Facilitating Program Verification with Dependent Types

Hongwei Xi
Computer Science Department
Boston University
hwxi@cs.bu.edu

Abstract

The use of types in capturing program invariants is overwhelming in practical programming. The type systems in languages such as ML and Java scale convincingly to realistic programs but they are of relatively limited expressive power. In this paper, we show that the use of a restricted form of dependent types can enable us to capture many more program invariants such as memory safety while retaining practical type-checking. The programmer can encode program invariants with type annotations and then verify these invariants through static type-checking. Also the type annotations can serve as informative program documentation, which are mechanically verified and can thus be fully trusted. We argue with realistic examples that this restricted form of dependent types can significantly facilitate program verification as well as program documentation.

1 Introduction

The verification of program correctness with respect to specification is a highly significant problem that is ever-present in programming. A great number of approaches have been developed to address the problem, but they are often too expensive to be put into software practice [10, 6]. On the other hand, the verification of the *type correctness* of programs, that is, type-checking, in languages such as ML [18] and Java [1] scales convincingly in practice. However, we must note that the types in programming languages such as ML and Java are of relatively limited expressive power. Therefore, we are naturally led to form type systems in which more sophisticated properties can be captured and then verified through type-checking.

A heavy-weighted approach is to adopt a type system in which highly sophisticated properties on programs can be expressed. For instance, the type system of NuPrl [3] based on Martin-Löf's constructive type theory [16] is such a case. In such a type system, even the (total) correctness of a program can be expressed in types and thus verified through type-checking. For instance, the reader can find a solid application in [15]. However, there is a steep price to pay for such an expressive type system. Type-checking in this setting usually involves a great deal of theorem proving and readily becomes intractable to automate[1]. Also a programming language with such a type system is often kept *pure* because it is at least unwieldy, if not impossible, to combine many realistic programming features with type systems similar to that of NuPrl. This is shown in the works such as allowing unlimited recursion [4], introducing recursive types [17], and incorporating effects [11], exceptions [19] and input/output. This is essentially an approach which strongly favors expressiveness over scalability.

We adopt a light-weighted approach, introducing a notion of restricted form of dependent types, where we clearly separate type index expressions from run-time expressions. In functional programming, we have enriched the type system of ML with such a form of dependent types, leading to a functional programming language DML (Dependent ML) [29, 23]. In imperative programming, we have designed a programming language *Xanadu* with C-like syntax to support such a form of dependent types [24].

We give some concrete examples before going into further details. As a dialect of DML, de Caml (dependent Caml) is currently implemented on top of Caml-light, a version of an ML-dialect Caml [22]. In de Caml, the type system supports dependent types for capturing many properties that are beyond the reach of the type system of ML. For instance, the program in Figure 1 implements a copy function on integer arrays in de Caml. The function `vect_length` computes the length of a vector, and `v.(ind) <- u.(ind)` means that we update the i^{th} cell in vector v with the value stored in the i^{th} cell in vector

[1]We point out that NuPrl is primarily a logic rather than a programming language and it is therefore natural to perform interactive theorem proving during type-checking.

```
let intcopy (u, v) =
  for ind = 0 to vect_length(u) - 1 do
    v.(ind) <- u.(ind)
  done
withtype
  {m:nat,n:nat | m <= n}
    int vect(m) * int vect(n) -> unit
```

Figure 1. A copy function in de Caml

u, where i is the value of `ind`. The novelty in the code is the `withtype` clause, which is a type annotation supplied by the programmer. The meaning of this type annotation is straightforward: for all natural numbers m and n satisfying $m \leq n$, the function `intcopy` accepts a pair of integer vectors of sizes m and n, respectively, and it returns no value. Note that the syntax `int vect(m)` stands for a dependent type for all integer vectors of length m.

In Xanadu, the `intcopy` function can be implemented as follows, where we use the function `arraysize` for computing the length of an array.

```
{m:nat, n:nat | m <= n}
unit intcopy(int u[m], int v[n]) {
  var: int ind ;;

  invariant: [i:nat] (ind: int(i))
  for (ind=0; ind < arraysize(u); ind=ind+1) {
    v[ind] = u[ind];
  }
}
```

We use the keyword `var` to start a variable declaration, which ends with double semicolon `;;`. The keyword `invariant` indicates that a program invariant follows. In this case, the invariant, which can help prove that all array subscripting operations in `intcopy` are safe (i.e., cannot be out-of-bounds), states that `ind` stores an integer of value i for some natural number i. Clearly, this simple invariant can be readily synthesized from the loop following it. However, we will soon show with realistic examples that automatic invariant synthesis in practice seems largely intractable.

