{\alpha}; \phi \vdash e[\tau] : \tau[\alpha := \tau]} \text{ (ty-tapp)}
\end{array}$$

In a straightforward manner, we can also incorporate into  $\text{ML}_0^{\Pi, \Sigma}$  some features involving side effects such as references and exceptions. A chapter on this issue can be found in (Xi 1998).

<sup>4</sup>The erasure  $|\mathbf{open} \ e_1 \ \mathbf{as} \ \langle a \mid x \rangle \ \mathbf{in} \ e_2|$  is  $\mathbf{let} \ x = |e_1| \ \mathbf{in} \ |e_2| \ \mathbf{end}$ .

<sup>5</sup>We are currently extending DML to support second-order polymorphism.

### 3 An Overview of Some Features in DML

The concrete syntax of DML, in which the programmer writes programs, largely resembles that of Standard ML (Milner, Tofte, Harper, and MacQueen 1997). New syntax is to be explained when it occurs in program examples. In this section, we focus on presenting some unique and significant programming features in DML, preparing for the examples to be presented in Section 4. The current implementation of DML, written in Objective Caml, is available at (Xi 2001).

#### 3.1 Dependent Datatypes

The programmer often declares datatypes when programming in ML. For instance, the following syntax declares a type constructor *list* and associates with it two value constructors *nil* and *cons*.

```
datatype 'a list = nil | cons of 'a * 'a list
```

Given a type  $\tau$ , the type constructor *list* constructs the type  $(\tau)list$  for lists in which each element has the type  $\tau$ , and the value constructors *nil* and *cons* are assigned the following types:

$$\begin{aligned} nil &: \forall\alpha.(\alpha)list \\ cons &: \forall\alpha.\alpha * (\alpha)list \rightarrow (\alpha)list \end{aligned}$$

However, the declared type constructor *list* cannot capture the length information of a list. For instance, we cannot use a list type to distinguish an empty list from a non-empty one. In DML, this drawback can be remedied by declaring *list* as a dependent type constructor:

```
datatype 'a list with nat =
  nil(0) | {n:nat} cons(n+1) of 'a * 'a list(n)
```

The syntax `datatype 'a list with nat` means that *list* is a type constructor that takes a type  $\tau$  and a type index  $i$  of sort *nat* to construct the type  $(\tau)list(i)$  for lists with length  $i$  in which each element has the type  $\tau$ . In addition, we have the following:

- The syntax `nil(0)` means that *nil* is given the type  $\forall\alpha.(\alpha)list(0)$ . In other words, *nil* represents a list with length 0.
- The syntax `{n:nat} cons(n+1) of 'a * 'a list(n)` means that *cons* is given the type:

$$\forall\alpha\Pi n : nat.\alpha * (\alpha)list(n) \rightarrow (\alpha)list(n + 1)$$

That is, for every natural number  $n$ , *cons* yields a list of type  $(\tau)list(n + 1)$  when applied to an element of type  $\tau$  and a list of type  $(\tau)list(n)$ . Note the syntax `{n:nat}` corresponds to  $\Pi n : nat$  in type theory.

The list types have now become more informative. The following code defines the append function on lists, where we use `::` as the infix operator for *cons*.

```
fun('a)
  append ([], ys) = ys
  | append (x :: xs, ys) = x :: append(xs, ys)
withtype {m:nat, n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)
```

The `withtype` clause is a type annotation supplied by the programmer, which simply states that the function returns a list of length  $m + n$  when given a pair of lists of length  $m$  and  $n$ , respectively. We now present an informal description on how type-checking is performed in this case. Please observe the application of the typing rules (**ty-clause**) and the rules in Figure 3.

For the first clause  $append(nil, ys) = ys$ , the type-checker assumes that *ys* has the type  $(\alpha)list(b)$  for some index variable  $b$  of the sort *nat*. This implies that  $(nil, ys)$  has the type  $(\alpha)list(0) * (\alpha)list(b)$ . The type-checker then instantiates  $m$  and  $n$  with 0 and  $b$ , respectively, and verify that the *ys* on the right side

of  $=$  has the type  $(\alpha)list(0 + b)$ . Since  $ys$  is assumed to have the type  $(\alpha)list(b)$ , the type-checker generates a constraint  $b = 0 + b$  under the assumption that  $b$  is a natural number. This constraint can be easily verified.

Let us now type-check the second clause  $append(x :: xs, ys) = x :: append(xs, ys)$ . Assume that  $xs$  and  $ys$  have the types  $(\alpha)list(a)$  and  $(\alpha)list(b)$  respectively, where  $a$  and  $b$  are index variables of sort  $nat$ . Then  $(x :: xs, ys)$  has the type  $(\alpha)list(a + 1) * (\alpha)list(b)$ , and we therefore instantiate  $m$  and  $n$  with  $a + 1$  and  $b$ , respectively. Also we infer that the right side  $x :: append(xs, ys)$  has the type  $(\alpha)list((a + b) + 1)$  since  $xs$  and  $ys$  are assumed to have types  $(\alpha)list(a)$  and  $(\alpha)list(b)$ , respectively. We need to prove that the right side has the type  $(\alpha)list(m + n)$  for  $m = a + 1$  and  $n = b$ . This leads to the generation of the following constraint,

$$(a + 1) + b = (a + b) + 1$$

which can be immediately verified under that assumption that  $a$  and  $b$  are natural numbers. This finishes type-checking the above DML program. The interested reader is referred to (Xi 1998) for a formal presentation of type-checking in DML.

Clearly, a natural question is whether the type for the function  $append$  can be reconstructed or synthesized. For such a simple example, this seems highly possible. However, our experience indicates that it seems exceedingly difficult in general to synthesize dependent types in practice, though we have not formally studied this issue.

## 3.2 Existential dependent types

Existential dependent types, that is, types of the form  $\Sigma a : \gamma. \tau$ , are of great use in practice. For instance, an important application of existential dependent types is to cope with existing library functions. Suppose  $f$  is some existing function that is already defined before the type constructor  $list$  is indexed with a natural number and the type of  $f$  is  $\forall \alpha. (\alpha)list \rightarrow (\alpha)list$ . We can now assign  $f$  the type  $\forall \alpha. (\Sigma a : nat. (\alpha)list(a)) \rightarrow (\Sigma a : nat. (\alpha)list(a))$ , which means that  $f$  takes a list with unknown length and returns a list with unknown length. This makes it possible for  $f$  to be applied to an argument  $l$  of a dependent type, say,  $(int)list(2)$ , by coercing  $l$  into  $\langle 2 \mid l \rangle$ , which has the type  $\Sigma a : nat. (int)list(a)$ .

