A Modular Type Inference and Specializer for Array Bound Checks Elimination

Dana N. Xu[1], Corneliu Popeea[2], Siau-Cheng Khoo[2], and Wei-Ngan Chin[2]
[1] Computer Laboratory, University of Cambridge
[2] Department of Computer Science, National University of Singapore
Na.Xu@cl.cam.ac.uk, {popeeaco|khoosc|chinwn}@comp.nus.edu.sg

ABSTRACT

We propose a modular inference mechanism, based on a dependent type system with Presburger arithmetic, to find array checks that can be safely eliminated. Our proposal works for a core imperative language with assignments, and can discover symbolic program states (or postconditions) to support the elimination of redundant checks. Our inference is *precise* as it is both path and context sensitive. It is also *efficient*, as we analyse each method *once* through a summary-based approach. Moreover, through suitable techniques that either weaken postconditions or strengthen preconditions, we can selectively reduce the sizes of our formulae. We also subject each inferred program to a flexivariant specialization process that can achieve a tradeoff between maximal elimination of array checks whilst being mindful on code explosion concerns. We have proven the soundness of our approach and have also implemented a prototype inference and specialization system. Initial experiments suggest that such a system can be both efficient and accurate.

1. INTRODUCTION

Array bound check optimization has been extensively investigated over the last three decades [25, 7, 26, 12], with renewed interests as recently as [2, 30, 9, 27]. Most array optimization techniques (*e.g.* [7, 28, 25]) focus on the elimination of totally redundant checks. To achieve this, whole program analysis is carried out to propagate analysis information (*e.g.* availability) to each program point. Even for techniques that handle partially redundant checks, such as partial redundancy elimination (PRE)[3], the focus has been on either moving these checks or restructuring the control flows, but without exploiting path-sensitivity or interprocedural relational analysis. These features are important for supporting precision and modularity.

In this paper, we propose a high-level approach towards array checks optimization that is both precise and efficient. Our approach is based on the derivation of a suitable precondition for each array check across the method boundary, followed by program specialization to eliminate array checks found to be redundant. We formalise our technique as a type inference system that is able to process each method independently, and yet exploits the different contexts of its multiple callers. Successful elimination of array checks depends on how accurately we are able to infer the states of the program variables. To achieve this, we employ a restricted form of dependent type [16, 4] with Presburger arithmetic, that can capture symbolic program states using a relational analysis. The key contributions of this paper include:

- **Imperative Inference :** We formulate a dependent type system for a core imperative language. A significant innovation over past work [28, 16, 5] is that our approach is able to *automatically infer* suitable preconditions and postconditions for each method, in the presence of updates/assignments.
- **Modularity:** We adopt a *summary-based approach* that gathers preconditions, postcondition and unsafe checks for each method. While summary-based techniques have already been proposed for a number of program analyses[22, 23, 31], we have generalised this approach to handle the inference of both preconditions and postconditions (Presburger arithmetic form) in the presence of recursion.
- **Precision :** Being based on a modular type system, our approach has the potential for increased precision. Through *disjunctive formulae* of dependent types, we can express both relational and path-sensitive analysis to give more precise program states.
- **Performance :** To obtain an efficient analysis, we devise new techniques to make *formulae smaller* (in size) by suitable strengthening of preconditions and weakening of postconditions. This approach trades some precision for speed and has been validated by experiments with our prototype inference system. Furthermore, we employ a *flexivariant* specialization scheme that is able to trade code space for optimization.

2. OVERVIEW

A key feature of our approach is the three-way classification of checks before optimization. Given a method definition with a set of parameters V and a set of checks C, our approach will classify each check ($c \in C$) that occurs at a location with a symbolic program state s, as follows:

- c is *safe* if it is redundant under its location's program state. This holds if the following is *valid*:

$$(s \Rightarrow c)$$

- c is *partially-safe* if it may become redundant under an extra condition. This holds if there exists a *satisfiable* precondition *pre* (expressed in terms of variables from only V) such that the following is *valid*.

$$(pre \wedge s \Rightarrow c) \quad (1)$$

  Such a precondition can be generated using $pre = (\forall L \cdot \neg s \vee c)$ where L is the set of local variables, denoted by $vars(s, c) - V$.

- c is *unsafe*, if no precondition can be found in (1). This implies that the check c *may* fail at runtime.

