Eliminating Array Bound Checking Through Dependent Types*

Hongwei Xi
Department of Mathematical Sciences
Carnegie Mellon University
hwxi@cs.cmu.edu

Frank Pfenning
Department of Computer Science
Carnegie Mellon University
fp@cs.cmu.edu

Abstract

We present a type-based approach to eliminating array bound checking and list tag checking by conservatively extending Standard ML with a restricted form of dependent types. This enables the programmer to capture more invariants through types while type-checking remains decidable in theory and can still be performed efficiently in practice. We illustrate our approach through concrete examples and present the result of our preliminary experiments which support support the feasibility and effectiveness of our approach.

1 Introduction

The absence of run-time array bound checks is an infamous source of fatal errors for programs in languages such as C. Nonetheless, compilers offer the option to omit array bound checks, since they can turn out to be expensive in practice (Chow 1983; Gupta 1994). In statically typed languages such as ML, one would like to provide strong guarantees about the safety of all operations, so array bound checks cannot be omitted in general. The same is true for Java bytecode interpreters or compilers (Sun Microsystems 1995) and proof-carrying code (Necula 1997), which are aimed at providing safety when transmitting code across a network to be executed at a remote site.

Tag checking in functional languages is similar to array bound checking. For example, we can more efficiently access the tail of a list if we know that the list is non-empty. This kind of situation arises frequently in dynamically typed languages such as Scheme, but it also arises from the compilation of pattern matches in ML.[1]

Traditional compiler optimizations do not fare well in the task of eliminating redundant array bound checks, so some special-purpose methods have been developed for automated analysis (see, for example, (Markstein and Markstein 1982; Gupta 1994)). With some notable exceptions (see below) these methods try to infer redundant checks without programmer annotations and are thus limited by their ability to synthesize loop invariants—a problem that is in theory undecidable and in practice very difficult (Susuki and Ishihata 1977). In contrast, we pursue a *type-based approach* within a language already statically typed, namely ML. We rely on the programmer to supply some additional type information, which is then used by the compiler to reduce static array bound checking to constraint satisfiability. The constraints consists of linear inequalities and can be solved efficiently in practice.

This approach leads to several language design and implementation questions, to which this paper provides a possible answer. We have validated our ideas through a prototype implementation for a fragment of ML large enough to encompass several standard programs, taken from existing library code. Our experiments demonstrate that

- the required extended type annotations are small compared to the size of the program,
- the constraints which arise during extended type checking can be solved efficiently in practice, and
- the compiled code can be significantly faster.

Moreover, with one exception (where we had to replace on occurrence of $<$ by $\leq$) we did not have to modify the existing code, only extend it with some annotations.

Our approach is based on the notion of *dependent type* (Martin-Löf 1980) which allows types to be *indexed* by terms. For example, in ML we have a type of integer lists `int list`. Using dependent types we can express the more precise type of integer lists of length 2 as `int list(2)`. In this example, 2 is the *index object*. An function for appending two integer lists would have type `int list(`n`) -> int list(`m`) -> int list (`$n+m$`)` for any n and m. Unfortunately, without any restrictions on the form of index objects, automatic type-checking in a language with dependent types is undecidable and impractical. We avoid such problems through the combination of several important ideas:

- We separate the language of type indices from the language of terms. Among other things, this separation avoids the question of the meaning of effects in type indices and permits a clear *phase distinction* between type-checking and evaluation.

*This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533.

[1] We are not aware of any empirical study regarding its practical significance.

```
assert length <| {n:nat} 'a array(n) -> int(n)
and    sub <| {n:nat} {i:nat | i < n} 'a array(n) * int(i) -> 'a

fun dotprod(v1, v2) =
    let
        fun loop(i, n, sum) =
            if i = n then sum
            else loop(i+1, n, sum + sub(v1, i) * sub(v2, i))
        where loop <| {n:nat} {i:nat | i <= n} int(i) * int(n) * int  -> int
    in
        loop(0, length v1, 0)
    end
where dotprod <| {p:nat} {q:nat | p <= q } int array(p) * int array(q) -> int
```

Figure 1: The dot product function

- We employ *singleton types* (Hayashi 1991) to allow the necessary interaction between the index and term languages.
- We only consider programs which are already well-typed in ML. This allows our extension to be *conservative*, that is, without the use of dependent types, programs will elaborate and evaluate exactly as in ML.
- We use bi-directional type analysis to generate linear inequality constraints with a minimum of annotations.
- The resulting constraints can be solved efficiently in practice with a variant of Fourier's method (Pugh and Wonnacott 1992).