It can be burdensome for the programmer to write annotations. Therefore, we do not force the programmer to do so. For instance, the above code can still compile without the invariant annotation, but run-time array bound checking is then needed to ensure memory safety. As it is a good practice to write comments in code, we feel that it is also the case to use dependent types in code and write type annotations. It is well understood in software engineering that the properties captured by types can be of great use for program documentation. Since the type annotations in programs are mechanically verified through type-checking, they can be fully trusted. On the other hand, if we simply require that the programmer write comments in programs, it seems less likely to guarantee the correctness of the comments. It is fairly common in practice to encounter the "code-changes-but-the-comments-stay-the-same" symptom.

The main contribution of the paper is the identification of a restricted form of dependent types in program verification and the presentation of some supporting examples. While it seems prohibitively expensive to prove the correctness of realistic programs, we present examples to show that there are many interesting program properties such as memory safety which can be practically verified. The use of dependent types in program verification is not new, but the use of a restricted form of dependent types in realistic programming is new. Up to now, it is at least rare to see the use of dependent types in practical programming and it is highly unclear as to what dependent types can actually do in practice. Therefore, we feel that the presentation of some concrete examples involving the use of dependent types is an indispensable step towards making dependent types available in practical programming as the theoretical work on dependent types alone simply seems not enough. In general, we claim that we have made some significant progress in advocating the use of light-weighted formal methods in software development.

In this paper, we are intentionally to avoid presenting the (intimidating) theoretical details on the dependent type systems of DML and Xanadu as much as possible, striving for giving a clean and intuitive introduction to the use of dependent types in practical programming. In particular, we present program examples in functional programming as well as in imperative programming. This not only makes it easier to reach a broader audience, but also yields some evidence to further support the usefulness of a restricted form dependent types. For the reader who prefers to learn more about the theoretical details, please refer to [29, 23, 24].

We organize the paper as follows. In Section 2, we present a brief overview on the form of dependent types we use to capture program invariants. In Section 3, we illustrate with short program examples some unique and significant programming features in de Caml and Xanadu. We then present some realistic examples in Section 4, illustrating the kind of properties that can be captured in the type systems of de Caml and Xanadu. Finally, we mention some related work and conclude.

2 Dependent Types

In this section, we present a brief explanation on the dependent types in DML. The dependent types in Xanadu are formed similarly except that function types are restricted since Xanadu does not support curried functions. 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.

Generally speaking, dependent types are types that depend on the values of language expressions. For instance, we may form a type $int(i)$ for each integer i to mean that

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

Figure 2. The syntax for type index expressions

every integer expression of this type must have value i, that is, $\mathit{int}(i)$ 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 that can be chosen as type index expressions. A 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 represent integers. We present the syntax for type index expressions in Figure 2, where we use a for type index variables and c for fixed integers. Note that the language for type index expressions is typed, and 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 $\{a : \mathit{int} \mid a \geq 0\}$.

Types in DML are formed as follows. We use α for type variables, δ for type constructors and $\mathbf{1}$ for the unit type.

$$\begin{array}{rccl} \text{types} & \tau & ::= & \alpha \mid (\tau_1, \ldots, \tau_n)\delta(i) \mid \\ & & & \mathbf{1} \mid \tau_1 * \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \\ & & & \Pi a:\gamma.\tau \mid \Sigma a:\gamma.\tau \end{array}$$

For instance, `list` is a type constructor and $(\texttt{int})\texttt{list}(n)$ stands for the type for integer lists of length n. The type expressions $\Pi a : \gamma.\tau$ and $\Sigma a : \gamma.\tau$ form a universal dependent type and an existential dependent type, respectively. For instance, the universal dependent type $\Pi a : \mathit{nat}.(\texttt{int})\texttt{list}(a) \rightarrow (\texttt{int})\texttt{list}(a)$ 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. We can use the existential dependent type $\Sigma a : \mathit{nat}.(\texttt{int})\texttt{list}(a)$ to mean an integer list of some unknown length. We show 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 the logic framework LF [9] or the theorem prover NuPrl [3]). 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:

$$\frac{}{\phi \vdash \alpha \doteq \alpha} \qquad \frac{}{\phi \vdash \mathbf{1} \doteq \mathbf{1}}$$

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

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

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

For the last two rules, we require that a have no free occurrences in ϕ. 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 $\phi \vdash (\texttt{int})\texttt{list}((a+n)+1) \equiv (\texttt{int})\texttt{list}(m+n)$.