Partially-safe checks are special in that they allow us to propagate checks across methods from callees to callers. This mechanism can help us exploit program states at callers' sites for the elimination of checks. Let us highlight this classification, together with key techniques we shall employ.

Consider the `foo` example from top of Figure 1 where `randInt` returns a random integer, while `abs` converts each number into its positive counterpart. The set of parameters V at method boundary is {`a`, `j`, `n`} where `a` is an array with indices from `0` to `len(a)` $-$ `1`. This method contains two array accesses at locations ℓ_1 and ℓ_2. The symbolic program states at these sites may be affected by the type invariants[1], conditionals, imperative updates and by prior calls. In particular, the symbolic states for the three locations in this example, including one at method entry ℓ_0, are:

$$\begin{aligned} \texttt{sps}(\ell_0) &= \texttt{len(a)>0} \\ \texttt{sps}(\ell_1) &= \texttt{sps}(\ell_0) \wedge \texttt{i=j+1} \wedge \texttt{(0<i<=n)} \\ \texttt{sps}(\ell_2) &= \texttt{sps}(\ell_0) \wedge \texttt{i=j+1} \wedge \texttt{m>=0} \end{aligned}$$

Based on the earlier classification of checks, we can establish that the low-bound checks (at ℓ_1 and ℓ_2) are safe, since:

$$\texttt{sps}(\ell_1) \Rightarrow \texttt{(i>=0)} \quad \text{and} \quad \texttt{sps}(\ell_2) \Rightarrow \texttt{(m>=0)}$$

For the high-bound checks (denoted by ℓ_1`.H` and ℓ_2`.H`), we derive (the weakest) preconditions through universal quantification of the local variables, as follows:

$$\begin{aligned} \texttt{pre}(\ell_1\texttt{.H}) &= \forall \texttt{i} \cdot \texttt{(sps(}\ell_1\texttt{)} \Rightarrow \texttt{i<len(a))} \\ &= \forall \texttt{i} \cdot \texttt{(len(a)>0} \wedge \texttt{i=j+1} \wedge \texttt{(0<i<=n)} \Rightarrow \texttt{i<len(a))} \\ &= \texttt{len(a)<=0} \vee \texttt{(j<=len(a)-2} \wedge \texttt{1<=len(a))} \\ &\quad \vee \texttt{(1<=len(a)<=j+1} \wedge \texttt{n<=j)} \\ \texttt{pre}(\ell_2\texttt{.H}) &= \forall \texttt{i,m} \cdot \texttt{(sps(}\ell_2\texttt{)} \Rightarrow \texttt{m<len(a))} \\ &= \forall \texttt{i,m} \cdot \texttt{(len(a)>0} \wedge \texttt{i=j+1} \wedge \texttt{m>=0} \Rightarrow \texttt{m<len(a))} \\ &= \texttt{len(a)<=0} \end{aligned}$$

While the derived preconditions may be the weakest, they do not take into account the *type invariant*, causing their formulae to be possibly larger than needed. For example, `len(a)>0` is a type invariant and can be used to simplify the precondition of `pre(`ℓ_2`.H)` to `false`. Similarly, `pre(`ℓ_1`.H)` can be simplified to `j<=len(a)-2` $\vee$ `n<=j` using the same type invariant. This simplified formula contains a condition `j<=len(a)-2` for satisfying the check, and another condition `n<=j` for avoiding the check.

[1]An example of a type invariant is that the size of an array `a`, denoted by `len(a)`, is positive (a design decision we took for our language).

We perform each simplification of a formula ϕ_1 under type invariant ϕ_2 by the operation (*gist* ϕ_1 *given* ϕ_2). This yields a simplified term ϕ_3 such that $\phi_3 \wedge \phi_2 \equiv \phi_1 \wedge \phi_2$ [21]. As it turns out, the same simplification technique can be used to trade-off precision for performance in our analysis, where we are able to derive smaller (but stronger) preconditions. This approach to better performance is further described in Section 8, and is crucial for overcoming the intractability of solving large Presburger arithmetic formulae.

One feature of our modular optimization is its formulation in two stages: *type inference followed by specialization*. The type inference stage processes methods in reverse topological order of the call graph. It computes post-states at each program point, classifies checks and propagates preconditions as new checks at each *method boundary*. It also marks all unsafe checks. These information are collected for each method declaration with a postcondition Δ, a set of preconditions Φ, a set of unsafe checks Υ, and size-annotated types $\tau_0, .., \tau_k$.

$$\tau_0 \ m \ (\tau_1 \ v_1, \ldots, \tau_k \ v_k) \ \texttt{where} \ \Delta; \ \Phi; \ \Upsilon \ \{body\}$$

For example, after type inference on the `foo` method, we would obtain the method displayed in the middle of the Figure 1, where the unchanged method body is replaced by $\{\ldots\}$.