We now mention a convention in DML. After declaring a dependent type as follows,

$$\text{datatype } (\alpha_1, \dots, \alpha_m) \delta \text{ with } (sort_1, \dots, sort_n) = \dots\dots$$

we may write  $(\tau_1, \dots, \tau_m)\delta$  to stand for the following type:

$$\Sigma a_1 : sort_1 \dots \Sigma a_n : sort_n. (\tau_1, \dots, \tau_m)\delta(a_1, \dots, a_n)$$

For example,  $(\tau)list$  stands for  $\Sigma a : nat. (\tau)list(a)$  (assuming no free occurrences of  $a$  in  $\tau$ ).

There is another important application of existential dependent types in practice. In order to guarantee practical type checking in DML, we must make constraints relatively simple. Currently, we only accept linear integer constraints. This immediately implies that there are many (realistic) constraints that are inexpressible in the type system of DML. For instance, the following code implements a filter function on a list that removes from all the elements from the list that does not satisfy the predicate  $p$ :

```
fun('a) filter p xs: 'a list =
  case xs of
    [] => []
  | x :: xs => if p(x) then x :: filter p xs else filter p xs
```

In general, it is impossible to know the length of the list returned by  $filter p xs$  without knowing what  $p$  and  $xs$  are. Therefore, it is impossible to type the function using only universal dependent types. Nonetheless, we know that the length of  $filter p xs$  is less than or equal to that of  $xs$ . This invariant can be captured by assigning  $filter$  the following type,

```
('a -> bool) -> {m:nat} 'a list(m) -> [n:nat | n <= m] 'a list(n)
```

which is formally written as follows:

$$\forall \alpha. (\alpha \rightarrow \mathbf{bool}) \rightarrow \Pi m : \mathit{nat}. (\alpha) \mathit{list}(m) \rightarrow \Sigma n : \{a : \mathit{nat} \mid a \leq m\}. (\alpha) \mathit{list}(n)$$

The quantification  $\Sigma n : \{a : \mathit{nat} \mid a \leq m\}$  essentially acts like a post-condition for the function *filter*.

Another significant use of existential dependent types is to represent a range of values. For instance, we can form a reference type  $(\Sigma a : \{a : \mathit{int} \mid i_1 \leq a \leq i_2\}. \mathit{int}(a)) \mathbf{ref}$  and assign it to a variable, requiring that only integers between  $i_1$  and  $i_2$  be stored in the variable. This is particularly useful in imperative programming (Xi 2000). In general, we view that the use of existential types in DML for handling functions like *filter* is crucial to the scalability of the type system of DML, since such functions are ubiquitous in practical programming.

### 3.3 Elaboration

A process that translates a program in the concrete syntax of DML into an expression in the explicitly typed internal language of DML is what we call *elaboration*. For instance, in Figure 2, the first representation of the Ackermann function in DML can be elaborated into the second one in  $\text{ML}_0^{\Pi, \Sigma}$ .

Elaboration in DML is involved. An overview of elaboration is given in (Xi and Pfenning 1999), and further details can be found in (Xi 1998). In the rest of the paper, we assume the availability of an elaboration process. We have carefully written our examples so that they can be elaborated in a simple and intuitive manner. The way in which the above list append function is handled can be applied to all examples in Section 4.

## 4 Programming Examples in DML

In this section, we present several examples to demonstrate the use of dependent datatypes in capturing invariants in data structures. All these examples in DML have been successfully verified in a prototype implementation of DML written in Objective Caml. The claim we make is that dependent datatypes can enable the programmer to implement algorithms in a way that is more efficient, more robust and easier to understand.

### 4.1 Arrays

Arrays are a widely used data structure in practice. We use *array* as a built-in type constructor that takes a type  $\tau$  and a natural number  $n$  to form the type  $(\tau) \mathit{array}(n)$  for arrays of size  $n$  in which each element has the type  $\tau$ .<sup>6</sup> We also have the built-in functions *sub*, *update* and *make\_array*, which are given the following types:

$$\begin{aligned} \mathit{sub} & : \forall \alpha \Pi n : \mathit{nat} \Pi i : \{a : \mathit{nat} \mid a < n\}. (\alpha) \mathit{array}(n) * \mathit{int}(i) \rightarrow \alpha \\ \mathit{update} & : \forall \alpha \Pi n : \mathit{nat} \Pi i : \{a : \mathit{nat} \mid a < n\}. (\alpha) \mathit{array}(n) * \mathit{int}(i) * \alpha \rightarrow \mathbf{1} \\ \mathit{make\_array} & : \forall \alpha \Pi n : \mathit{nat}. \mathit{int}(n) * \alpha \rightarrow (\alpha) \mathit{array}(n) \end{aligned}$$

There is no built-in function for computing the size of an array. Notice that the type of *sub* indicates that the function can be applied to an array and an index only if the value of the index is a natural number less than the size of the array. In other words, the quantification  $\Pi n : \mathit{nat} \Pi i : \{a : \mathit{nat} \mid a < n\}$  acts like a pre-condition for the function *sub*. The type of *update* imposes a similar requirement. We may, however, encounter a situation where the programmer knows or believes for some reason that the value of the index is within the bounds of the array, but this property is difficult or even impossible to be captured in the type system of DML. In such a situation, the programmer may need to use run-time array bound checks to overcome the difficulty. We now present a type-theoretical explanation on array bound checking in DML.

In Figure 6, we declare a type constructor *Array* for forming types for arrays with size information. The only value constructor *Array* associated with the type constructor<sup>7</sup> is assigned the following type:

$$\forall \alpha \Pi n : \mathit{nat}. \mathit{int}(n) * (\alpha) \mathit{array}(n) \rightarrow (\alpha) \mathit{Array}(n)$$

<sup>6</sup>Each valid index of an array is a natural number less than the size of the array

<sup>7</sup>Yes, we overload the name *Array* for both type and value constructors

```

datatype 'a Array with nat = {n:nat} Array(n) of int(n) * 'a array(n)

exception Subscript

fun('a) Sub (Array(n, a), i) =
  if (i < 0) then raise Subscript
  else if (i >= n) then raise Subscript
  else sub (a, i)
withtype {n:nat,i:int} 'a Array(n) * int(i) -> 'a

fun('a) Update (Array(n, a), i, x) =
  if (i < 0) then raise Subscript
  else if (i >= n) then raise Subscript
  else update (a, i, x)
withtype {n:nat,i:int} 'a Array(n) * int(i) * 'a -> unit

fun('a) Make_Array (n, x) = Array (n, make_array (n, x))
withtype {n:nat} int(n) * 'a -> 'a Array(n)

fun('a) Sizeof (Array(n, _)) = n
withtype {n:nat} 'a Array(n) -> int(n)

```

Figure 6: A datatype for arrays with size information and some related functions

The defined functions *Sub*, *Update* and *Make\_Array* correspond to the functions *sub*, *update* and *make\_array*, respectively. Note that run-time array bound checks are inserted in the implementation of *Sub* and *Update*. For an array carrying size information, the function *Sizeof* simply extracts out the information.

