Dependently Typed Data Structures*

Hongwei Xi

Oregon Graduate Institute

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 the invariants inherent in data structures. We remedy the situation with the introduction of dependent datatypes so that we can model data structures with significantly more accuracy. We present a few interesting examples such as implementations of red-black trees and binomial heaps to illustrate the use of dependent datatypes in capturing some sophisticated invariants in data structures. We claim that dependent datatypes can enable the programmer to implement algorithms in a way that is more robust and easier to understand.

1 Introduction

The mechanism that allows the programmer to declare datatypes seems indispensable in functional programming languages such as Standard ML [15] and Haskell [19]. In practice, we often encounter situations where the declared datatypes do not accurately capture what we really need. For instance, if what we need is a data structure for the pairs of integer lists of the same 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 program errors. A typical scenario is that a function which should only receive as its argument a pair of integer lists of the same length is mistakenly applied to a pair of integer lists of different lengths. 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 hide in a program unnoticed until at run-time, when debugging often becomes much more demanding than at compile-time.

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 some relatively complex 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 yield a red-black tree. If we can form a datatype to precisely capture the properties of a red-black tree, then it becomes possible to detect a program error through type-checking when such an error leads to the violation of one of these 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), 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 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 DML(C) can then be reduced to constraint satisfaction in C. In this paper, we restrict C to some integer domain and use the name DML for this particular DML(C). A variant of DML, *de Caml*, has been implemented on top of Caml-light [14]. This implementation essentially replaces the front-end of

*Partially supported by the United States Air Force Materiel Command (F19 628-96-C-0161) and the Department of Defense.

Caml-light with a dependent type-checker and keeps the back-end of Caml-light intact. It also modifies many library functions, assigning to them more accurate types.

An alternative approach to forming more accurate datatypes is to use nested datatypes [2]. For instance, a nested datatype exactly representing red-black trees can be readily formed. However, there exist various significant differences between DML-style dependent types and nested datatypes, which we will illustrate later.

We use `typewriter font` in this paper to represent code in de Caml, all of which have been verified in a prototype implementation. A significant consequence of the introduction of dependent types is the loss of the notion of principal types in DML. For instance, both of the following types can be assigned to an implementation in de Caml which 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 given a pair of lists with length n. Notice that we use `'a list(n)` for the type of a list with length n in which every element is of type `'a`. If a dependent type is to be assigned to a function in 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 ML and in DML. In practice, we observe that the type annotations in a typical DML program often constitutes less than 20% of the entire code. Since dependent type annotations can often lead to more accurate reports of type error messages and serve as informative program documentation, we feel that the DML programming style is acceptable from a practical point of view. We will provide some concrete examples for the reader to judge this claim, including implementations of red-black trees and binomial heaps. Both of these implementations are adopted from the corresponding ones in [17]. The implementations in de Caml have several advantages over the original ones. We have verified more invariants in the de Caml implementations. For instance, it is verified in the type system of de Caml that the function which merges two binomial heaps indeed yields a binomial heap. Also the type annotations in the implementations, which can be fully trusted since they are mechanically verified, offer some pedagogical values. We feel it is easier to understand the de Caml implementations because the reader can reason in the presence of these informative dependent types.

In this paper, it is neither possible nor necessary to formally present DML. Instead, we focus on presenting some concrete examples in the programming language *de Caml*, a variant of DML, as well as some intuitive explanation. We refer the interested reader to [22] for the formal development of DML, though we strongly believe that this is largely unnecessary for comprehending this paper.

The rest of the paper is organized as follows. In Section 2, we give a brief overview of the types in DML. We then introduce de Caml in Section 3, presenting some of its main features and illustrating type-checking in de Caml with a short example. Some case studies are given in Section 4, including implementations of Braun trees, random-access lists, red-black trees and binomial heaps. Lastly, we discuss some related work and then conclude.

2 Types in Dependent ML

In this section, we present a brief explanation on the types in DML. The reader is encouraged to skip this section and read it later, though it could be helpful to gather some intuition before studying the concrete examples in Section 3.