We can form the erasure $\|\tau\|$ of a dependent type τ by removing from τ all syntax related to type index expressions. This can be formally defined as follows.

$$\|\alpha\| = \alpha \qquad \|\mathbf{1}\| = \mathbf{1}$$

$$\|(\tau_1, \ldots, \tau_n)\delta(i)\| = (\|\tau_1\|, \ldots, \|\tau_n\|)\delta$$

$$\|\tau_1 * \tau_2\| = \|\tau_1\| * \|\tau_2\| \qquad \|\tau_1 \rightarrow \tau_2\| = \|\tau_1\| \rightarrow \|\tau_2\|$$

$$\|\Pi a:\gamma.\tau\| = \|\tau\| \qquad \|\Sigma a:\gamma.\tau\| = \|\tau\|$$

We can then relate a type erasure to a dependent type by interpreting $(\tau_1, \ldots, \tau_n)\delta$ as $\Sigma a : \gamma.(\tau_1, \ldots, \tau_n)\delta(a)$, where γ is a sort associated with δ when δ is declared. For instance, `int` is interpreted as $\Sigma a : \mathit{int}.\texttt{int}(a)$, Also, $(\alpha)\texttt{list}$, the polymorphic type for lists, is interpreted as $\Sigma a : \mathit{nat}.(\alpha)\texttt{list}(a)$ in the following presentation, meaning a list with unknown length.

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 [23]. Also more details on Xanadu can be found in [24].

3 Unique Programming Features

In this section, we use examples to present some unique and significant features in de Caml and Xanadu. These features will be needed in Section 4 for studying more sophisticated examples.

The programmer often declares datatypes when programming in ML. For instance, the following datatype declaration in Caml-light defines a polymorphic datatype `'a list` for modeling lists.

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

However, the declared type `'a list` is imprecise. 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 syntax `refine 'a list with nat` means that we refine the type `'a list` with an index of sort `nat`, that is, the index represents a natural number. In this case, the index stands for the length of a list.

- The syntax `nil(0)` means that the constructor `nil` is assigned type `'a list(0)`, that is, it is a list of length 0.
- The following syntax

  ```
  {n:nat} cons(n+1) of 'a * 'a list(n)
  ```

  indicates that `cons` is assigned the following 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 that `{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 reverse append function on lists. We use `[]` for `nil` and `::` as the infix operator for `cons`.

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

The function is defined with pattern matching, an attractive feature in many functional programming languages. For example, the first matching clause `([], ys) -> ys` means that `revApp` returns list `ys` when given a pair of lists `[]` and `ys`. 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 length m and n, respectively. We now present some informal description about type-checking in this case.

For the first clause `([], ys) -> ys`, the type-checker assumes that `ys` is of type `'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, namely the following one.

```
(x :: xs, ys) -> revApp (xs, x :: 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 `revApp (xs, x :: 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 the assumption that a and b are natural numbers. This finishes type-checking the above de Caml program. The interested reader can always see [23] for the formal presentation of type-checking in DML. When type-checking a program in de Caml, we currently only solve linear constraints on integers, rejecting non-linear ones. This practice leads to a decidable type-checking algorithm.

Instead of refining a type, it is also allowed to declare a dependent datatype 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 dependent type theory. This provides a smooth interaction between ML types and dependent types. Suppose that function 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.

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 a given property `p`.

```
let filter p = function
    nil -> nil
  | 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 result of `(filter p l)` without knowing what `p` and `l` are. Therefore, it is impossible to type the function using only universal dependent types. Nonetheless, we know that the length of the result of `(filter p l)` is less than or equal to that of `l`. 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)
```

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

Another significant use of existential dependent types is to represent a range of values. We can use the syntax `([n:nat] int(n)) vect` to represent a type for the vectors whose elements are natural numbers. This is useful, for instance, to eliminate run-time array bound checks [28]. 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.

We now turn our attention to the programming language Xanadu. Probably, the most significant feature in Xanadu, which we think is also novel, is that the type of a variable in a program is allowed to change during program execution. For instance, after the assignment `x = 1` is done, the type of `x` becomes `int(1)`, reflecting that an integer equal to 1 is stored in x. Suppose we now execute `x = x+1`. This assignment changes the type of `x` into `int(2)` since `+` is assigned the following type