Clearly, the programmer now has the option to choose which subscripting (updating) function should be used: *Sub* or *sub* (*Update* or *update*)? Since the former is less efficient and may incur a run-time exception, the latter is certainly preferred. However, in order to use the latter, the programmer often needs to capture more program invariants by supplying type annotations. This point is clearly illustrated when we compare the two implementations of the usual binary search on integer arrays in Figure 7. In the first implementation, we use the array subscripting function *Sub*, which incurs run-time array bound checks. In the second implementation, we use *sub*, which incurs no run-time array bound checks. Clearly, the second implementation is superior to the first one when either safety or efficiency is concerned. However, the programmer needs to provide a more informative type for the inner function *loop* in order to eliminate the array bound checks. In this case, the provided type captures the invariant that  $i \leq j + 1 \leq n$  holds whenever *loop*(*l*, *u*) is called, where *i* and *j* are values of *l* and *u*, and *n* is the size of the array *Vec*.

It is often argued that it can be a significant burden for the programmer to write type annotations when programming in DML. However, this argument is largely misleading. Type-checking in DML currently consists of two phases; in the first phase, the type-checker simply ignores type indexes and does ML-like type-checking; only in the second phase, type constraints are generated and solved. Hence, if the programmer uses few dependent types, then few type annotations are required. The above example clearly shows that the programmer can certainly choose not to write the type annotation; instead, he or she can employ run-time array bound checks to preclude out-of-bounds array access. Unfortunately, this strategy, though better than no checks at all, does not really enhance the robustness of the implementation since there is simply no sensible recovery code to execute when a run-time array bound check fails. In this respect, DML is superior to ML as it allows the programmer to prove within the type system of DML that a given array subscripting operation can never go out of bounds.

```

fun binary_search_1 key Vec = let
  fun loop (l, u) =
    if u < l then NONE
    else let
      val m = 1 + (u-1) / 2
      val x = Sub (Vec, m)
    in
      if x < key then loop (m+1, u)
      else if x > key then loop (l, m-1) else SOME (m)
    end
  (* the following type annotation can be inferred *)
  withtype int * int -> int option
in loop (0, Sizeof Vec - 1) end
withtype int -> {n:nat} int Array(n) -> int option

fun binary_search_2 key Vec = let
  val Array (n, vec) = Vec
  fun loop (l, u) =
    if u < l then NONE
    else let
      val m = 1 + (u-1) / 2
      val x = sub (vec, m)
    in
      if x < key then loop (m+1, u)
      else if x > key then loop (l, m-1) else SOME (m)
    end
  withtype
    {i:int,j:int | 0 <= i <= j+1 <= n} int(i) * int(j) -> int option
in loop (0, n-1) end
withtype int -> {n:nat} int Array(n) -> int option

```

Figure 7: An implementation of binary search on integer arrays in DML

## 4.2 Random-Access Lists

A random-access list is a list representation such that list lookup (update) can be implemented in an efficient way. In this case, the lookup (update) function takes  $O(\log n)$  time in contrast to the usual  $O(n)$  time (on average), where  $n$  is the length of the input list.

### 4.2.1 The First Implementation

We present an implementation of random-access lists in Figures 8. We first declare the dependent datatype for random-access lists. Note that  $(\tau)ralist(n)$  stands for the type of random-access lists with length  $n$  in which each element has type  $\tau$ . The value constructors *Nil* and *One* are for constructing empty and singleton random-access lists, respectively. Furthermore, the value constructors *Even* and *Odd* are used to form random-access lists of even and odd lengths, respectively. If  $l_1$  and  $l_2$  represent lists  $x_1, \dots, x_n$  and  $y_1, \dots, y_n$  for some  $n > 0$ , respectively, then *Even*( $l_1, l_2$ ) represents the list  $x_1, y_1, \dots, x_n, y_n$ . Similarly, if  $l_1$  and  $l_2$  represent lists  $x_1, \dots, x_n, x_{n+1}$  and  $y_1, \dots, y_n$  for some  $n > 0$ , respectively, *Odd*( $l_1, l_2$ ) represents the list  $x_1, y_1, \dots, x_n, y_n, x_{n+1}$ . With such a data structure, we can implement a lookup (update) function on random-access list which takes  $O(\log n)$  time. A crucial invariant on this data structure is that  $l_1$  and  $l_2$  must have the same length if *Even*( $l_1, l_2$ ) is formed or  $l_1$  contains one more element than  $l_2$  if *Odd*( $l_1, l_2$ ) is formed. This is clearly captured by the dependent datatype declaration for *ralist*. The function *cons* appends an element to a list and *uncons* decomposes a list into a pair consisting of the head and the tail of the list. Note that the type of *uncons* requires that this function only be applied to a non-empty list. Both *cons* and *uncons* take  $O(\log n)$  time.

The function *lookup\_safe* deserves some explanation. The type of this function indicates that it can be applied to  $i$  and  $l$  only if  $i$  is a natural number and its value is less than the length of  $l$ . Notice that the *lookup\_safe*( $i, l$ ) simply return  $x$  when  $l$  matches the pattern *One*( $x$ ). There is no need to check whether  $i$  equals 0: It must since  $i$  is a natural number and  $i$  is less than the length of  $l$ , which is 1 in this case. The usual lookup function can be implemented as usual or as follows.

```
fun('a) lookup (i, l) =
  if i < 0 then raise Subscript
  else if i >= length l then raise Subscript
  else lookup_safe (i, l)
withtype int * 'a ralist -> 'a
```

### 4.2.2 The Second Implementation

There is an implementation of random-access lists in (Okasaki 1998), which uses the feature of nested datatypes. Okasaki's implementation supports (on average)  $O(1)$ -time consing and unconsing operations and thus are superior to the above implementation in this respect. We now present such an implementation, showing that dependent datatypes can be readily combined with nested datatypes.