Intuitively speaking, dependent types are types which depend on the values of language expressions. For instance, we may form a type $int(i)$ for each integer i to mean that every integer expression of this type must have value i, that is, $int(i)$ is a singleton type. Note that i is the expression on which this type depends. We use the name *type index expression* for such an expression. There are various compelling reasons, such as practical type-checking, for imposing restrictions on expressions which can be chosen as type index expressions. A

$$
\begin{array}{rrcl}
\text{index expressions} & i, j & ::= & a \mid c \mid i + j \mid i - j \mid i * j \mid i \div j \mid \min(i,j) \mid \max(i,j) \mid \mathrm{mod}(i,j) \\
\text{index propositions} & P & ::= & i < j \mid i \leq j \mid i \geq j \mid i > j \mid i = j \mid i \neq j \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \\
\text{index sorts} & \gamma & ::= & int \mid \{a : \gamma \mid P\} \mid \gamma_1 * \gamma_2 \\
\text{index contexts} & \phi & ::= & \cdot \mid \phi, a : \gamma \mid \phi, P
\end{array}
$$

Figure 1: The syntax for type index expressions

novelty in DML is to require that type index expressions be drawn only from a given constraint domain. For the purpose of this paper, we restrict type index expressions to integers. We present the syntax for type index expressions in Figure 1, where we use a for type index variables and c for a fixed integer. Note that the language for type index expressions is typed. We use *sorts* for the types in this language in order to avoid potential confusion. We use $\cdot$ for the empty index variable context and omit the standard sorting rules for this language. We also use certain transparent abbreviations, such as $0 \leq i < j$ which stands for $0 \leq i \wedge i < j$. The subset sort $\{a : \gamma \mid P\}$ stands for the sort for those elements of sort γ which satisfy the proposition P. For example, we use *nat* as an abbreviation for the subset sort $\{a : int \mid a \geq 0\}$.

Types in DML are formed as follows. We use α for type variables and δ for type constructors.

$$
\text{types} \quad \tau \quad ::= \quad \alpha \mid (\tau_1, \ldots, \tau_n)\delta(i) \mid \mathbf{1} \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2 \mid \Pi a : \gamma.\tau \mid \Sigma a : \gamma.\tau
$$

For instance, $list$ is a type constructor and $(int)list(n)$ stands for the type of an integer list of length n. $\Pi a : \gamma.\tau$ and $\Sigma a : \gamma.\tau$ form a universal dependent type and an existential dependent type, respectively. For instance, the universal dependent type $\Pi a : nat.(int)list(a) \to (int)list(a)$ captures the invariant of a function which, for every natural number a, returns an integer list of length a when given an integer list of length a. Also we can use the existential dependent type $\Sigma a : nat.(int)list(a)$ to mean an integer list of some unknown length. We demonstrate how a type constructor is declared in Section 3.

The typing rules for this language should be familiar from a dependently typed λ-calculus (such as the ones underlying Coq or NuPrl). The critical notion of *type conversion* uses the judgment $\phi \vdash \tau_1 \equiv \tau_2$ which is the congruent extension of equality on index expressions to arbitrary types:

$$
\frac{\phi \models \tau_1 \equiv \tau_1' \quad \cdots \quad \tau_n \equiv \tau_n' \quad \phi \models i \doteq i'}{\phi \vdash (\tau_1, \ldots, \tau_n)\delta(i) \equiv (\tau_1', \ldots, \tau_n')\delta(i')}
$$

$$
\frac{\phi \vdash \tau_1 \equiv \tau_1' \quad \phi \vdash \tau_2 \equiv \tau_2'}{\phi \vdash \tau_1 * \tau_2 \equiv \tau_1' * \tau_2'} \qquad \frac{\phi \vdash \tau_1' \equiv \tau_1 \quad \phi \vdash \tau_2 \equiv \tau_2'}{\phi \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'}
$$

$$
\frac{\phi, a : \gamma \vdash \tau \equiv \tau'}{\phi \vdash \Pi a : \gamma.\tau \equiv \Pi a : \gamma.\tau'} \qquad \frac{\phi, a : \gamma \vdash \tau \equiv \tau'}{\phi \vdash \Sigma a : \gamma.\tau \equiv \Sigma a : \gamma.\tau'}
$$

Notice that it is the application of these rules which generates constraints. For instance, the constraint $\phi \models (a + n) + 1 \doteq m + n$ is generated in order to derive $\phi \vdash (int)list((a + n) + 1) \equiv (int)list(m + n)$.

It is difficult to present more details given the space limitation. For those who are interested, we point out that the detailed formal development of DML can be found in [22].

3 Some Features in de Caml

In this section, we use examples to present some unique and significant features in de Caml, preparing for the case studies in Section 4.

The programmer often declares datatypes when programming in ML. For instance, the following datatype declaration defines a type constructor $list$.