```
{i:int,j:int} int(i) * int(j) -> int(i+j)
```

```
('a){m:nat, n:nat}
<'a> list(m+n)
revApp (xs: <'a> list(m), ys: <'a> list(n)) {
  var: 'a x;
  invariant:
    [m1:nat, n1:nat | m1+n1 = m+n]
    (xs: <'a> list(m1), ys: <'a> list(n1))
  while (true) {
    switch (xs) {
      case Nil: return ys;
      case Cons (x, xs): ys = Cons(x, ys);
    }
  exit;
  }
}
```

Figure 3. An implementation of the reverse append function on lists in Xanadu

in Xanadu. Let us now execute `x = factorial(x)`, where `factorial` is given the type `(int) -> int`. This changes the type of `x` into `int` rather than `int(2)`.

In Xanadu, one can declare union types, which correspond to datatypes in ML. For instance, the following corresponds to the previous datatype declaration in de Caml[2].

```
union <'a> list with nat = {
  Nil(0);
  {n:nat} Cons(n+1) of 'a * <'a> list(n);
}
```

A type $(\tau_1, \ldots, \tau_n) \rightarrow \tau$ in Xanadu is for a function that takes n arguments of types $\tau_1, \ldots, \tau_n$, respectively, and returns a result of type τ. In Figure 3, the reverse append function on lists is implemented in Xanadu. The function header means that `revApp` has the following type.

```
{m:nat,n:nat}
  (<'a> list(m), <'a> list(n)) -> <'a> list(m+n)
```

The invariant in the implementation states that xs and ys are always lists of lengths $m1$ and $n1$, respectively, at the beginning of the loop, where $m1 + n1 = m + n$ holds. With this, it can be verified in the type system of Xanadu that `revApp` takes two lists as arguments and, if it returns, returns a list whose length is the sum of the lengths of two arguments.

The `switch` statement in the implementation uses pattern matching. For instance, if the following clause

```
case Cons (x, xs): ys = Cons(x, ys)
```

is chosen at run-time, the head and tail of xs are assigned to x and xs, respectively, and $Cons(x, ys)$ is assigned to ys, and then the loop repeats.

[2]We require that constructor names in Xanadu begin with capital letters for parsing purpose. The brackets `<>` in `<'a> list` are also for parsing purpose.

In Xanadu, it is also allowed to declare dependent record types. For instance, a record type for a heap can be declared as follows,

```
{n:nat} record <'a> heap(n) {
  max: int(n);
  /* int[0,n] is the type for *
   * integers between 0 and n */
  size: int[0,n];
  data[n]: 'a
}
```

where `max` is for the maximum heap size, `size` is for the actual heap size and `data` is for heap elements. We use `int[i,j]` as an abbreviation for the syntax `[a:int | i <= a <= j] int(a)`. Hence, `int[0,n]` is the type for all integers between 0 and n, inclusive. The declaration states that for every type `'a` and every natural number `n`, we can form a type `<'a> heap(n)`; this type contains three components `max`, `size` and `data`; the types of these components indicate that `max` can only store an integer equal to `n`, `size` can only store a natural number less than or equal to `n` and `data` is an array of size `n` in which each element is of type `'a`.

In theory, we have the freedom to assign a value of any type to a variable in Xanadu, but we find it a good practice to impose some restrictions on this freedom. We assign each variable x in Xanadu a type τ that is called the *master type* of x; if an assignment $x = e$ occurs in Xanadu for some expression e, we must prove that e can be coerced into an expression of type τ. The notion of coercion is closely related to subtyping. In particular, if $\tau = \Sigma a : \gamma.\tau_1$ and e has type $\tau_1[a := i]$ and i has sort γ, then we say that e can be coerced into an expression of type τ, where $\tau_1[a := i]$ is the result of substituting i for a in τ_1. For instance, every expression of type $\texttt{int}(i)$ can be coerced into an expression of type `int`, which is a short hand for $\Sigma a : int.\texttt{int}(a)$.

For a variable appearing as an argument in a function declaration, the master type of the variable is the type erasure of the type of the argument. On the other hand, for a variable declared in the body of a function, the type declared for the variable is the master type of the variable. For instance, the master types of variables `xs` and `ys` in Figure 3 are `<'a>list` and the master type for `x` there is `'a`.

We refer the reader to [26] for other programming language features in Xanadu.

4 Examples

In this section, we present several examples in Xanadu to show the use of a restricted form of dependent types in capturing program properties. For the sake of limited space, some presented examples are incomplete. We refer the reader to [26] for complete versions of these examples and many other examples. Also, we stress that all these examples have been verified in the current prototype implementation of Xanadu.