Besides the fact that programs run faster (which tends to be a strong motivator for programmers), our system enhances many of the benefits one derives from static typing. The dependent types help the programmer to think about the properties he expects to hold, and many (often trivial) errors can be detected early, during dependent type checking rather than at run-time. The dependent type annotations serve as formal and machine-checked documentation of program invariants which greatly aids maintainability throughout the software life-cycle. Dependent types allow program invariants and properties to be communicated and checked across module boundaries if they are included in signatures.[2]

Most closely related to our work is the work on *shape checking* by Jay and Sekanina (Jay and Sekanina 1996). They also pursue a language-based approach with a restricted form of dependent types. However, their language and programs are rather restricted and different from the kind of programs typically written in ML (including, for example, explicit shape conditionals). This allows them to perform shape analysis through a process of partial evaluation rather than constraint simplification, but it does not seem to interact well with general recursion. We believe that their approach is well-suited for languages such as NESL (Blelloch 1993), but that it is too restrictive to be practical for ML.

Dependent types also form the basis of general theorem proving and verified program development environments such as Coq (Dowek, Felty, Herbelin, Huet, Murthy, Parent, Paulin-Mohring, and Werner 1993), Nuprl (Constable et al. 1986), or PVS (Owre, Rajan, Rushby, Shankar, and Srivas 1996).

[2]We have not not yet explored this possibility in our prototype implementation, which is restricted to the core language.

Our work can be seen as an attempt to narrow the gap between full verification, which often only works for unrealistically small languages or is too time-consuming for practicing programmers, and static type systems for programming languages, which allow only a very restricted set of program properties to be expressed and checked. In this way, our work is also related to work on *refinement types* (Freeman 1994; Davies 1997) in which ML types are refined into finitely many inductively defined sorts.

This work is part of a larger effort to introduce dependent types over tractable constraint domains into ML (Xi 1998). In particular, the basic language architecture and the elaboration algorithm which generates index constraints do not depend on particular properties of linear arithmetic and can be used to capture other program invariants.

2 Preliminaries

In this section we sketch our type system and give some illustrative examples. Please see (Xi 1998) for a formal description.

2.1 An introductory example

The code in Figure 1 is an implementation of the dot product on integer arrays. Even automatic methods are able to eliminate array bounds checks for this example—we use it here to introduce the language, not to illustrate its full expressive power.

This example should be read as follows.

- `int(n)` is a built-in singleton type which contains only the integer n. The type `int` used later is the type of all integers.
- `'a array(n)` is a built-in polymorphic type of arrays of size n whose elements are of type `'a`.
- `length <| {n:nat} 'a array(n) -> int(n)` expresses that `length` is a function which, when given an array of size n yields an integer of type `int(n)` (which must therefore be equal to n). In a full language implementation, this would be a pervasive declaration; here, we assert it explicitly.
- `sub <| {n:nat} {i:nat | i < n} 'a array(n) * int(i) -> 'a` means that sub can only be applied to an array of size n and an integer i such that $i < n$ holds. It always yields a a value of type `'a`.

We use {n:nat} as an explicit universal quantifier or *dependent function type constructor*. Conditions may be attached, so they can be used to describe certain forms of *subset types*, such as {n:nat | i < n} in the example. The two "where" clauses are present in the code for type-checking purposes, giving the dependent type of the local tail-recursive function loop and the function dotprod itself. After type-checking the code, we are sure that the array accesses through sub cannot result in array bound violations, and therefore there is no need for inserting array bound checks when we compile the code. Similarly, if we use an array update function update with the following type,

```
update <| {n:nat} {i:nat  | i < n}
          'a array(n) * int(i) * 'a -> unit
```

then no array bound checks are needed at run-time.

Notice that we can also index lists (and not just arrays) by their lengths and declare

```
nth <| {l:nat} {n:nat | n < l}
        'a list(l) * int(n) -> 'a
```

thereby eliminating the need for list tag checks. Because of the similarity of our approach to eliminating array bound checks and list tag checks, we shall focus on the former in this paper.

2.2 The language of types

Type indices may be integer or boolean expressions of the form defined below. We use a to range over index variables.

$$
\begin{array}{rrcl}
\text{Integer index} & i,j & ::= & a \mid i+j \mid i-j \mid i*j \mid div(i,j) \\
 & & & \mid min(i,j) \mid max(i,j) \\
 & & & \mid abs(i) \mid sgn(i) \mid mod(i,j) \\
\text{Boolean index} & b & ::= & a \mid false \mid true \\
 & & & \mid i<j \mid i \le j \\
 & & & \mid i=j \mid i \ge j \mid i>j \\
 & & & \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\
\text{Index} & d & ::= & i \mid b
\end{array}
$$

We also use certain transparent abbreviations, such as $0 \le i < n$ which stands for $0 \le i \wedge i < n$.