We use the following syntax to declare a *nested dependent datatype* for random-access lists. For each natural number  $n$ ,  $(\tau)ralist(n)$  is the type for random-access lists with length  $n$  in which each element has the type  $\tau$ .

```
datatype 'a ralist with nat =
  Nil(0) | One(1) of 'a
  | {n:nat | n > 0} Even(n+n) of ('a * 'a) ralist(n)
  | {n:nat | n > 0} Odd(n+n+1) of 'a * ('a * 'a) ralist(n)
```

For example, the following two expressions represent lists 1, 2, 3, 4, 5 and 0, 1, 2, 3, 4, 5, respectively.

`Odd (1, Even (One ((2, 3), (4, 5))))` and `Even (Odd ((0,1), (One ((2,3), (4,5)))))`

Please refer to (Okasaki 1998) for an explanation on such a list representation.

We present in Figure 9 an implementation of random-access lists that uses the above nested dependent datatype. Note that we need polymorphic recursion for type-checking the code, which is supported in

```

datatype 'a ralist with nat =
  Nil(0) | One(1) of 'a
  | {n:nat | n > 0} Even(n+n) of 'a ralist(n) * 'a ralist(n)
  | {n:nat | n > 0} Odd(n+n+1) of 'a ralist(n+1) * 'a ralist(n)

fun('a)
  cons (x, Nil) = One x
  | cons (x, One y) = Even(One(x), One(y))
  | cons (x, Even(l1, l2)) = Odd(cons (x, l2), l1)
  | cons (x, Odd(l1, l2)) = Even(cons (x, l2), l1)
withtype {n:nat} 'a * 'a ralist(n) -> 'a ralist(n+1)

fun('a)
  uncons (One x) = (x, Nil)
  | uncons (Even(l1, l2)) =
    let
      val (x, l1) = uncons l1
    in
      case l1 of Nil => (x, l2) | _ => (x, Odd(l2, l1))
    end
  | uncons (Odd(l1, l2)) = let val (x, l1) = uncons l1 in (x, Even(l2, l1)) end
withtype {n:nat | n > 0} 'a ralist(n) -> 'a * 'a ralist(n-1)

fun('a)
  length (Nil) = 0
  | length (One _) = 1
  | length (Even (l1, _)) = 2 * (length l1)
  | length (Odd (_, l2)) = 2 * (length l2) + 1
withtype {n:nat} 'a ralist(n) -> int(n)

fun('a)
  lookup_safe (i, One x) = x
  | lookup_safe (i, Even(l1, l2)) =
    if i % 2 = 0 then lookup_safe (i / 2, l1) else lookup_safe (i / 2, l2)
  | lookup_safe (i, Odd(l1, l2)) =
    if i % 2 = 0 then lookup_safe (i / 2, l1) else lookup_safe (i / 2, l2)
withtype {i:nat, n:nat | i < n} int(i) * 'a ralist(n) -> 'a

fun('a)
  update_safe (i, x, One y) = One x
  | update_safe (i, x, Even(l1, l2)) =
    if i % 2 = 0 then Even(update_safe (i / 2, x, l1), l2)
    else Even(l1, update_safe (i / 2, x, l2))
  | update_safe (i, x, Odd(l1, l2)) =
    if i % 2 = 0 then Odd(update_safe (i / 2, x, l1), l2)
    else Odd(l1, update_safe (i / 2, x, l2))
withtype {i:nat, n:nat | i < n} int(i) * 'a * 'a ralist(n) -> 'a ralist(n)

```

Figure 8: An implementation of random-access lists in DML

```

fun('a)
  cons (x, Nil) = One x
| cons (x, One y) = Even(One (x, y))
| cons (x, Even(l)) = Odd(x, l)
| cons (x, Odd(y, l)) = Even(cons ((x, y), l))
withtype {n:nat} 'a * 'a ralist(n) -> 'a ralist(n+1)

fun('a)
  uncons (One x) = (x, Nil)
| uncons (Even l) =
  let
    val ((x, y), l) = uncons (l)
  in
    case l of Nil => (x, One y) | _ => (x, Odd (y, l))
  end
| uncons (Odd (x, l)) = (x, Even l)
withtype {n:nat | n > 0} 'a ralist(n) -> 'a * 'a ralist(n-1)

fun('a) lookup_safe (i, l) =
  case l of
    One (x) => x
  | Odd (x, l) => if i = 0 then x else lookup_safe (i-1, Even l)
  | Even l =>
    let
      val (x, y) = lookup_safe (i / 2, l)
    in
      if i % 2 = 0 then x else y
    end
withtype {i:nat, n:nat | i < n} int(i) * 'a ralist(n) -> 'a

fun('a) fupdate_safe (f, i, l) =
  case l of
    One (x) => One (f x)
  | Odd (x, l) =>
    if i = 0 then Odd (f x, l) else cons (x, fupdate_safe (f, i-1, Even l))
  | Even l =>
    let
      fun f' (x, y) = if i % 2 = 0 then (f x, y) else (x, f y)
      withtype 'a * 'a -> 'a * 'a
    in
      Even (fupdate_safe (f', i / 2, l))
    end
withtype {i:nat, n:nat | i < n} ('a -> 'a) * int(i) * 'a ralist(n) -> 'a ralist(n)

fun('a) update_safe (i, l) = fupdate_safe (fn x => x, i, l)
withtype {i:nat, n:nat | i < n} int(i) * 'a ralist(n) -> 'a ralist(n)

```

Figure 9: Another implementation of random-access lists in DML

DML. Obviously, this implementation captures many more program invariants when compared with the one in (Okasaki 1998). For instance, the type of the function *uncons* indicates that the function can only be applied to a random-access list with positive length, and the length of its output equals that of its input minus 1. Therefore, the expression *uncons(uncons(One(0)))* leads to a type error since the type of the expression *uncons(One(0))* is *(int)ralist(0)*. However, such a type error cannot be captured by using only nested datatypes.

### 4.3 Red-Black Trees

A red-black tree (RBT) is a balanced binary tree that satisfies the following conditions: (a) all leaves are marked black and all other nodes are marked either red or black; (b) for every node there are the same number of black nodes on every path connecting the node to a leaf, and this number is called the *black height* of the node; (c) the two sons of every red node must be black. It is a common practice to use the RBT data structure for implementing a dictionary. We declare a datatype in Figure 10, which precisely captures these properties of being a RBT.