```
{n:nat}
int bsearch(key: float, vec[n]: float) {
  var: int low, mid, high;
       float x;;

  low = 0; high = arraysize(vec) - 1;

  invariant:
    [i:int, j:int | 0 <= i <= n, 0 <= j+1 <= n]
    (low: int(i), high: int(j))
  while (low <= high) {
    mid = (low + high) / 2;
    x = vec[mid];
    if (key ==. x) { return mid; }
    else if (key <. x) { high = mid - 1; }
         else { low = mid + 1; }
  }
  return -1;
}
```

Figure 4. Binary Search

4.1 Binary Search

We present the implementation of a binary search function on an array of floating point numbers in Figure 4, where `key ==. x` (`key <. x`) tests whether `key` is equal to (less than) `x`. In this implementation, we can prove in the type system of Xanadu that the value of `mid` is always within the bounds of `vec` when `x = vec[mid]` is evaluated. This, with the type safety of the program, guarantees that the execution of the program will never cause memory violations. Therefore, the memory safety of the program has been statically verified.

In this example, the invariant given ahead of the `while` loop is needed for proving the safety of the array subscripting operation in the loop, which basically states that there exist integers i and j satisfying $0 \leq i \leq n$ and $0 \leq j + 1 \leq n$ such that `low` and `high` equal i and j, respectively, at the entry point of the loop. Obviously, a natural question is whether such an invariant can be synthesized from the structure of the program. We are less enthusiastic about synthesizing invariants for essentially two reasons; in general, it seems highly intractable to synthesize such invariants in realistic programming; also we encourage the programmer to write type annotations since they can serve as informative program documentation.

4.2 Sparse Array Multiplication

A polymorphic record `<'a>sparseArray(m,n)` is declared in Figure 5 for representing two-dimensional sparse arrays of dimension `m` by `n` in which each element is of type

```
{m:nat,n:nat}
record <'a> sparseArray(m, n) {
  row: int(m);
  col: int(n);
  data[m]: <int[0,n) * 'a> list
}

{n:nat}
float
list_vec_mult (xs:: <int[0,n) * float> list,
               vec[n]: float) {
  var: int i; float f, sum;;

  sum = 0.0;
  while (true) {
    switch (xs) {
      case Nil: return sum;
      case Cons((i, f), xs):
        sum = sum +. f *. vec[i];
    }
  }
  exit;
}

{m:nat,n:nat}
<float> array(m)
mat_vec_mult (mat: <float> sparseArray(m,n),
              vec[n]: float) {
  var: nat i; float result[];;

  result = newarray(mat.row, 0.0);

  for (i = 0; i < mat.row; i = i + 1) {
    result[i] = list_vec_mult (mat.data[i], vec);
  }
  return result;
}
```

Figure 5. Sparse Array Multiplication

`'a`. Let `r` be a record of the type `<'a>sparseArray(m, n)`. Then `r` has three components, namely, `row`, `col` and `data`. Clearly, the types assigned to `row` and `col` indicate that `r.row` and `r.col` return the dimensions of `r`, and the type assigned to `data` states that `r.data` is an array of size m, in which each element is a list of pairs that represents a row in a sparse array and each pair consists of a natural number less than n and an element of type `'a`. Note that we use `int[0,n)` as an abbreviation for the syntax `[i:int | 0 <= i < n]`, that is, it is the type for all natural numbers less than n. For instance, a list consisting of two pairs $(6, 2.7183)$ and $(23, 3.1416)$ represents a row in a sparse array where the 6th and 23rd elements are 2.7183 and 3.1416, respectively and the rest are 0.0.

The function `list_vec_mult` computes the dot product of a row in a sparse array and a given vector, and the function `mat_vec_mult` multiplies a given vector by a sparse matrix and then returns a vector as the result. Notice that all array subscripting here is proven safe in the type system of Xanadu.