A system of *dependent types* allows types to be indexed by terms. For the purpose of this paper, indices are restricted to the integer and boolean expressions given above, with the additional constraint of linearity. We have considered a more general language schema in (Xi 1998). We use δ for base types or basic *type families*, either built-in (such as int or array) or user-declared. α stands for type variables as usual.

$$
\begin{array}{rrcl}
\text{index sort} & \gamma & ::= & int \mid bool \mid \{a:\gamma \mid b\} \\
\text{types} & \tau & ::= & \alpha \mid (\tau_1,\ldots,\tau_n)\delta(d_1,\ldots,d_k) \\
 & & & \mid \tau_1 * \cdots * \tau_n \mid \tau_1 \to \tau_2 \\
 & & & \mid \Pi a:\gamma.\tau \mid \Sigma a:\gamma.\tau
\end{array}
$$

When a type constructor has no arguments or no indices, we omit the empty parentheses on the left or right or the constructor, respectively; when a product has no components we write unit.

The sort $\{a:\gamma \mid b\}$ stands for those elements of γ satisfying the boolean constraint b. We use *nat* as an abbreviation for $\{a:int \mid a \ge 0\}$. Also notice that a is universally quantified in $\Pi a:\gamma.\tau$ and existentially quantified in $\Sigma a:\gamma.\tau$.

In the concrete syntax, we use {a:g} t for $\Pi a:\gamma.\tau$ and [a:g] t for $\Sigma a:\gamma.\tau$. We can combine several quantifiers by separating the quantified variables by commas and directly attach a condition to the quantifier. So $\Pi a:\{a:\gamma \mid b\}.\tau$ can be written as {a:g | b} t. We took advantage of these shorthands in the dot product example above.

Our language extension is intended to encompass all of Standard ML. Our current prototype implementation includes recursion, higher-order functions, polymorphism (with a value restriction), datatypes, pattern matching, and arrays, but at present no exceptions or module-level constructs, which are left to future work. We believe that only the extension to modules involves non-trivial language design issues.

2.3 Built-in type families

We have built-in type families for integers, booleans and arrays.

- For every integer n, int(n) is a singleton type which only contains n.
- For false and true, bool(false) and bool(true) are singleton types which only contain false and true, respectively.
- For a natural number n, 'a array(n) is the type of arrays of size n whose elements are of type 'a.

Indices may be omitted in types, in which case they are interpreted existentially. For example, the type int array stands for $\Sigma n : nat.$int array(n), that is, an integer array of some unknown size n.

2.4 Refinement of datatypes

Besides the built-in type families int, bool, and array, any user-defined data type may refined by explicit declarations. An an example, consider the declaration of 'a list:

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

After this declaration, the constructor nil has type 'a list and :: is of type 'a * 'a list -> 'a list. The following declaration indexes the type of a list by a natural number representing its length.

```
typeref 'a list of nat
with nil <| 'a list(0)
   | :: <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)
```

The structure of the dependent types for the constructors nil and :: must match the corresponding ML types.

Figure 2 displays an implementation of the reverse function on lists. Notice that the type of reverse ensures that this function always returns a list of length n when given one of length n.

This illustrates the need for giving explicit types to local functions (rev, in this case), since they are often more general than the externally visible type (for reverse in this case) and cannot be synthesized automatically in general. However, no types need to be given for bound variables.

The next example illustrates the need for existentially quantified dependent types. The filter function removes all the elements in a list l which do not satisfy a given property p. Clearly, the length of the resulting list cannot be expressed as a type index since it depends on arbitrary computation, which is not permitted in type indices. Nonetheless, we know that the resulting list will be of the length less than

```
fun reverse(l) =
let
    fun rev(nil, ys) = ys
      | rev(x::xs, ys) = rev(xs, x::ys)
    where rev <| {m:nat} {n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)
in
    rev(l, nil)
end
where reverse <| {n:nat} 'a list(n) -> 'a list(n)
```

Figure 2: The reverse function for lists

or equal to that of the original list. This information can be incorporated into the type of the `filter` function through existentially quantified dependent types.

```
fun filter p nil = nil
  | filter p (x::xs) =
    if p(x) then x::(filter p xs)
    else filter p xs
where filter <| {m: nat}
                ('a -> bool) -> 'a list(m) ->
                [n:nat | n <= m] 'a list(n)
```

The result of this function has type `[n:nat | n <= m] 'a list(n)` which is concrete syntax for $\Sigma n : \{n : nat \mid n \leq m\}.\alpha\texttt{list}(n)$.

