Eliminating Array Bounds Checking Through Dependent Types*
Presented by Cherif Salama

Gregory M. Malecha

Febuary 01, 2008

Array indexing is a common problem in almost all languages which current mainstream type systems are unable to accurately capture. In languages such as Java, all array accesses are bounds checked, which keeps the language safe but at the cost of performance which can become significant. The goal is to be able to statically eliminate bounds checking to guarantee additional safety properties as well as gain run-time performance.

According to the paper, there are two basic approaches to eliminating array bounds checking. The distinction between the two approaches is the amount of programmer annotations needed. *Automated analysis* attempts to eliminate checks without programmer supplied annotations. This requires that it derive loop-invariants and function contracts which is both theoretically undecidable and practically difficult. The alternative approach, proposed in the paper is to use *dependent types* as annotations. **Aside:** It was brought up as to which of these is more powerful. Which seems unclear from the paper, however the difficulty of the automated approach seems to suggest that it is not feasible or dependable.

Dependent types in this paper are based on the same concepts as in Hongwei's other papers. Namely, indexes which are of one of two sorts, Boolean and integer. While the paper lists many operations on indexes, it is important to note that the constraint solver is unable to solve non-linear constraints and so they are dropped. Types are annotated by indexes and can quantify both existentially (denoted Σ) and universally (denoted Π).

$$
\begin{array}{rrcl}
\text{integer indexes} & i, j & ::= & a|i+j|i-j|i*j|div(i,j)|min(i,j)|max(i,j)|abs(i)|sgn(i)|mod(i,j) \\
\text{Boolean indexes} & b & ::= & a|false|true|i<j|i\leq j|i=j|i\geq j|i>j|\neg b|b_1\wedge b_2|b_1\vee b_2 \\
\text{index sorts} & \gamma & ::= & int|bool|\{a:\gamma|b\} \\
\text{types} & \tau & ::= & \alpha|(\tau_1,...,\tau_n)\delta(d_1,..,d_k)|\tau_1*...*\tau_n|\tau_1\rightarrow\tau_2|\Pi a:\gamma.\tau|\Sigma a:\gamma.\tau
\end{array}
$$

Figure 1 illustrates a way that dependent types can be used to eliminate array bounds checking. The first two lines declare types for functions which are not shown. `length` takes input of an array of size n and returns n and `sub` is the subscript operation. Note that subscript requires that

*Eliminating Array Bound Checking Through Dependent Types. Hongwei Xi and Frank Pfenning (PLDI'98)

the index being accessed is less than the length of the array (encoded as the condition $i < n$). The `dotprod` function takes in two arrays in which the second is at least as large as the first and computes the dot-product by recurring with `loop`. Since the programmer specifies range invariants for the values in `loop`, the compiler does not need to derive these which makes the problem more tractable.

```
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: Dependently typed array dot-product

Constraint solving is done after standard type inference. Constraints are extracted from the program and solved with Fourier variable elimination; however, other methods are currently believed work better now, namely the Simplex method. Consider generating constraints for accumulator-style reverse:

```
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)
```

The first case yields the constraint:

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

Where N is the instantiation of the index variable n and M is the instantiation of m. The $\subset$ denotes implication. This constraint can be simplified to $\forall n : nat.0 + n = n$ which is trivially solvable. The second line gives a similar constraint which simplifies to:

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

Which can be solved with associativity of addition.

Cherif is planning on adding wire size checking to VPP to be able to make additional guarantees about the correctness when working with circuit generators. Before implementation, he is planning on looking into other methods for doing this.