```
{size:nat, s:nat, e:nat | s < e <= size}
[n:nat | s <= n < e] int(n)
split (start: int(s),
       end: int(e),
       vec[size]: float) {
  var: int pivot; float x, tmp;
       l: int[s, e]; h: int[s, e);;

  pivot = get_pivot(start, end, vec);
  l = start; h = end - 1; x = vec[pivot];

  while (true) {
    while (l < end) {
      if (vec[l] <=. x) l = l + 1; else break;
    }
    while (start < h) {
      if (vec[h] >. x) h = h - 1; else break;
    }
    if (l < h) {
      tmp = vec[l];
      vec[l] = vec[h];
      vec[h] = tmp;
      l = l + 1; h = h - 1;
    } else return h;
  }
  exit; /* can never be reached! */
}
```

Figure 6. Quicksort on Arrays

4.3 Quicksort on Arrays

We can implement quicksort on arrays in Xanadu and show in the type system of Xanadu that this implementation is memory safe. We present in Figure 6 a key function `split` in the implementation, which uses a pivot value returned by `get_pivot` to perform array rearrangement.

The header of the function `split` states that for natural numbers $size$, s and e satisfying $s < e \leq size$, `split` takes an integer equal to s and another integer equal to e and an integer array of size $size$, and then returns an integer n satisfying $s \leq n < e$. In Xanadu, we also use the syntax $x : \tau$ to declare that x is a variable of type τ. For instance, the syntax `l: int[s,e]`, where `int[s,e]` is an abbreviation for `[i:int | s <= i <= e] int(i)`, declares that `l` is an integer variable which can only store an integer i satisfying $s \leq i \leq e$. Similarly, the syntax `h: int[s, e)`, where `int[s, e)` is an abbreviation for `[i:int | s <= i < e] int(i)`, means that `h` can only store an integer i satisfying $s \leq j < e$.

It can be verified in the type system of Xanadu that every array subscripting operation in the body of `split` is safe, that is, each subscript used is within the bounds of the subscripted array.

```
/*
 * var: float px[n+1], py[n+1];
 *      i: [i:nat] int(i) ;;
 */

j = 1;

invariant: [j:int | 1 <= j <= n] (j: int(j))
for (i = 1; i < n; i++) {

  if (i < j) {
    xt = px[j]; px[j] = px[i]; px[i] = xt;
    xt = py[j]; py[j] = py[i]; py[i] = xt;
  }

  k = n / 2;

  invariant: [k:nat | 2 * k <= n] (k: int(k))
  while (k < j) { j = j - k; k = k / 2; }