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

Roughly speaking, this declaration states that a polymorphic list is formed with two constructors `nil` and `cons`, whose types are `'a list` and `'a * 'a list -> 'a list`, respectively. We use `'a` for a type variable. However, the declared type `'a list` is coarse. For instance, we cannot use the type to distinguish an empty list from a non-empty one. In de Caml, this type can be refined as follows.

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

The clause `refine 'a list with nat` means that we refine the type `'a list` with an index of sort `nat`, that is, the index is a natural number. In this case, the index stands for the length of a list.

- `nil(0)` means that `nil` is of type `'a list(0)`, that is, it is a list of length 0.
- `{n:nat} cons(n+1) of 'a * 'a list(n)` means that `cons` is of type

  `{n:nat} 'a * 'a list(n) -> 'a list(n+1)`,

  that is, for every natural number `n`, `cons` yields a list of length $n+1$ when given an element of type `'a` and a list of length n. Note `{n:nat}` is a universal quantifier, which is usually written as $\Pi n : nat$ in type theory.

Now list types have become more informative. The following code defines the append function on lists. We use `[]` for `nil` and `::` as the infix operator for `cons`.

```
let rec append = function
    ([], ys) -> ys
  | (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 of $m+n$ when given a pair of lists of lengths m and n, respectively. We now present an informal description about type-checking in this case.

For the first clause `([], ys) -> ys`, the type-checker assumes that `ys` is of types `'a list(b)` for some index variable b of sort `nat`. This implies that `([], ys)` is of type `'a list(0) * 'a 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 `->` is of type `'a list(0+b)`. Since `ys` is of type `'a list(b)` under assumption, 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 `(x :: xs, ys) -> x :: append(xs, ys)`. Assume that `xs` and `ys` are of type `'a list(a)` and `'a list(b)`, respectively, where a and b are index variables of sort `nat`. Then `(x :: xs, ys)` is of type `'a list(a+1) * 'a 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)` is of type `'a list((a+b)+1)` since xs and ys are assumed of types `'a list(a)` and `'a list(b)`, respectively. We need to prove that the right side is of type `int list(m+n)` for $m = a+1$ and $n = b$. This leads to 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 de Caml program. The interested reader is referred to [22] for the formal presentation of type-checking in DML.

Clearly, a natural question is whether the type for `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.

Instead of refining a type, it is also allowed to declare a dependent type in de Caml. For instance, we can declare the following.

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

The declaration is basically equivalent to the refinement we made earlier. However, there is also a significant difference. When we declare a refinement, we must be able to interpret the corresponding unrefined types in terms of refined ones. For example, after refining the type `'a list`, we must interpret this type in terms of the refined list type. We need existential dependent types for this purpose. `'a list` is interpreted as `[n:nat] 'a list(n)`, that is, `'a list` is `'a list(n)` for some (unknown) natural number n. Note that `[n:nat]` is an existential quantifier, which is often written as $\Sigma n : nat$ in type theory. This provides a smooth interaction between ML types and dependent types. Suppose that f is defined before the list type is refined and its type is `'a list -> 'a list`. After refining the list type, we can assign to f the type `([n:nat] 'a list) -> [n:nat] 'a list`, that is, 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 of dependent type, say, `int list(2)`. This is also essential for ensuring backward compatibility, a very important issue when the use of existing ML code is concerned.

However, there is a need for imposing some restriction on datatype refinement. We give a short example to illustrate such a need. The datatype `'a tree` is declared as follows for *all* binary trees.

```
datatype 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

Suppose we declare the following refinement, where the type index standards for the height of a tree.

```
refine 'a tree with nat =
  Leaf(0) | {h:nat} Node(h+1) of 'a tree(h) * 'a * 'a tree(h)
```

This refinement is problematic since the type `[h:nat] 'a tree(h)` now standards for the type of all *perfect* binary trees, and therefore it cannot be used to represent the original `'a tree`, which is the type for all binary trees. There is some syntactic restriction that can be imposed to rule out such problematic datatype refinements. We stop mentioning the restriction since it is simply not needed in this paper.