A sort *color* is declared for the type index expressions representing the colors of nodes. We use 0 for black and 1 for red. For simplicity, we use integers for keys. The type *rbtree* is indexed with a triple  $(c, bh, v)$ , where  $c, bh, v$  stand for the color of the node, the black height of the tree, and the number of color violations, respectively. We record one color violation if a red node is followed by another red one, and thus a RBT must have no color violations. Clearly, the types of value constructors associated with the type constructor *rbtree* indicate that color violations can only occur at the top node. Also, notice that a leaf, that is, *E*, is considered black. Given the datatype declaration and the explanation, it should be clear that the type of a RBT is simply  $\Sigma c : color. \Sigma bh : nat. rbtree(c, bh, 0)$ , that is, a RBT is a tree that has some top node color  $c$  and some black height  $bh$  but no color violations.

It is an involved task to implement RBT. The implementation we present is basically adopted from the one in (Okasaki 1998), though there are some minor modifications. We explain how the insertion operation on a RBT is implemented. Clearly, the invariant we intend to capture is that inserting an entry into a RBT yields another RBT. In other words, we intend to declare that the insertion operation has the following type:

$$key * (\Sigma c : color. \Sigma bh : nat. rbtree(c, bh, 0)) \rightarrow \Sigma c : color. \Sigma bh : nat. rbtree(c, bh, 0)$$

If we insert an entry into a RBT, some properties on RBT may be violated. These properties can be restored through some rotation operations. The function *restore* in Figure 10 is defined for this purpose.

The type of *restore*, though long, is easy to understand. It states that this function takes a tree with at most one color violation, an entry and a RBT, and returns a RBT. The two trees in the argument must have the same black height  $bh$  for some natural number  $bh$  and the black height of the returned RBT is  $bh + 1$ . This information can be of great help for understanding the code. It is not trivial at all to verify the information manually, and we could imagine that almost everyone who did this would appreciate the availability of a type-checker to perform it automatically.

There is a great difference between type-checking a pattern matching clause in DML and in ML. The operational semantics of ML requires that pattern matching be performed sequentially, that is, the chosen pattern matching clause is always the first one that matches a given value. For instance, in the definition of the function *restore*, if the last clause is chosen at run-time, then we know the argument of *restore* does not match either of the clauses ahead of the last one. This must be taken into account when we type-check pattern matching in DML. One approach is to expand patterns into disjoint ones. For instance, the pattern  $(a, x, b)$  expands into 36 patterns  $(pattern_1, x, pattern_2)$ , where  $pattern_1$  and  $pattern_2$  range over the following six patterns:

$$R(B-, -, B-), R(B-, -, E), R(E, -, B-), R(E, -, E), B-, E$$

Unfortunately, such an expansion may lead to combinatorial explosion. An alternative is to require the programmer to indicate whether such an expansion is needed. Neither of these is available in the current implementation of DML, and the author has taken the inconvenience to expand patterns into disjoint ones when necessary. We emphasize that the code in Figure 10 must be thus expanded in order to pass

```

type key = int

sort color = {a:int | 0 <= a <= 1}

datatype rbtree with (color, nat, nat) =
  E(0, 0, 0)
  | {cl:color, cr:color, bh:nat}
    B(0, bh+1, 0) of rbtree(cl, bh, 0) * key * rbtree(cr, bh, 0)
  | {cl:color, cr:color, bh:nat}
    R(1, bh, cl+cr) of rbtree(cl, bh, 0) * key * rbtree(cr, bh, 0)

fun restore (R(R(a, x, b), y, c), z, d) = R(B(a, x, b), y, B(c, z, d))
  | restore (R(a, x, R(b, y, c)), z, d) = R(B(a, x, b), y, B(c, z, d))
  | restore (a, x, R(R(b, y, c), z, d)) = R(B(a, x, b), y, B(c, z, d))
  | restore (a, x, R(b, y, R(c, z, d))) = R(B(a, x, b), y, B(c, z, d))
  | restore (a, x, b) = B(a, x, b)
withtype {cl:color, cr:color, bh:nat, vl:nat, vr:nat | vl+vr <= 1}
  rbtree(cl, bh, vl) * key * rbtree(cr, bh, vr) ->
  [c:color] rbtree(c, bh+1, 0)

exception Item_already_exists

fun insert (x, t) =
  let
    fun ins (E) = R(E, x, E)
      | ins (B(a, y, b)) =
        if x < y then restore(ins a, y, b)
        else if y < x then restore(a, y, ins b) else raise Item_already_exists
      | ins (R(a, y, b)) =
        if x < y then R(ins a, y, b)
        else if y < x then R(a, y, ins b) else raise Item_already_exists
    withtype {c:color, bh:nat}
      rbtree(c, bh, 0) -> [c':color][v:nat | v <= c] rbtree(c', bh, v)
  in
    case ins t of
      R(a, y, b) => B(a, y, b)
    | t => t
  end
withtype {c:color, bh:nat} key * rbtree(c, bh, 0) -> [bh':nat] rbtree(0, bh', 0)

```

Figure 10: A red-black tree implementation

```

datatype HOAS with ty =
  Int(I) of int
| Bool(B) of bool
| Add(I) of HOAS(I) * HOAS(I)
| Sub(I) of HOAS(I) * HOAS(I)
| Mul(I) of HOAS(I) * HOAS(I)
| Div(I) of HOAS(I) * HOAS(I)
| IsZero(B) of HOAS(I)
| {t: ty} If(t) of HOAS(B) * HOAS(t) * HOAS(t)
| {t1: ty, t2: ty} Lam(Arrow(t1,t2)) of HOAS(t1) -> HOAS(t2)
| {t1: ty, t2: ty} App(t2) of HOAS(Arrow(t1, t2)) * HOAS(t1)
| {t1: ty, t2: ty} Let(t2) of HOAS(t1) * (HOAS(t1) -> HOAS(t2))
| {t: ty} Fix(t) of HOAS(t) -> HOAS(t)

```

Figure 11: A datatype for higher-order abstract syntax

type-checking in DML. Though this can be fixed straightforwardly, it is currently unclear what method can solve the problem best.

The complete implementation of the insertion operation follows immediately. Notice the type of function *ins* indicates that *ins* may return a tree with one color violation if it is applied to a tree with a red top node. This can be handled by replacing the top node with a black one for every returned tree with a red top node.

Moreover, we can use an extra index to capture the size information of a RBT. If we do so, we can then show that the *insert* function always returns a RBT of size  $n + 1$  when given a RBT of size  $n$  (note that an exception is raised if the entry to be inserted already exists in the tree).