  j = j + k;
}
```

Figure 7. Bit reversing in Xanadu

4.4 FFT in Xanadu

We can also give an implementation of FFT in Xanadu and verify that the implementation is memory safe. This is an involved example in which loops are embedded up to the fourth level (counting the outermost loop level 1). We present in Figure 7 a fragment of the implementation which performs bit reversing in FFT, where we have already declared `px` and `py` as floating point arrays of length $n+1$, i as a natural number and j as an integer between 0 and n. The types of `i` and `j` guarantee that all array subscripting in bit reversing is safe.

Note the occurrence of the syntax `k: int(k)` in the invariant: *the first* `k`*, a run-time variable in Xanadu, is different from the second* `k`*, which is a type index variable.* We informally explain how the assignment `j = j + k` is type-checked, using j, k, n for the integer values stored in `j,k,n`; $k < j$ is obviously false when this assignment is evaluated since it is right after the `while` loop with loop condition `k < j`; also, the loop invariant indicates that $2 * k \leq n$ holds at this point; therefore, `j+k` has a value between 0 and n, which is allowed to be assigned to `j` according the type of `j`.

4.5 More Examples

Many more programming examples involving the use of dependent types are available at [25] and [26], including implementations of binary heap, heapsort, Gaussian elimination, etc. A distribution of de Caml, which is implemented on top of Caml-light [13], is available at [25]. The implementation of Xanadu is ongoing and the current code is available at [26].

5 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 of relatively limited expressive power for the sake of practical type-checking. The idea of using types as means for verification can be traced back at least to Milner: *well-typed programs can't go wrong*. However, this is true in ML only if one admits that "going wrong" does not include raising uncaught exceptions. In general, the use of types in program verification is effective but too limited.

Our work falls in between full program verification, either in type theory or systems such as PVS [20], and traditional type systems for programming languages. When compared to verification, our system is less expressive but more automatic. Our work can be viewed as providing a systematic and uniform language interface for a verifier intended to be used as a type system during the program development cycle. Our primary motivation is to allow the programmer to express more program properties through types and thus catch more errors at compile-time. In addition, types can serve as informative program documentation, facilitating program comprehension.

There are already some type systems such as the ones underlying Coq [7] and NuPrl [3], which are far more refined than the type system of DML. However, type-checking in these type systems is interactive and may often become a daunting task for programmers. Since minor changes in a program may often mean a major change in a proof (of its well-typedness) and there are many changes to be made during the program development cycle, the cost in effort and time often deters the user from programming in such a setting.

The notion of qualified types in [14] provides a general framework for the combination of polymorphism and overloading. In the presence of qualified types, type-checking involves constraint satisfaction. The Damas/Milner type inference algorithm is extended to support qualified types, which in turn specifies the set of all possible types for any term. In DML, the notion of principal types is lost. Instead, we adopt a bi-directional type inference algorithm, requiring the programmer to supply types annotations for functions with dependent types. Therefore, as far as type-checking is concerned, there is little overlapping between qualified types and the DML-style dependent types.

The work on extended static checking (ESC) [5] also emphasize the use of formal annotations to capture program invariants. These invariants can then be verified through (light-weighted) theorem proving. ESC is developed on top of the imperative programming language Modula-3. It provides a specification language for the programmer to specify properties including a list of variables that a procedure may modify, a precondition that must be satisfied before a function call, a postcondition that must hold when a function terminates, etc. A significant difference between ESC and DML is that the former takes an approach based on first-order logic assertions while the latter adopts a type-based one. Further study is needed to determine whether ESC can be readily extended to handling higher-order functions.

We have also designed a dependently typed assembly language (DTAL) in which a restricted form of dependent types can capture the memory safety property of assembly code [27]. The design of Xanadu is partly motivated by a need for generating DTAL code from source level programs.

The theoretical development of DML is presented in [29], but it is less clear to the reader how the developed theory can be applied in practice. In this paper, some realistic examples in de Caml are given to address this question. Also, it is shown with various examples in Xanadu that the restricted form of dependent types in DML can also be applied to imperative programming. This gives the research a potential to reach a (much) broader audience than [29].

It should be stressed that 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. This is also true in Xanadu. We deliberately choose binary search, quicksort and FFT as our examples because of the difficulty in proving these programs to be memory safe. We see it less relevant to simply try large programs: if a program does not use dependent types, no constraints are generated from type-checking the program no matter how large it is.

In [29], we have already compared our work with the notion of refinement types [8, 21], the notion of indexed types [31] (an earlier version of which is described in [30]), the notion of sized types [12], and the programming language Cayenne [2].

6 Conclusion

In general, it is still largely an elusive goal to verify the correctness of programs. Therefore, it is important to identify the properties that can be practically verified for realistic programs. We have shown with examples the use of a restricted form of dependent types in facilitating program verification and documentation. A large number of automated program verification approaches often focus on verifying sophisticated properties of some particularly chosen programs. We feel that it is at least equally important to study scalable approaches to verifying elementary properties of programs in general programming as we have advocated in this paper.

In practice, comments in programs often constitute a large portion of program documentation. It is a common scenario that comments are often outdated or simply wrong, and therefore cannot be fully trusted. With a dependent type system, we can capture many sophisticated program invariants through type annotations and mechanically verify them, yielding both correct and informative program documentation.

Dependent types can also facilitate compiler optimization. For instance, it is unnecessary to insert run-time array bound checks when we compile the binary search function in Figure 4 since the type system of Xanadu has already guaranteed the memory safety of the implementation. This can lead to some significant performance gains [28].

In general, we are interested in promoting the use of light-weighted formal methods in practical programming, enhancing the robustness of software. We have presented some benefits of dependent types in program verification and documentation in this paper in support of such a promotion. Also we would like to use Xanadu as an example to raise the awareness of the benefits of dependent types outside the functional programming community.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1997.

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

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

[4] R. L. Constable and S. F. Smith. Partial objects in constructive type theory. In *Proceedings of Symposium on Logic in Computer Science*, pages 183–193, Ithaca, New York, June 1987.

[5] D. Detlefs. An overview of the extended static checking system. In *Workshop on Formal Methods in Software Practice*, 1996.

[6] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[7] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Rapport Technique 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

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

[9] R. W. Harper, F. Honsell, and G. D. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[11] F. Honsell, I. A. Mason, S. Smith, and C. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, 15 May 1995.

[12] J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Conference Record of 23rd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '96)*, pages 410–423, 1996.

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

[14] M. P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, November 1994.

[15] C. Kreitz, M. Hayden, and J. Hickey. A proof environment for the development of group communication systems. In H. Kirchner and C. Kirchner, editors, *15th International Conference on Automated Deduction*, LNAI 1421, pages 317–332, Lindau, Germany, July 1998. Springer-Verlag.

[16] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, Italy, 1984.

[17] N. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proceedings of Symposium on Logic in Computer Science*, pages 30–36, Ithaca, New York, June 1987. The Computer Society of the IEEE.

[18] R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[19] H. Nakano. A constructive logic behind the catch and throw mechanism. *Annals of Pure and Applied Logic*, 69(2–3):269–301, October 1994.

[20] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV '96)*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag LNCS 1102.

[21] D. Rowan. Practical refinement-type checking. Thesis Proposal, November 1997.

[22] P. Weis and X. Leroy. *Le langage Caml*. InterEditions, Paris, 1993.

[23] H. 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`.

[24] H. Xi. Imperative Programming with Dependent Types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, pages 375–387, Santo Barbara, June 2000.

[25] H. Xi. Dependent ML. Available at `http://www.cs.bu.edu/~hwxi/DML/DML.html`, 2001.

[26] H. Xi. Xanadu: Imperative Programming with Dependent Types, 2001. Available at `http://www.cs.bu.edu/~hwxi/Xanadu/`.

[27] H. Xi and R. Harper. A Dependently Typed Assembly Language. Technical Report CSE-99-008, Oregon Graduate Institute, July 1999. Also available as `http://www.cs.bu.edu/~hwxi/academic/papers/DTAL.ps`.

[28] 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, Montréal, Canada, June 1998.

[29] 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, Texas, January 1999.

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

[31] C. Zenger. *Indizierte Typen*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 1998.

We present an implementation of Gaussian elimination in Figure 8 without further explanation, where the type system of Xanadu guarantees that the implementation is memory safe.