There is another important use of existential dependent types. In order to guarantee practical type checking in de Caml, 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 de Caml. For instance, the following code implements a filter function on a list which removes from the list all elements not satisfying the property `p`.

```
let filter p = function
    [] -> []
  | 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 `(filter p l)` without knowing what `p` is. 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 `l`. This invariant can be captured by assigning `filter` the following types.

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

Note that `[n:nat | n <= m]` stands for $\Sigma n : \{a : nat \mid a \leq m\}$.

Another significant use of existential dependent types is to represent a range of values. We can use `([n:nat] int(n)) array` to represent the type for the vectors whose elements are natural numbers.

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

let rec diff k = function
    L -> 0
  | B(_, l, r) ->
    if k = 0 then 1
    else if k mod 2 = 1 then diff (k/2) l else diff (k/2 - 1) r
withtype {k:nat}{n:nat | k <= n <= k+1}
         int(k) -> 'a brauntree(n) -> int(n-k) ;;

let rec size = function
    L -> 0 | B(_, l, r) -> let n = size r in 1 + n + n + diff n l
withtype {n:nat} 'a brauntree(n) -> int(n) ;;
```

Figure 2: An implementation of the size function on Braun trees

This is very useful for eliminating array bound checks at run-time [20]. In general, we view that the use of existential types in de Caml for handling functions like `filter` is crucial to the scalability of the type system of de Caml since such functions are abundant in practice.

Lastly, we mention a convention in de Caml. After declaring a dependent type as follows,

$$\texttt{datatype}\ (\alpha_1, \ldots, \alpha_m)\ \delta\ \texttt{with}\ (sort_1, \ldots, sort_n) = \cdots\cdots$$

we may write $(\tau_1, \ldots, \tau_m)\delta$ to stand for the following.

$$[a_1 : sort_1] \cdots [a_n : sort_n].(\tau_1, \ldots, \tau_m)\delta(a_1, \ldots, a_n)$$

For example, `'a list` stands for `[n:nat] 'a list(n)`.

4 Case Studies

In this section, we present some examples to demonstrate the use of dependent datatypes in capturing invariants in data structures. All these examples in de Caml have been successfully verified in a prototype compiler for de Caml, which is written on top of the Caml-light compiler [14]. The claim we make is that dependent datatypes enable the programmer to implement algorithms in a way that is more robust and easier to understand.

4.1 Braun Trees

A Braun tree is a balanced binary tree [4] such that for every branch node in the tree, its left subtree is either the same size as its right subtree, or contains one more element. Braun trees can be used to give neat implementations for flexible arrays and priority queues. In [16], there is an algorithm which computes the size of a Braun tree in $O(\log^2 n)$ time, where n is the size of the Braun tree. We implement this algorithm in Figure 2. We first declare a dependent datatype `'a brauntree(n)` for Braun trees of size n. Note that the type of `B` is

```
{m:nat}{n:nat | n <= m <= n+1}
'a * 'a brauntree(m) * 'a brauntree(n) -> 'a brauntree(m+n+1)
```

which states that `B` yields a Braun tree of size $m+n+1$ when given an element, a Braun tree of size m and a Braun tree of size n where $n \leq m \leq n+1$ holds. This exactly captures the invariant on Braun trees mentioned above.

Given a natural number k and a Braun tree of size n satisfying $k \leq n \leq k+1$, the function `diff` yields the difference between n and k. With this function, the size function on Braun trees can be defined straightforwardly. An interesting point in this example is that the type of the function `size` precisely indicates that this is the size function on Braun trees since it states that the function returns an integer of value n when given a Braun tree of size n.

The reason that `diff n l` yields the difference between $|l|$, the size of l, and n can be found in [16]. We give some brief explanation below. It is clear that $|l| - n$ is either 0 or 1. If l is a leaf, $|l| - n$ must be 0. Otherwise, $|l| = 1 + |l'| + |r'|$, where l' and r' are the left and right branches of l, respectively. If n is odd, then $n = 1 + \lfloor n/2 \rfloor + \lfloor n/2 \rfloor$ and thus

$$|l| - n \quad = \quad 1 + |l'| + |r'| - 1 + \lfloor n/2 \rfloor + \lfloor n/2 \rfloor = (|l'| - \lfloor n/2 \rfloor) + (|r'| - \lfloor n/2 \rfloor)$$

Since $|l'| - 1 \leq |r'| \leq |l'|$ holds, we have the following.

$$2(|l'| - \lfloor n/2 \rfloor) - 1 \leq |l| - n \leq 2(|l'| - \lfloor n/2 \rfloor)$$

It can now be readily verified that $|l| - n = 1$ if $|l'| - \lfloor n/2 \rfloor = 1$ and $|l| - n = 0$ if $|l'| - \lfloor n/2 \rfloor = 0$. Therefore, if n is odd, $|l| - n = |l'| - \lfloor n/2 \rfloor$. With some similar reasoning, we can eventually prove the correctness of the defined function `diff`.