The results of inference are then used by the specialization stage to insert runtime tests to guard unsafe checks and to derive *target programs that are well-typed*. Well-typed specialised methods are decorated with a postcondition Δ and a precondition ϕ_{pre} as part of its method header. The precondition ϕ_{pre} is a conjunction of checks from Φ that are guaranteed safe from each call site, as outlined below.

$$\tau_0 \ m \ (\tau_1 \ v_1, \ldots, \tau_k \ v_k) \ \texttt{where} \ \Delta; \ \phi_{pre} \ \{body\}$$

For example, if `pre(`ℓ_1`.H)` is found to be safe when analyzing the call sites of method `foo`, we can generate the specialised (and well-typed) method at the bottom of Figure 1.

```
Float foo(Float[] a, Int j, Int n)
 ℓ0:{ Float v=0.0; Int i=j+1;
      if (0<i<=n) then v=(ℓ1:a[i]) else ();
      Int m=abs(randInt());
      v+(ℓ2:a[m]) }

              ⇓ Inference

Float foo(Float[Int^s] a, Int^j j, Int^n n)
  where (n≤j) ∨ (j≤s−2∧j≤n−1) ;
     {ℓ1.H : j≤s−2)}; {ℓ2.H} {···}

              ⇓ Specialization

Float foo(Float[Int^s] a, Int^j j, Int^n n)
  where (n≤j) ∨ (j≤s−2∧j≤n−1); (j≤s−2)
 ℓ0:{ Float v=0.0; Int i=j+1;
      if (0<i<=n) then v=(ℓ1:a[i]) else ();
      Int m=abs(randInt());
      v+(if (m<len(a)) then ℓ2:a[m] else error) }
```

Figure 1: Infer and Specialize : An Example

Well-typed programs are safe in that no array bound errors are ever encountered by each array access during program execution. This safety property is guaranteed by either the program context (for array checks ℓ_1`.L` and ℓ_2`.L`),

$$
\begin{array}{rcl}
\texttt{P} & ::= & md^* \\
md & ::= & \texttt{t}\ m\ (\texttt{t}_1\ v_1, \ldots, \texttt{t}_k\ v_k)\ \{e\} \\
e & ::= & c \mid v \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid v = e \\
 & & \mid \texttt{t}\ v{=}e_1\ ;\ e_2 \mid m(v_1, \ldots, v_k) \\
\texttt{t} & ::= & \underline{\texttt{t}} \mid \underline{\texttt{t}}[\texttt{Int}, \ldots, \texttt{Int}] \\
\underline{\texttt{t}} & ::= & \texttt{Void} \mid \texttt{Int} \mid \texttt{Bool} \mid \texttt{Float}
\end{array}
$$

Figure 2: Source Imp language

or the precondition of each method (for array check $\ell_1.\texttt{H}$) or by inserted runtime test (for array check $\ell_2.\texttt{H}$). In the rest of this paper, we shall formalise a type inference system to derive well-typed programs (with safe array accesses) for a core imperative language.

3. AN IMPERATIVE LANGUAGE

A traditional mechanism for reasoning about imperative programs is Hoare logic[15], whereby precondition must be verified and postcondition may be asserted for each program fragment, denoted by $\{pre\}$ `code` $\{post\}$. In this paper, we propose the use of an advanced type system with similar reasoning capability as Hoare logic, except that it shall be fully automatic.