```
{m:nat, n:nat}
record <'a> matrix(m,n) {
  row: int(m); col: int(n); data[m][n]: 'a
}

('a){m:nat,n:nat,i:nat,j:nat | i < m, j < m}
unit rowSwap(data[m][n]:'a, i:int(i), j:int(j)) {
  var: temp[]: 'a;;
  temp=data[i]; data[i]=data[j]; data[j]=temp;
}

{n:nat,i:nat | i < n}
unit norm(r[n]: float, n: int(n), i: int(i)) {
  var: float x;;
  x = r[i]; r[i] = 1.0; i = i + 1;
  invariant: [i:nat] (i: int(i))
  while (i < n) { r[i] = r[i] /. x; i = i + 1;}
}

{n:nat, i:nat | i < n}
unit rowElim (r[n]: float, s[n]: float,
              n: int(n), i: int(i)) {
  var: float x;;
  x = s[i]; s[i] = 0.0; i = i + 1;
  invariant: [i:nat] (i: int(i))
  while (i < n) {
    s[i] = s[i] -. x *. r[i]; i = i + 1;
  }
}

{m:nat, n:nat, i:nat | m > i, n > i}
int[0, m)
rowMax (data[m][n]: float, m: int(m), i: int(i)) {
  var: nat j; float x, y; max: int[0, m);;
  /* fabs: absolute value function for float */
  max = i; j = i + 1; x = fabs(data[i][i]);
  while (j < m) {
    y = fabs(data[j][i]);
    if (y >. x) { x = y; max = j; }
    j = j + 1;
  }
  return max;
}

{n:nat | n > 0}
unit gauss (mat: <float> matrix(n,n+1)) {
  var: float data[][n+1]; nat i, j, max, n;;

  data = mat.data; n = mat.row;
  for (i = 0; i < n; i = i + 1) {
    max = rowMax(data, n, i);
    norm (data[max], n+1, i);
    rowSwap(data, i, max);
    for (j = i + 1; j < n; j = j + 1) {
      rowElim(data[i], data[j], n+1, i);
    }
  }
}
```

Figure 8. Gaussian Elimination