This example also shows that although the datatype type declaration for Braun trees contains size information, this information is not available at run-time and therefore a recursive walk through the tree is necessary to determine the size of a tree.

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 (worst case), where n is the length of the input list.

We present an implementation of random-access list in Figures 3 and 4. We first declare the dependent datatype for random-access lists. Note that `'a rlist(n)` stands for the type of random-access lists with length n. `Nil` and `One` are the constructors for empty and singleton random-access lists, respectively. Furthermore, the constructors `Even` and `Odd` are to form random-access lists of even and odd lengths, respectively. If `l1` and `l2` represent lists $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ for some $n > 0$, respectively, then `Even (l1, l2)` represents the list $x_1, y_1, \ldots, x_n, y_n$. Similarly, if `l1` and `l2` represent lists $x_1, \ldots, x_n, x_{n+1}$ and $y_1, \ldots, y_n$ for some $n > 0$, respectively, `Odd(l1, l2)` represents $x_1, y_1, \ldots, x_n, y_n, x_{n+1}$. 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 `l1` and `l2` must have the same length if `Even(l1, l2)` is formed or `l1` contains one more element than `l2` if `Odd(l1, l2)` is formed. This is clearly captured by the dependent datatype declaration for `'a rlist`. 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 this function only to be applied to a non-empty list. Both `cons` and `uncons` takes $O(\log n)$ 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 `look_up i l` simply return `x` when the `l` matches the pattern `One x`. There is no need to check whether

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

exception Subscript ;;

let rec cons x = function
    Nil -> One x
  | One y ->  Even(One(x), One(y))
  | Even(l1, l2) -> Odd(cons x l2, l1)
  | Odd(l1, l2) -> Even(cons x l2, l1)
withtype {n:nat} 'a -> 'a rlist(n) -> 'a rlist(n+1) ;;

let rec uncons = function
    One x -> (x, Nil)
  | Even(l1, l2) ->
    let (x, l1) = uncons l1 in begin
      match l1 with
        Nil -> (x, l2) | _ -> (x, Odd(l2, l1))
    end
  | Odd(l1, l2) -> let (x, l1) = uncons l1 in (x, Even(l2, l1))
withtype {n:nat | n > 0} 'a rlist(n) -> 'a * 'a rlist(n-1) ;;

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

let rec lookup_safe i = function
    One x -> x
  | Even(l1, l2) ->
    if i mod 2 = 0 then lookup_safe (i / 2) l1
    else lookup_safe (i / 2) l2
  | Odd(l1, l2) ->
    if i mod 2 = 0 then lookup_safe (i / 2) l1
    else lookup_safe (i / 2) l2
withtype {i:nat}{n:nat | i < n} int(i) -> 'a rlist(n) -> 'a ;;
```

Figure 3: An implementation of random-access lists in de Caml (I)

```
let rec update_safe i x = function
    One y -> One x
  | Even(l1, l2) ->
    if i mod 2 = 0 then Even(update_safe (i / 2) x l1, l2)
    else Even(l1, update_safe (i / 2) x l2)
  | Odd(l1, l2) ->
    if i mod 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 rlist(n) -> 'a rlist(n) ;;
```

Figure 4: An implementation of random-access lists in de Caml (II)

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

Figure 5: A nested dependent datatype for random access lists

`i` is 0: it must be 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.

```
let rec 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 rlist -> 'a ;;
```

We point out that an implementation of random-access lists is given in [17], which uses the feature of nested datatypes. Okasaki's implementation supports (on average) $O(1)$-time consing and unconsing operations and are thus superior to our implementation in this respect. On the other hand, the update function in Okasaki's implementation requires the use of some higher-order feature, which does not exist in our implementation. We view this as an edge of our implementation.

It should be stressed that nested datatypes and DML-style dependent types are orthogonal to each other. For instance, we can form a nested dependent datatype in Figure 5 for random-access lists, imitating a corresponding datatype in [17]. Unfortunately, we currently cannot experiment with such a dependent datatype because polymorphic recursion is not supported in Caml-light.

4.3 Red-Black Trees

A red-black tree (RBT) is a balanced binary tree which 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 6, which precisely captures these properties of a RBT.