Existential types can also be used to express subset types (note that this is different from subset sorts ascribed to index variables). For instance, we can use `[i:int | 0 <= i+1 ] int(i)` to represent the type for integers which are greater than or equal to -1. This feature is exploited to eliminate array bound checks in the implementation of Knuth-Morris-Pratt string matching algorithm shown in Appendix A. A detailed description of the algorithm can be found in (Corman, Leiserson, and Rivest 1989). Notice that several array bounds checks in the body of `computePrefixFunction` cannot be eliminated. Elimination of these checks would require a representation of deep invariants of the algorithm which are not expressible in our type system.

Existential types are also used to interpret indexed types such as `int`, when used without an index. For example, `int` is interpreted as $\Sigma i : int.\texttt{int}(i)$ (or `[i:int] int(i)`, in concrete syntax). Thus existential types provide a smooth boundary between annotated and unannotated programs in the context of a larger implementation. For larger and more interesting examples, we refer the reader to (Xi 1997).

3 Elaboration

The elaboration process transforms a program written in the source language into an expression in an explicitly typed internal language, performing type-checking along the way. Since it is beyond the scope of the paper to present a detailed treatment of this process, we shall highlight a few major features through examples.

One can think of elaboration as a two-phase process. In the first phase, we ignore dependent type annotations and simply perform the type inference of ML. If the term is well-typed, we traverse it again in the second phase and collect constraints from the index expressions occurring in type families. Constraints are boolean index expressions b enriched with explicit quantifiers and implication. The latter is necessary for type-checking pattern matching expressions.

The syntax for the constraints is given as follows.

$$\text{Constraints} \quad \phi \quad ::= \quad b \mid \phi_1 \wedge \phi_2 \mid b \supset \phi \mid \exists a : \gamma.\phi \mid \forall a : \gamma.\phi$$

3.1 Generating constraints

The following is the auxiliary tail-recursive function in the implementation of the reverse function in Figure 2.

```
fun rev(nil, ys) = ys
  | rev(x::xs, ys) = rev(xs, x::ys)
where rev <| {m:nat} {n:nat}
             'a list(m) * 'a list(n) ->
             'a list(m+n)
```

Let us elaborate the clause `rev(nil, ys) = ys`. According to the form of the type assigned to `rev`, we introduce two index variables M and N, and check `nil` against type `'a list(`M`)` and `ys` against type `'a list(`N`)`.

This generates two constraints $M = 0$ and $N = \text{n}$, where `ys` is assumed to be of type `'a list(n)`. Then we check the type of the right hand side of the clause, `ys` against `'a list(`M`+`N`)`, the result type specified for `rev`. This yields the constraint $M + N = \text{n}$.

Thus analyzing the first clause in the definition of `rev` generates the constraint

$$\forall n : nat.\exists M : nat.\exists N : nat.(M = 0 \wedge N = n \supset M + N = n).$$

We then eliminate existential variables, simplifying the constraint to

$$\forall n : nat.0 + n = n$$

which is entered into a constraint store and later easily verified.

Note that we have been able to eliminate all the existential variables in the above constraint. This is true in all our examples, but, unfortunately, we have not yet found a clear theoretical explanation why this is so. In practice, it is crucial that we eliminate all existential variables in constraints before passing them to a constraint solver. Otherwise, we would have to deal with arbitrary formulas in Presburger arithmetic, which is decidable, but for which there are no practically efficient decision procedures available.

For the second clause in the definition of reverse

`rev(x::xs, ys) = rev(xs, x::ys)`,

we obtain the constraint

$$\forall m : nat.\forall n : nat.(m + 1) + n = m + (n + 1)$$

following the same procedure, where `xs` and `ys` are assumed to be of type `'a list(m)` and `'a list(n)`, respectively. Note that m and n are universally quantified, and the constraint can be solved easily.

```
fun('a){size:nat}
bsearch cmp (key, arr) = let
    fun look(lo, hi) =
        if hi >= lo then
            let
                val m = lo + (hi - lo) div 2
                val x = sub(arr, m)
            in
                case cmp(key, x) of
                    LESS => look(lo, m-1)
                  | EQUAL => (SOME(m, x))
                  | GREATER => look(m+1, hi)
            end
        else NONE
    where look <| {l:nat | 0 <= l <= size} {h:int | 0 <= h+1 <= size}
                    int(l) * int(h) -> 'a answer
in
    look (0, length arr - 1)
end
where bsearch <| ('a * 'a -> order) -> 'a * 'a array(size) -> 'a answer
```