#### 4.4 A Type-Preserving Evaluator

In this section, we implement an evaluator for an object language based on simply typed  $\lambda$ -calculus, capturing in the type system of DML that the evaluator is type-preserving at the object level. This is the only example in this paper where we need a constraint domain consisting of first-order algebraic terms instead of integers.

We use the following syntax to define a sort *ty* for representing simple types in the object language. For instance, the term  $Arrow(I, Arrow(I, B))$  represents the type  $\text{int} \rightarrow (\text{int} \rightarrow \text{bool})$  in the object language.

```
sort ty = B | I | Arrow of (ty, ty)
```

We use higher-order abstract syntax (Church 1940; Pfenning ) to represent expressions in the object language. In Figure 11, we declare a type constructor *HOAS*, which takes an index expression  $t$  of sort *ty* and forms a type  $HOAS(t)$  for the closed expressions in the object language that have the type represented by  $t$ . For example, the function  $\lambda x : \text{int}. x + x$  in the object language is represented as  $Lam(fn\ x \Rightarrow Add(x, x))$ , which has the type  $HOAS(Arrow(I, I))$ ; the usual factorial function can be represented as follows (in the concrete syntax of DML), which also has the type  $HOAS(Arrow(I, I))$ .

```

Fix (fn f =>
  Lam (fn x =>
    If (IsZero (x), Int(1), Mul (x, App (f, Sub (x, Int(1)))))))

```

We now implement a function *evaluate* in Figure 12. The function is an evaluator for the object language, taking an expression and returning its value. Notice the function is assigned the type  $\Pi t : ty. HOAS(t) \rightarrow HOAS(t)$ , indicating that the function is type-preserving at the object level.

Clearly, a natural question is whether we can also implement a type-preserving evaluator for an object language based on the second-order polymorphic  $\lambda$ -calculus (Girard 1972). In order to do so, we need to go beyond algebraic terms, employing  $\lambda$ -terms to encode polymorphic types in the object language. First we extend the definition of the sort *ty* as follows.

```

fun evaluate (v as Int _) = v
| evaluate (v as Bool _) = v
| evaluate (Add (e1, e2)) =
  let
    val Int (i1) = evaluate e1 and Int (i2) = evaluate e2
  in
    Int (i1+i2)
  end
| evaluate (Sub (e1, e2)) =
  let
    val Int (i1) = evaluate e1 and Int (i2) = evaluate e2
  in
    Int (i1-i2)
  end
| evaluate (Mul (e1, e2)) =
  let
    val Int (i1) = evaluate e1 and Int (i2) = evaluate e2
  in
    Int (i1*i2)
  end
| evaluate (Div (e1, e2)) =
  let
    val Int (i1) = evaluate e1 and Int (i2) = evaluate e2
  in
    Int (i1/i2)
  end
| evaluate (IsZero e) =
  let
    val Int (n) = evaluate e
  in
    Bool (n=0)
  end
| evaluate (If (e0, e1, e2)) =
  let
    val Bool (b) = evaluate e0
  in
    if b then evaluate e1 else evaluate e2
  end
| evaluate (App (e1, e2)) =
  let
    val Lam (f) = evaluate e1
  in
    evaluate (f (evaluate e2))
  end
| evaluate (Let (e, f)) = evaluate (f (evaluate e))
| evaluate (v as Lam _) = v
| evaluate (Fix (f)) = evaluate (f (Fix (f)))
withtype {t: ty} HOAS(t) -> HOAS(t)

```

Figure 12: An implementation of a type-preserving evaluation function in DML

```
sort ty = ... | All of ty -> ty
```

Given a term  $f$  of sort  $ty \rightarrow ty$ ,  $All(f)$  represents the type  $\forall\alpha.\tau$  if for each type  $\tau_0$ ,  $f(t)$  represents the type  $\tau[\alpha := \tau_0]$  as long as  $t$  represents the type  $\tau_0$ . For instance,  $All(\lambda a.Arrow(a, Arrow(a, I)))$  represents the type  $\forall\alpha.\alpha \rightarrow \alpha \rightarrow \text{int}$ ; the term  $All(\lambda a.(All(\lambda b.Arrow(a, Arrow(b, a)))))$  represents the type  $\forall\alpha\forall\beta.\alpha \rightarrow \beta \rightarrow \alpha$ . With this strategy, we see no difficulty in implementing a type-preserving evaluator for an object language based on the second-order polymorphic language calculus. However, we emphasize that what is outlined above is not available in DML at present as no higher-order terms can be used as type index expressions.

## 5 Limitations

We mention some limitations of dependent datatypes in this section.

In order to capture certain invariants in a data structure, we are often in need of declaring new datatypes instead of using existing ones. For instance, we can use the following syntax to declare a type constructor *ointlist* such that for each integer  $i$ , *ointlist*( $i$ ) is the type for ordered ascending integer lists that have the first element equal to  $i$  if not empty.

```
datatype ointlist with int =
  {i:nat} onil(i) | {i:int,j:int | i <= j} ocons(i) of int(i) * ointlist(j)
```

We may want to fetch the  $n$ th element in such an ordered integer list or sum up its elements, but we cannot use the corresponding functions already defined for (integer) lists. Instead, we have to redefine these functions for ordered integer lists. In other words, the use of dependent datatypes can adversely affect code reuse.

Another limitation can be illustrated by using the following example, where we declare a type constructor *braintree* for constructing types for Braun trees (Braun and Rem 1983).

```
datatype 'a braintree with nat =
  L(0)
  | {m:nat, n:nat | n <= m <= n+1}
  B(m+n+1) of 'a * 'a braintree(m) * 'a braintree(n)
```

A Braun tree is a form of balanced binary tree such that for every branch node in the tree, its left subtree either has the same size as its right subtree, or contains one more element. Suppose that  $B(x, l, r)$  occurs in the code where *the programmer knows or believes* for some reason that  $l$  has the same size as  $r$  or contains one more element but this property cannot be established in the type system of DML. In this case, the code is rejected by the DML type-checker, though the code will cause no run-time error (if the programmer is correct). The situation is very similar to the case where we move from an untyped programming language into a typed one. A solution to this problem is that we introduce some run-time checks. For instance, we may define the following function and replace  $B(x, l, r)$  with *make\_braintree*( $x, l, r$ ), where the function *make\_braintree* is defined as follows:

```
fun make_braintree (x, l, r) =
  let
    val m = size(l) and n = size(r)
  in
    if n <= m andalso m <= n+1 then B(x, l, r) else raise Illegal_argument
  end
withtype int * braintree * braintree -> braintree
```

The function *size*, which is assigned the type  $\forall\alpha\Pi n : \text{nat}.(\alpha)\text{braintree}(n) \rightarrow \text{int}(n)$ , computes the size of a Braun tree. The function *make\_braintree* can readily pass type-checking in DML but the penalty is that *make\_braintree* takes  $O(\log^2 n)$  time to build a Braun tree of size  $n$  (as there is an  $O(\log^2 n)$  algorithm for computing the size of a Braun tree (Okasaki 1998)), though this can be avoided if we store size information in each node.