```
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) ;;

let restore = function
    (R(R(a, x, b), y, c), z, d) -> R(B(a, x, b), y, B(c, z, d))
  | (R(a, x, R(b, y, c)), z, d) -> R(B(a, x, b), y, B(c, z, d))
  | (a, x, R(R(b, y, c), z, d)) -> R(B(a, x, b), y, B(c, z, d))
  | (a, x, R(b, y, R(c, z, d))) -> R(B(a, x, b), y, B(c, z, d))
  | (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 ;;

let insert x t =
  let rec ins = function
      E -> R(E, x, E)
    | 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
    | 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
  match ins t with
    R(a, y, b) -> B(a, y, b)
  | t -> t
withtype {c:color}{bh:nat} key -> rbtree(c, bh, 0) ->
         [bh':nat] rbtree(0, bh', 0) ;;
```

Figure 6: A red-black tree implementation

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. Of course, one can readily use other ordered data structures. The type `rbtree` is indexed with a triple `(c, bh, v)`, where c is the color of the node, bh is the black height of the tree, and v is the number of color violations. We record one color violation if a red node is followed by another red node, and thus a RBT must have no color violations. Clearly, the types of constructors indicate that color violations cano 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

```
[c:color][bh:nat] rbtree(c,bh,0),
```

that is, a tree which 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 [17], 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 is of the following type.

```
key->[c:color][bh:nat] rbtree(c,bh,0)->[c:color][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 6 is defined for this purpose.

The type of `restore` is easy to understand. It states that this function takes an entry, a tree with at most one color violation and a RBT and returns a RBT tree. The two trees in the argument must have the same black height bh for some natural number bh and the returned RBT has black height $bh + 1$. This information can be of great help for understanding the code. If the information had been informally expressed through comments, it would be difficult to know whether the comments can be trusted. Also notice that it is not trivial at all to verify the information manually. 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 which 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-checking 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 _`, and `E`. Unfortunately, such expansion may lead to combinatorial explosion. An alternative is to require the programmer to indicate whether such expansion is needed. Neither of these is currently available in de Caml, and the author has taken the inconvenience to expand patterns into disjoint ones when necessary. We emphasize that the code in Figure 6 must be thus expanded in order to pass type-checking in de Caml. 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 that the type of function `ins` indicates that `ins` may return a tree with one color violation if it is applied to a tree with red top node. This is fixed 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 indicate the size 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 inserted entry already exists in the tree). Please refer to [23] for details.

4.4 Binomial Heaps

A binomial tree is defined recursively; a binomial tree B_0 with rank 0 consists of a single node and a binomial tree B_{k+1} of rank $k + 1$ consists of two linked binomial trees B_k of rank k such that the root of one B_k is the

leftmost son of the other B_k. A binomial heap H is a collection of binomial trees that satisfy the properties: (a) each binomial tree in H is heap-ordered, that is, the key of a node is greater that or equal to the key of its parent, and (b) there is at most one binomial tree in H whose root has a given degree. Please refer to [6] for details.

We declare some datatypes in Figure 7 for forming binomial heaps. The type `tree(n)` is for binomial trees of rank n, and the type `treelist(n)` is for a list of binomial trees with *decreasing* ranks and $n = m+1$ if the list is not empty, where m is the rank of the first binomial tree in the list. We represent a binomial heap as a list of binomial trees with *increasing* ranks. For a heap of type `heap(n)`, if $n = 0$ then the heap is empty; otherwise $n = m + 1$ where m is the rank of the first binomial tree in the heap. Notice that we attach rank to each tree node in order to efficiently compute the rank of a tree while using the type of `Node` to guarantee that the first component of each node indeed represents the rank of that node.

Notice that the datatype for binomial trees *does not* capture the invariant stating that these trees are heap-ordered. This seems to be beyond the reach of dependent datatypes. Also note that we would not be able to capture some of the invariants if we used the ordinary list constructors, that is, `nil` and `cons`, to form tree lists. This leads to the introduction of `Tempty`, `Tcons`, `Hempty` and `Hcons`. This special feature in programming with dependent datatypes has an unpleasant consequence, which we mention in Section 5.

The implementation in Figure 7 and 8 is largely adopted from [17]. Since the type for the function `merge` is relatively complex, we explain it as follows. This type states that given two binomial heaps of types `heap(m)` and `heap(n)`, respectively, this function returns a binomial heap of type `heap(l)` for some l such that $l = m$ if $n = 0$, or $l = n$ if $m = 0$, or $l \geq \min(m, n) > 0$ otherwise.

5 Limitation

We mention some limitations of dependent datatypes in this section.