Figure 3: The binary search function

$$\forall h : int.\forall l : nat.\forall size : nat.(0 \leq h+1 \leq size \wedge 0 \leq l \leq size \wedge h \geq l) \supset (l + (h-l)/2) \leq size$$
$$\forall h : int.\forall l : nat.\forall size : nat.(0 \leq h+1 \leq size \wedge 0 \leq l \leq size \wedge h \geq l) \supset 0 \leq l + (h-l)/2 - 1 + 1$$
$$\forall h : int.\forall l : nat.\forall size : nat.(0 \leq h+1 \leq size \wedge 0 \leq l \leq size \wedge h \geq l) \supset l + (h-l)/2 - 1 + 1 \leq size$$
$$\forall h : int.\forall l : nat.\forall size : nat.(0 \leq h+1 \leq size \wedge 0 \leq l \leq size \wedge h \geq l) \supset 0 \leq l + (h-l)/2 + 1$$
$$\forall h : int.\forall l : nat.\forall size : nat.(0 \leq h+1 \leq size \wedge 0 \leq l \leq size \wedge h \geq l) \supset l + (h-l)/2 + 1 \leq size$$

Figure 4: Sample constraints

In the standard basis we have refined the types of many common functions on integers such as addition, subtraction, multiplication, division, and the modulo operation. For instance,

```
+ <| {m:int} {n:int} int(m) * int(n) -> int(m+n)
```

is declared in the system. The code in Figure 3 is an implementation of binary search through an array. As before, we assume

```
sub <| {n:nat} {i:nat | i < n}
          'a array(n) * int(i) -> 'a
```

The explicit type parameter 'a is a recent feature of Standard ML to allow explicit scoping of type variables. We extend this notation to encompass type index variables, {size:nat} in this case.

We list some sample constraints generated from type-checking the above code in Figure 4. All of these can be solved easily.

Note that if we program binary search in C, the array bound check cannot be hoisted out of loops using the algorithm presented in (Gupta 1994) since it is neither increasing nor decreasing in terms of the definition given there. On the other hand, the method in (Susuki and Ishihata 1977) could eliminate this array bound check by synthesizing an induction hypothesis similar to our annotated type for `look`. Unfortunately, synthesizing induction hypotheses is often prohibitively expensive in practice. In future work we plan in investigate extensions of the type-checker which could infer certain classes of generalizations, thereby relieving the programmer from the need for certain kinds of "obvious" annotations.

3.2 Solving constraints

When all existential variables have been eliminated and the resulting constraints collected, we check them for linearity. We currently reject non-linear constraints rather than postponing them as *hard constraints* (Michaylov 1992), which is planned for future work. If the constraints are linear, we negate them and test for unsatisfiability. Our technique for solving linear constraints is mainly based on Fourier variable elimination, but there are many other methods available for this purpose such as the SUP-INF method (Shostak 1977) and the well-known simplex method. We have chosen Fourier's method mainly for its simplicity.

We now briefly explain this method. We use x for integer variables, a for integers, and l for linear expressions. Given a set of inequalities S, we would like to show that S is unsatisfiable. We fix a variable x and transform all the linear inequalities into one of the forms $l \leq ax$ or $ax \leq l$ for $a \geq 0$. For every pair $l_1 \leq a_1 x$ and $a_2 x \leq l_2$, where $a_1, a_2 > 0$, we introduce a new inequality $a_2 l_1 \leq a_1 l_2$ into S, and then remove all the inequalities involving x from S. Clearly, this is a sound but incomplete procedure. If x were a real variable, then the elimination would also be complete.

In order to handle modular arithmetic, we also perform another operation to rule out non-integer solutions: we transform an inequality of form

$$a_1 x_1 + \cdots + a_n x_n \leq a$$

into

$$a_1 x_1 + \cdots + a_n x_n \leq a',$$

where a' is the largest integer such that $a' \leq a$ and the

Program	constraints			type annotations		
	number	SML of NJ	MLWorks	total number	total lines	code size
bcopy	187	0.59/1.17	0.72/1.37	13	50	281 lines
binary search	13	0.07/0.02	0.10/0.04	2	2	33 lines
bubble sort	15	0.08/0.03	0.11/0.06	3	3	37 lines
matrix mult	18	0.10/0.04	0.16/0.06	5	10	50 lines
queen	18	0.11/0.03	0.14/0.04	9	9	81 lines
quick sort	135	0.29/0.58	0.37/0.68	16	40	200 lines
hanoi towers	29	0.10/0.09	0.13/0.13	4	10	45 lines
list access	4	0.07/0.01	0.08/0.01	2	3	18 lines

Table 1: Constraint generation/solution, time in secs

greatest common divisor of $a_1, \ldots, a_n$ divides a'. This is used in type-checking an optimized byte copy function.

The above elimination method can be extended to be both sound and complete while remaining practical (see, for example, (Pugh and Wonnacott 1992; Pugh and Wonnacott 1994)). We hope to use such more sophisticated methods which appear to be practical, although we have not yet found the need to do so in the context of our current experiments.