In general, if the programmer anticipates the above situation to occur frequently, then he or she should either make sure that run-time checks can be done efficiently or switch back to non-dependent datatypes. We recommend that the programmer avoid complex encodings when using dependent datatypes to capture invariants in data structures.

## 6 A Comparison with Nested Datatypes

Both dependent datatypes and nested datatypes generalize the notion of datatypes, enabling the programmer to capture more invariants in data structures. For instance, either dependent datatypes or nested datatypes can be constructed to precisely model various balanced trees such Braun trees and red-black trees. However, we find it difficult to relate the ways in which they capture invariants.

Sometimes, we find it both less intuitive and difficult to capture size information with nested datatypes. Although there are certain common techniques (Hinze 2001), the issue is still not trivial at all. For instance, the reader may find the following declaration of a nested datatype intriguing.

```
datatype ('a, 'b) square_matrix' =
  Zero
| Even of ('a * 'a * 'a * 'a, 'b * 'b) square_matrix'
| Odd of 'a * 'b * 'b * ('a * 'a * 'a * 'a, ('a * 'b) * ('a * 'b)) square_matrix'

type 'a square_matrix = ('a, unit) square_matrix'
```

Actually, as its name suggests, values of the type  $(\tau)square\_matrix$  represent square matrices whose elements have the type  $\tau$ .<sup>8</sup> As an example, the following  $3 \times 3$  matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

can be represented as follows.

```
Odd (1, (), (), Odd ((5,6,8,9), ((2,()), (3, ())), ((4,()), (7, ())), Zero)
```

While it is difficult to construct such a nested datatype, it seems even more difficult to program with it. To fully understand the point, the reader may try to implement an  $O(\log n)$ -time subscripting function for such a matrix representation!

On the other hand, it is trivial to form a dependent type for square matrices. For instance, we can use the type  $(\tau)SquareMatrix(n) = ((\tau)ralist(n))ralist(n)$  for square matrices with dimension  $n \times n$  in which each element has the type  $\tau$ . This allows us to implement a subscripting function for square matrices by using the already defined lookup function for random-access lists, facilitating code reuse.

There is a much more serious problem with nested datatypes. We find that while nested datatypes are effective in capturing invariants within a data structure, they are ineffective in doing so between data structures. For instance, we can define a function *dimen* as follows to compute the dimension of a square matrix:

```
fun ('a, 'b)
  dimen Zero = 0
| dimen (Even mat) = 2 * dimen mat
| dimen (Odd (_, _, _, mat)) = 1 + 2 * dimen mat
withtype ('a, 'b) square_matrix' -> int
```

Unfortunately, we cannot establish any relation between *mat* and *dimen(mat)* in terms of their types. Hence, there is simply no possibility to implement a subscripting function whose type can prevent it from

<sup>8</sup>An elegant approach to constructing nested datatypes for square matrices can be found in (Okasaki 1999), which uses the feature of higher-order type constructors.

being applied to out-of-bounds indexes. Also, it is impossible to implement a matrix multiplication function that can enforce through its type the requirement that its two arguments have the same dimension. To a large extent, the type  $(\tau) \text{square\_matrix}$  is like the type  $\Sigma n : \text{nat}(\tau) \text{SquareMatrix}(n)$ : while the invariant of being a square matrix is captured, there is no information available on the dimension of the matrix.

We feel that dependent datatypes and nested datatypes are basically two orthogonal approaches to forming more accurate datatypes. They can complement each other in programming as is shown in Section 4.2: The first implementation of random-access lists, which makes use of only dependent datatypes, cannot achieve  $O(1)$ -time consing and unconsing, but the second implementation, which makes use of both dependent and nested datatypes, can.

## 7 Related Work

The use of types in program error detection is ubiquitous in programming. Usually, the types in general purpose programming languages such as ML and Java are relatively inexpressive for the sake of practical type-checking. In these languages, the use of types in program verification is effective but too limited. DML aims at providing a more expressive and yet still practical type system to allow the programmer to capture more program properties through types and thus detect more program errors at compile-time. In addition, types in DML can serve as informative program documentation, facilitating program comprehension. We assign priority to the practicality of type-checking in our language design and emphasize the need for restricting the expressiveness of a type system.

Both inspired by and related to Martin-Löf’s type theory (Martin-Löf 1984; Nordström, Petersson, and Smith 1990; Thompson 1991), our work falls in between full program verification, either in type theory such as NuPrl (Constable et al. 1986) and Coq (Dowek, Felty, Herbelin, Huet, Murthy, Parent, Paulin-Mohring, and Werner 1993), or systems such as PVS (Owre, Rajan, Rushby, Shankar, and Srivas 1996), and traditional type systems for programming languages. When compared to verification, our system is less expressive but more automatic when constraint domains with practical constraint satisfaction problems are chosen.

Most closely related to our work is the system of *indexed types* developed independently by Zenger in his Ph.D. Thesis (Zenger 1998) (an earlier version of which is described in (Zenger 1997)). He works in the context of lazy functional programming. His language is clean and elegant and his applications (which significantly overlap with ours) are compelling. In general, his approach seems to require more changes to a given Haskell program to make it amenable to checking indexed types than is the case for our system and ML. This is particularly apparent in the case of existential dependent types, which are tied to data constructors. This has the advantage of a simpler algorithm for elaboration and type-checking than ours, but the program (and not just the type) has to be more explicit.

When compared to traditional type systems for programming languages, perhaps the closest related work is refinement types (Freeman and Pfenning 1991), which also aims at expressing and checking more properties of programs that are already well-typed in ML, rather than admitting more programs as type correct, which is the goal of most other research on extending type systems. However, the mechanism of refinement types is quite different and incomparable in expressive power: while refinement types incorporate intersection and can thus ascribe multiple types to terms in a uniform way, dependent types can express properties such as “*these two argument lists have the same length*” which are not recognizable by tree automata (the basis for type refinements).