In order to capture invariants, we may have to declare new datatypes instead of using existing ones. For instance, we declared the datatype `treelist` in Figure 7 instead of using the existing list constructors to form a list of trees. The reason is that we wanted to only form lists of binomial trees with decreasing rank. Similarly, we introduced the datatype `heap` to capture the invariant that a binomial heap is a list of trees with increasing order. This forces us to define the function `to_heap` later, which essentially reverses a list of trees and append it to a heap. If we used the existing list constructors without declaring either of `treelist` and `heap`, we could then use some existing function on lists instead of defining `to_heap`. In order words, using dependent datatypes may lose some opportunities for code reuse.

Another limitation can be illustrated using the following example. Let `B` be the constructor declared in Figure 2, which is used to form Braun trees. Suppose that `B(x, l, r)` occurs in the code where *the programmer knows* for some reason that `l` is the same size as `r` or contains one more element but this cannot be established in the type system of de Caml. In this case, the code is to be rejected by the de Caml type-checker, though the code will cause no run-time error (if we trust the programmer). 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_brauntree x l r`.

```
let make_brauntree x l r =
  let m = size(l) and n = size(r) in
    if n <= m && m <= n+1 then B(x, l, r) else raise Illegal_argument
withtype int -> brauntree -> brauntree -> brauntree
```

The function `make_brauntree` can readily pass type-checking in de Caml (we refer the interested reader to [22] for further details). The penalty in this case is that `make_brauntree` takes $O(\log^2 n)$ time to build a tree of size n, 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 she or he should either make sure that run-time checks can be done efficiently or switch back to non-dependent datatypes.

```
datatype tree with nat =
    {n:nat} Node(n) of int(n) * int * treelist(n)

and treelist with nat =
    Tempty(0)
  | {m:nat}{n:nat | m >= n} Tcons(m+1) of tree(m) * treelist(n) ;;

datatype heap with nat =
    Hempty(0)
  | {m:nat}{n:nat | n = 0 \/ m+1 < n} Hcons(m+1) of tree(m) * heap(n) ;;

let rank = function Node(r, _, _) -> r
withtype {n:nat} tree(n) -> int(n) ;;

let root = function Node(_, x, _) -> x
withtype {n:nat} tree(n) -> int ;;

let link (Node(r, x1, ts1) as t1) = function
    Node(_, x2, ts2) as t2 ->
    if (x1 <= x2) then Node(r+1, x1, Tcons(t2, ts1))
    else Node(r+1, x2, Tcons(t1, ts2))
withtype {r:nat} tree(r) -> tree(r) -> tree(r+1) ;;

let rec insTree t = function
    Hempty -> Hcons(t, Hempty)
  | Hcons(t', ts') as ts ->
    if rank t < rank t' then Hcons(t, ts) else insTree (link t t') ts'
withtype {r:nat}{n:nat | n = 0 \/ r < n}
         tree(r) -> heap(n) -> [l:nat | l > r] heap(l) ;;

let insert x hp = insTree (Node(0, x, Tempty)) hp
withtype int -> [n:nat] heap(n) -> [n:nat | n > 0] heap(n) ;;

let rec merge = function
    (hp1, Hempty) -> hp1
  | (Hempty, hp2) -> hp2
  | (Hcons(t1, hp1') as hp1), (Hcons(t2, hp2') as hp2) ->
    if rank t1 < rank t2 then Hcons(t1, merge(hp1', hp2))
    else if rank t1 > rank t2 then Hcons(t2, merge(hp1, hp2'))
    else let hp = merge(hp1', hp2') in insTree (link t1 t2) hp
withtype {m:nat}{n:nat} heap(m) * heap(n) ->
         [l:nat | (n = 0 /\ l = m) \/ (m = 0 /\ l = n) \/
                  (l >= min(m, n) > 0)] heap(l) ;;
```

Figure 7: An implementation of binomial heapin de Caml (I)

```
exception Heap_is_empty ;;

let rec removeMinTree = function
    Hempty -> raise Heap_is_empty
  | Hcons(t, Hempty) -> (t, Hempty)
  | Hcons(t, hp) ->
    let (t', hp') = removeMinTree hp in
    if root t < root t' then (t, hp) else (t', Hcons(t, hp'))
withtype {n:nat}
         heap(n) ->
         [r:nat][l:nat | l = 0 \/ l >= n > 0] (tree(r) * heap(l)) ;;

let findMin hp = let (t, _) = removeMinTree hp in root t
withtype {n:nat} heap(n) -> int ;;

let rec to_heap hp = function
    Tempty -> hp
  | Tcons(t, ts) -> to_heap (Hcons(t, hp)) ts
withtype {m:nat}{n:nat | m = 0 \/ m > n}
         heap(m) -> treelist(n) -> heap ;;

let deleteMin hp =
  let (Node(_, x, ts), hp) = removeMinTree hp
  in merge (to_heap Hempty ts, hp)
withtype heap -> heap ;;
```

Figure 8: An implementation of binomial heap in de Caml (II)

We recommend that the programmer avoid complex encodings when using dependent datatypes to capture invariants in data structures.