4 Experiments

We have performed some experiments on a small set of programs. Note that three of them (bcopy, binary search, and quicksort) were written by others and just annotated, providing evidence that a natural ML programming style is amenable to our type refinements.

The first set of experiments were done on a Dec Alpha 3000/600 using SML of New Jersey version 109.32. The second set of experiments were done on a Sun Sparc 20 using MLWorks version 1.0. Sources of the programs can be found in (Xi 1997).

Table 1 summarizes some characteristics of the programs. We show the number of constraints generated during type-checking and the time taken for generating and solving them using SML of New Jersey and MLWorks. Also we indicate the number of total type annotations in the code, the number lines they occupy, and the code size. Note that some of the type annotations are already present in non-dependent form in ML, depending on programming style and module interface to the code. A brief description of the programs is given below.

bcopy This is an optimized implementation of the byte copy function used in the Fox project. We used this function to copy 1M bytes of data 10 times in a byte-by-byte style.

binary search This is the usual binary search function on an integer array. We used this function to look for 2^{20} randomly generated numbers in a randomly generated array of size 2^{20}.

bubble sort This is the usual bubble sort function on an integer array. We used this function to sort a randomly generated array of size 2^{13}.

matrix mult This is a direct implementation of the matrix multiplication function on two-dimensional integer arrays. We applied this function to two randomly generated arrays of size 256×256.

queen This is a variant of the well-known eight queens problem which requires positioning eight queens on a 8×8 chessboard without one being captured by another. We used a chessboard of size 12×12 in our experiment.

quick sort This implementation of the quick sort algorithm on arrays is copied from the SML of New Jersey library. We sorted a randomly generated integer array of size 2^{20}.

hanoi towers This is a variant of the original problem which requires moving 64 disks from one pole to another without stacking a larger disk onto a smaller one given the availability of a third pole. We used 24 disks in our experiments.

list access We accessed the first sixteen elements in a randomly generated list at total of 2^{20} times.

We used the standard, safe versions of `sub` and `update` for array access when compiling the programs into the code with array bound checks. These versions always perform run-time array bound checks according to the semantics of Standard ML. We used unsafe versions of `sub` and `update` for array access when generating the code containing no array bound checks. These functions can be found in the structure `Unsafe.Array` (in SML of New Jersey), and in `MLWorks.Internal.Value` (in MLWorks). Our unsafe version of the `nth` function used `cast` for list access without tag checking.

Notice that unsafe versions of `sub`, `update` and `nth` can be used in our implementation only if they are assigned the corresponding types mentioned in Section 2.1.

In Table 2 and Table 3, we present the effects of eliminating array bound checks and list tag checks. Note that the difference between the number of eliminated array bound checks in Table 2 and Table 3 reflects the difference between randomly generated arrays used in two experiments.

It is clear that the gain is significant in all cases, rewarding the work of writing type annotations. In addition, type annotations can be very helpful for finding and fixing bugs, and for maintaining a software system since they provide the user with informative documentation. We feel that these factors yield a strong justification for our approach.

5 Related work

From the point of view of language design, our work falls in between full program verification, either in type theory (Constable et al. 1986; Dowek, Felty, Herbelin, Huet, Murthy,

Program	with checks	without checks	gain	checks eliminated
bcopy	6.52	4.40	32%	20,971,520
binary search	40.40	30.10	25%	19,072,212
bubble sort	58.90	34.25	42%	134,429,940
matrix mult	30.62	16.79	45%	33,619,968
queen	15.85	11.06	30%	77,392,496
quick sort	29.85	25.32	15%	64,167,588
hanoi towers	11.34	8.28	27%	50,331,669
list access	2.24	1.24	45%	1,048,576

Table 2: Dec Alpha 3000/600 using SML of NJ working version 109.32, time unit = sec.

Program	with checks	without checks	gain	checks eliminated
bcopy	9.75	2.01	79%	20,971,520
binary search	31.78	25.00	21%	19,074,429
bubble sort	46.78	25.84	45%	134,654,868
matrix mult	60.43	51.27	15%	33,619,968
queen	29.81	14.81	50%	77,392,496
quick sort	79.95	70.28	12%	63,035,841
hanoi towers	9.59	7.20	25%	50,331,669
list access	1.58	0.77	51%	1,048,576

Table 3: Sun Sparc 20 using MLWorks version 1.0, time unit = sec.

Parent, Paulin-Mohring, and Werner 1993) or systems such as PVS (Owre, Rajan, Rushby, Shankar, and Srivas 1996), and traditional type systems for programming languages. When compared to verification, our system is less expressive but more automatic, when compared to traditional programming languages our system is more expressive, but also more verbose. Since we extend ML conservatively, dependent types can be used sparingly, and existing ML programs will work as before if there is no keyword conflict.