There have been many recent studies on the use of nested datatypes (Bird and Meertens 1998) in constructing (sophisticated) datatypes to capture more invariants in data structures. For instance, a variety of examples can be found in (Bird and Paterson 2001; Okasaki 1999; Hinze 2001; Hinze 1999). We feel that the advantage of this approach is that it requires relatively minor language extensions, which may include polymorphic recursion, higher-order kinds, rank-2 polymorphism, to existing functional programming languages such as Haskell, while type-checking in DML is more involved. On the other hand, this approach seems less flexible, often requiring some involved treatment at both type and program level. The important notion of datatype refinement in DML cannot be captured with nested datatypes. For instance, it is impossible to form a nested datatype that can capture the notion of the length of a list since this would imply that one could simply use such types to distinguish non-empty lists from empty ones. In

general, we think that these two approaches are essentially orthogonal in spite of some similar motivations behind their development and they can be readily combined with little effort.

## 8 Conclusion

The use of dependent datatypes in capturing invariants in data structures is novel. This practice can offer many advantages in implementing algorithms. The most significant advantage is probably in program error detection, making programs more robust. We argued in Section 1 that the imprecision of datatypes in Standard ML or Haskell in capturing invariants in data structures can be a rich source for run-time program errors. In addition, the dependent type annotations supplied by the programmer are mechanically verified and can thus be fully trusted. They can serve as valuable program documentation, facilitating program understanding. The use of dependent types in eliminating run-time array bound checks can also lead to more efficient program execution in safe languages like ML and Java, where out-of-bounds array subscripting is disallowed.

Type-checking in DML is largely independent of the size of a program since a type-checking unit is roughly the body of a toplevel function. In general, what matters in type-checking is the difficulty level of the properties that are to be checked. A more serious issue is about accurately reporting error messages in case of type errors. The type-checking in DML implements a top-down style algorithm, which usually pinpoints to the location of a type error. Unfortunately, the author finds that it may often be surprisingly difficult to figure out the cause of a type error. However, on the positive side, the type-checker of DML is often capable of detecting a variety of subtle errors. For instance, the author once used `Even(11, 12)` to form a random-list (in Figure 8) and the type-checker raised an error because it could not prove that `11` cannot be `Nil`. If this had gone unnoticed, it would have invalidated some invariant assumed by the programmer, potentially causing (difficult) run-time errors. We are currently in the process of implementing DML, planning a documented release.

The usual focus of data structure design is mainly on enhancing time and/or space efficiency, and less attention is paid to program error detection. The introduction of dependent datatypes provides an opportunity to remedy the situation. In general, we are interested in promoting the use of light-weight formal methods in practical programming. We have presented some concrete examples of dependent datatypes in this paper in support of such a promotion. We hope these examples can raise the awareness of dependent datatypes and their use in implementing algorithms.

## 9 Acknowledgment

I thank Chris Okasaki, Ralf Hinze and some anonymous referees for their constructive comments, which have undoubtedly raised the quality of the paper.

## References

- Bird, R. and L. Meertens (1998). Nested datatypes. In *Mathematics of program construction*, pp. 52–67. Springer-Verlag LNCS 1422.
- Bird, R. and R. Paterson (2001). de Bruijn notation as a nested datatype. *Journal of Functional Programming*.
- Braun, W. and M. Rem (1983). A logarithmic implementation of flexible arrays. Technical Report Memorandum MS83/1, Eindhoven University of Technology.
- Church, A. (1940). A formulation of the simple type theory of types. *Journal of Symbolic Logic* 5, 56–68.
- Constable, R. L. et al. (1986). *Implementing Mathematics with the NuPrl Proof Development System*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Dowek, G., A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner (1993). The Coq proof assistant user’s guide. Rapport Technique 154, INRIA, Rocquencourt, France. Version 5.8.

- Freeman, T. and F. Pfenning (1991). Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Ontario, pp. 268–277.
- Girard, J.-Y. (1972, June). *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université de Paris VII, Paris, France.
- Hinze, R. (1999, September). Constructing Red-Black Trees. In *Proceedings of Workshop on Algorithmic Aspects of Advanced Programming Languages*.
- Hinze, R. (2001, September). Manufacturing Datatypes. *Journal of Functional Programming* 11(5), 493–524.
- Kahrs, S. (2001, July). Functional pearl: red-black trees with types. *Journal of Functional Programming* 11(4), 425–432.
- Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Naples, Italy: Bibliopolis.
- Milner, R., M. Tofte, R. W. Harper, and D. MacQueen (1997). *The Definition of Standard ML (Revised)*. Cambridge, Massachusetts: MIT Press.
- Nordström, B., K. Petersson, and J. M. Smith (1990). *Programming in Martin-Löf's Type Theory*, Volume 7 of *International Series of Monographs on Computer Science*. Oxford: Clarendon Press.
- Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press.
- Okasaki, C. (1999, September). From Fast Exponentiation to Square Matrices: An Adventure in Types. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*.
- Owre, S., S. Rajan, J. Rushby, N. Shankar, and M. Srivas (1996, July/August). PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger (Eds.), *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV '96)*, New Brunswick, NJ, pp. 411–414. Springer-Verlag LNCS 1102.
- Peyton Jones, S. et al. (1999, February). Haskell 98 – A non-strict, purely functional language. Available at <http://www.haskell.org/onlinereport/>.
- Pfenning, F. *Computation and Deduction*. Cambridge University Press. (to appear).
- Thompson, S. (1991). *Type Theory and Functional Programming*. Reading, Massachusetts: Addison-Wesley.
- Xi, H. (1998). *Dependent Types in Practical Programming*. Ph. D. thesis, Carnegie Mellon University. pp. viii+189. Available at <http://www.cs.cmu.edu/~hwxi/DML/thesis.ps>.
- Xi, H. (2000, June). Imperative Programming with Dependent Types. In *Fifteenth IEEE Symposium on Logic in Computer Science*, Santo Barbara, pp. 375–387.
- Xi, H. (2001). Dependent ML. Available at <http://www.cs.bu.edu/~hwxi/DML/DML.html>.
- Xi, H. (2002, March). Dependent Types for Program Termination Verification. *Journal of Higher-Order and Symbolic Computation* 15(1), 91–132.
- Xi, H. and F. Pfenning (1998, June). Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montréal, Canada, pp. 249–257.
- Xi, H. and F. Pfenning (1999, January). Dependent Types in Practical Programming. In *Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, San Antonio, Texas, pp. 214–227. ACM press.
- Zenger, C. (1997). Indexed types. *Theoretical Computer Science* 187, 147–165.
- Zenger, C. (1998). *Indizierte Typen*. Ph. D. thesis, Fakultät für Informatik, Universität Karlsruhe.