6 Related Work

The use of type systems in program error detection is ubiquitous. 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. Our work can be viewed as providing a more expressive type system to allow the programmer to capture more program properties through types and thus catch more errors at compile-time. As a consequence, types 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.

In [21], we have compared our work with some traditional dependent type systems such as the ones underlining Coq [8] and NuPrl [5], which are far more refined than the type system of DML. There, we also give comparison to the notion of indexed types [25] (an earlier version of which is described in [24]), the notion of refinement types [9, 7], the notion of sized types [13], and the programming language Cayenne [1].

There have been many recent studies on the use of nested datatypes [2] in constructing (sophisticated) datatypes to capture more invariants in data structures. For instance, a variety of examples can be found in

[3, 18, 10, 12, 11]. 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 much 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 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.

7 Conclusion

The use of dependent datatypes in capturing invariants in data structures is novel. This practice can offer many advantages when we implement algorithms in advanced programming languages equipped with such a mechanism. The most significant advantage is probably in program error detection. We argued in Section 1 that the imprecision of datatypes in Standard ML or Haskell in capturing invariants 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. There are also various uses of dependent datatypes in compiler optimization.

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 how to report error messages in case of type errors. The type-checking in de Caml 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. On the positive side, the type-checker of de Caml is often capable of detecting a variety of subtle errors. For instance, the author once used `Even(l1, l2)` to form a random-list (in Figure 3) and the type-checker raised an error because it could not prove that `l1` 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 gathering more statistics regarding the use of de Caml.

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, enhancing the robustness of programs. 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.

8 Acknowledgment

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

References

[1] Lennart Augustsson. Cayenne – a language with dependent types. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, 1998.

[2] Richard Bird and Lambert Meertens. Nested datatypes. In *Mathematics of program construction*, pages 52–67. Springer-Verlag LNCS 1422, 1998.

[3] Richard Bird and Ross Paterson. de bruijn notation as a nested datatypes. *Journal of Functional Programming*, To appear.

[4] W. Braun and M. Rem. A logarithmic implementation of fexible arrays. Technical Report Memorandum MS83/1, Eidenhoven University of Technology, 1983.

[5] Robert L. Constable et al. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[6] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1989.

[7] Rowan Davies. Practical refinement-type checking. Thesis Proposal, November 1997.

[8] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[9] Tim Freeman and Frank Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277, Toronto, Ontario, 1991.

[10] Ralf Hinze. Numerical Representations as Higher-Order Nested Types. Technical Report IAI-TR-98-12, Institut für Informatik III, Universität Bonn, April 1998.

[11] Ralf Hinze. Constructing Red-Black Trees. In *Proceedings of Workshop on Algorithmic Aspects of Advanced Programming Languages*, September 1999.

[12] Ralf Hinze. Manufacturing Datatypes. In *Proceedings of Workshop on Algorithmic Aspects of Advanced Programming Languages*, September 1999. Also available as Technical Report IAI-TR-99-5, Institut für Informatik III, Universität Bonn.

[13] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of 23rd ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 410–423, 1996.

[14] INRIA. Caml-light. `http://caml.inria.fr`.

[15] Robin Milner, Mads Tofte, Robert W. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1997.

[16] Chris Okasaki. Three Algorithms on Braun Trees by Chris Okasaki. *Journal of Functional Programming*, 7(6):661–666, November 1997.

[17] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[18] Chris Okasaki. From Fast Exponentiation to Square Matrices: An Adventure in Types. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, September 1999.

[19] Simon Peyton Jones et al. Haskell 98 – A non-strict, purely functional language. Available from `http://www.haskell.org/onlinereport/`, February 1999.

[20] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[21] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

[22] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. pp. viii+189. Available as `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

[23] Hongwei Xi. Some programming examples in de Caml. Available at `http://www.cse.ogi.edu/~hongwei/DML/deCaml/examples/`, 1999.

[24] Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.

[25] Christoph Zenger. *Indizierte Typen*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 1998. Forthcoming.