Hayashi proposed a type system ATTT (Hayashi 1991), which allows a notion of refinement types as in (Freeman and Pfenning 1991), plus union and singleton types. He demonstrated the value of singleton, union and intersection types in extracting realistic programs, which is similar to our use of the corresponding logical operators on constraints. However, his language does not have effects and he does not address the practical problem of type checking or partial inference.

We have already compared some of the work on array bound checking for other languages (Markstein and Markstein 1982; Gupta 1994; Susuki and Ishihata 1977), most of which is based on automated analysis or inference, and thus more limited while requiring no annotations. In many cases a considerable number of array bound checks remain, which limits the efficiency gains. Furthermore, these methods provide no feedback to the programmer regarding the correctness of his code, which is an important component of our solution. We also deal with advanced features of ML such as higher-order functions and polymorphism. The work by Jay and Sekanina (Jay and Sekanina 1996) which includes these features and has similar goals and approach to ours is more restrictive in the design and seems more promising for languages based on iteration schemas rather than general recursion.

Also related is the work on a *certifying compiler* by Necula and Lee, which introduces precondition annotations for a type-safe subset of C in order to eliminate array bound checks (Necula and Lee 1998) and generate proof-carrying code (Necula 1997). Their language is significantly simpler (for example, it does not include higher-order functions or polymorphism), which allows them to formulate their extensions without constructing a full type system. They also do not include existential types, which we found necessary in a number of our examples.

6 Conclusion and future work

We have demonstrated the practicality of the use of dependent types in a statically typed functional language to eliminate dynamic array bound and tag checks. The required additional type annotations are concise, intuitive and aid the programmer in writing correct and in many cases significantly more efficient programs. The necessary constraint simplification, though theoretically intractable, has proved practically feasible, even with a simple-minded implementation and currently incomplete algorithm.

Our immediate goal is to extend our system to accommodate full Standard ML which involves treating exceptions and module-level constructs. We would also like to incorporate the ideas and observations from (Pugh and Wonnacott 1994) into our constraint solver and improve its efficiency.

We also plan to pursue using our language as a front-end for a certifying compiler for ML along the lines of work by Necula and Lee (Necula and Lee 1998) for a safe subset of C. We can propagate program properties (including array bound information) through a compiler where they can be used for optimizations or safety certificates in proof-carrying code (Necula 1997).

This work arose from a larger effort to incorporate a more general form of dependent types into ML (Xi 1998). Our extended type checking algorithm is robust (in the sense that it can collect constraints independently of their domain), because we separated the language of indices and programs. This allows other program invariants or properties to be

expressed, propagated, and checked, and we plan to investigate the use of other constraint domains and simplification procedures.

At present, unsolved constraints generated during type-checking may provide some hints on where type errors originate, but they are often inaccurate and obscure. Therefore, we plan to investigate how to generate more informative error messages should dependent type-checking fail. In contrast, many other approaches to eliminating array bound checking could give the user little or no feedback when an error is found.

7 Acknowledgements

We thank Peter Lee for providing us with many interesting examples and comments. We also gratefully acknowledge discussions with Rowan Davies and George Necula regarding the subject of the paper, and the help received from Kenneth Cline on measurements. Finally, we would like to thank the anonymous referees for their valuable comments.

References

Blelloch, G. E. (1993, April). NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University.

Chow, F. (1983). *A portable machine-independent global optimizer - design and measurements.* Ph. D. dissertation, Stanford University. Technical Report 83-254.

Constable, R. L. et al. (1986). *Implementing Mathematics with the Nuprl Proof Development System.* Englewood Cliffs, New Jersey: Prentice-Hall.

Corman, T. H., C. E. Leiserson, and R. L. Rivest (1989). *Introduction to Algorithms.* Cambridge, Massachusetts: The MIT Press.

Davies, R. (1997, November). Practical refinement-type checking. Thesis Proposal.

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

Freeman, T. (1994, March). *Refinement Types for ML.* Ph. D. dissertation, Carnegie Mellon University. Available as Technical Report CMU-CS-94-110.

Freeman, T. and F. Pfenning (1991). Refinement types for ML. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, pp. 268–277.

Gupta, R. (1994). Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems 2*(1–4), 135–150.

Hayashi, S. (1991). Singleton, union and intersection types for program extraction. In A. R. Meyer (Ed.), *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pp. 701–730.

Jay, C. and M. Sekanina (1996). Shape checking of array programs. Technical Report 96.09, University of Technology, Sydney, Australia.

Markstein, V., C. J. and P. Markstein (1982). Optimization of range checking. In *SIGPLAN '82 Symposium on Compiler Construction*, pp. 114–119.

Martin-Löf, P. (1980). Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pp. 153–175. North-Holland.

Michaylov, S. (1992, August). *Design and Implementation of Practical Constraint Logic Programming Systems.* Ph. D. thesis, Carnegie Mellon University. Available as Technical Report CMU-CS-92-168.

Necula, G. (1997). Proof-carrying code. In *Conference Record of 24th Annual ACM Symposium on Principles of Programming Languages*, pp. 106–119. ACM press.

Necula, G. and P. Lee (1998, June). The design and implementation of a certifying compiler. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation.* ACM press.

Owre, S., S. Rajan, J. Rushby, N. Shankar, and M. Srivas (1996, July/August). PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger (Eds.), *Computer-Aided Verification, CAV '96*, Volume 1102 of *LNCS*, New Brunswick, NJ, pp. 411–414. Springer-Verlag.

Pugh, W. and D. Wonnacott (1992). Eliminating false data dependences using the Omega test. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 140–151. ACM Press.

Pugh, W. and D. Wonnacott (1994, November). Experience with constraint-based array dependence analysis. Technical Report CS-TR-3371, University of Maryland.

Shostak, R. E. (1977, October). On the SUP-INF method for proving Presburger formulas. *Journal of the ACM 24*(4), 529–543.

Sun Microsystems (1995). The Java language specification. Available as `ftp://ftp.javasoft.com/docs/javaspec.ps.zip`.

Susuki, N. and K. Ishihata (1977). Implementation of array bound checker. In *4th ACM Symposium on Priciples of Programming Languages*, pp. 132–143.

Xi, H. (1997, November). Some examples of DML programming. Available at `http://www.cs.cmu.edu/~hwxi/DML/examples/`.

Xi, H. (1998). *Dependent Types in Practical Programming.* Ph. D. thesis, Carnegie Mellon University. Forthcoming.

A Knuth-Morris-Pratt string matching

The following is an implementation of the Knuth-Morris-Pratt string matching algorithm using dependent types to eliminate most array bound checks.

```
assert length <| {n:nat} 'a array(n) -> int(n)

and sub <| {size:int, i:int | 0 <= i < size} 'a array(size) * int(i) -> 'a
    (* sub requires NO bound checking *)

and subCK <| 'a array * int -> 'a (* subCK requires bound checking *)

type intPrefix = [i:int| 0 <= i+1] int(i) (* notice the use of existential types *)

assert arrayPrefix <| {size:nat} int(size) * intPrefix -> intPrefix array(size)

and subPrefix <| {size:int, i:int | 0 <= i < size} intPrefix array(size) * int(i) -> intPrefix
    (* subPrefix requires NO bound checking *)

and subPrefixCK <| intPrefix array * int -> intPrefix (* subPrefixCK requires bound checking *)

and updatePrefix <| {size:int, i:int | 0 <= i < size}
                    intPrefix array(size) * int(i) * intPrefix -> unit
    (* updatePrefix requires NO bound checking *)

(* computePrefixFunction generates the prefix function table for the pattern pat *)
fun computePrefixFunction(pat) = let
        val plen = length(pat)
        val prefixArray = arrayPrefix(plen, ~1)

        fun loop(i, j) = (* calculate the prefix array *)
            if (j >= plen) then ()
            else
                if sub(pat, j) <> subCK(pat, i+1) then
                    if (i >= 0) then loop(subPrefixCK(prefixArray, i), j)
                    else loop(~1, j+1)
                else (updatePrefix(prefixArray, j, i+1); loop(subPrefix(prefixArray, j), j+1))
        where loop <| {j:nat} intPrefix * int(j) -> unit
    in
        (loop(~1, 1); prefixArray)
    end
where computePrefixFunction <| {p:nat} int array(p) -> intPrefix array(p)

fun kmpMatch(str, pat) = let
        val strLen = length(str)
        and patLen = length(pat)

        val prefixArray = computePrefixFunction(pat)

        fun loop(s, p) =
            if s < strLen then
                if p < patLen then
                    if sub(str, s) = sub(pat, p) then loop(s+1, p+1)
                    else
                        if (p = 0) then loop(s+1, p)
                        else loop(s, subPrefix(prefixArray, p-1)+1)
                else (s - patLen)
            else ~1
        where loop <| {s:nat, p:nat} int(s) * int(p) -> int
    in
        loop(0, 0)
    end
where kmpMatch <| {s:nat, p:nat} int array(s) * int array(p) -> int
```

Figure 5: An Implementation of Knuth-Morris-Pratt String Matching Alogirithm