One key feature of our type system is that it captures program variables and data structures through the use of types annotated with size variables. For example, a boolean value can be denoted by $\texttt{Bool}^b$ where $b = 0$ represents `false` and $b = 1$ represents `true`; an integer value can be denoted by $\texttt{Int}^n$ with n to denote its integer value, while $\texttt{Float}[\texttt{Int}^s]$ can denote an array of floats with s elements. Input-output relation between size variables from method parameters and result is capture after the `where` keyword:

$$
\begin{array}{l}
\texttt{Int}^r\ \texttt{randInt()}\ \ \texttt{where true};\ldots \\
\texttt{Int}^r\ \texttt{abs(Int}^a\ \texttt{v)} \\
\quad \texttt{where}\ (a{<}0 \wedge r{=}{-}a \vee a{\geq}0 \wedge r{=}a) \wedge (a'{=}a);\ldots \\
\texttt{Int}^r\ \texttt{add(Int}^a\ \texttt{x, Int}^b\ \texttt{y)} \\
\quad \texttt{where}\ (r{=}a{+}b) \wedge no\mathcal{X}\{a,b\};\ldots \\
\texttt{Bool}^r\ \texttt{lessThan(Int}^a\ \texttt{x, Int}^b\ \texttt{y)} \\
\quad \texttt{where}\ (a{<}b \wedge r{=}1 \vee a{\geq}b \wedge r{=}0) \wedge no\mathcal{X}\{a,b\};\ldots
\end{array}
$$

The prime notation is used to denote the state of size variables at the end of the method. Parameter values that are unchanged across method calls are captured using $no\mathcal{X}\{a,b\} {\equiv} (a'{=}a \wedge b'{=}b)$ as a shorthand for "no change in state". Note that `true` for `randInt` signifies that r is unbounded. Also, non-trivial size relations can be supported through disjunctive formulae.

3.1 Core Language

To formalise our type inference we first introduce a source language Imp (see Figure 2), where types, denoted by `t` and $\underline{\texttt{t}}$, do not have annotations. Imp has support for assignments, conditionals, local declarations, method calls, and multidimensional arrays. Typical language constructs, such as multi-declaration block, sequence, calls with complex arguments can be automatically translated to constructs in Imp. In addition, loops can be viewed as syntactic abbreviations for tail-recursive methods, and are supported by our analysis. Array operations are implemented as calls to library (or primitive) functions (e.g. `newArr`, `len`, `sub`, `assign`).

In Figure 3 we introduce a language with dependent type, Imp_I, which is designed to capture the result of our inference. Each method declaration captures three information:

$$
\begin{array}{rcl}
md & ::= & \tau\ m\ (\tau_1\ v_1, \ldots, \tau_k\ v_k)\ \texttt{where}\ \Delta; \Phi; \Upsilon\ \{e\} \\
p & ::= & \tau\ m\ (\tau_1\ v_1, \ldots, \tau_k\ v_k)\ \texttt{where}\ \Delta; \Phi; \texttt{C} \\
e & ::= & \cdots \mid \ell : m\ (v_1, \ldots, v_k) \\
\tau, \hat{\tau} & ::= & \underline{\tau} \mid \underline{\tau}[\texttt{Int}^{s_1}, \ldots, \texttt{Int}^{s_k}] \\
\underline{\tau} & \in & \textbf{PrimAnnType} \\
 & ::= & \texttt{Void} \mid \texttt{Int}^n \mid \texttt{Bool}^n \mid \texttt{Float} \\
\Phi & ::= & \{\ (l^+ : \phi)^*\ \}\ \ \text{(Labelled Preconditions)} \\
\Upsilon & ::= & \{\ (l^+)^*\ \}\ \ \text{(Unsafe Checks)} \\
\texttt{C} & ::= & \{\ (l^+ : e)^*\ \}\ \ \text{(Labelled Runtime Checks)} \\
\ell & \in & \textbf{Label} \\
\ell^+ & ::= & \ell \mid \ell_1.\cdots.\ell_n \\
\phi, \Delta & ::= & b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid q\langle n^* \rangle \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1{=}a_2 \mid a_1{\neq}a_2 \mid a_1{<}a_2 \mid a_1{\leq}a_2 \\
a & ::= & c \mid n \mid c{*}a \mid a_1{+}a_2 \mid -a \\
 & & \text{where}\ \ n\ \text{denotes a size variable} \\
\mathcal{Q} & ::= & \{(q\langle n^* \rangle = \phi)^*\}
\end{array}
$$

Figure 3: Inferred Imp_I Language

an input-output relation (postcondition) Δ, a set Φ that contains a precondition for each partially-safe check, and a set of labels Υ, representing locations of unsafe checks. The labels from Φ and Υ identify call sites from the body of the current method. This is enabled in our language since every method call is *uniquely labelled*. The suffix notation s^* denotes a list of zero or more distinct syntactic terms separated by appropriate separators, while s^+ represents a list of one or more distinct syntactic terms.

For a non-recursive method `mn`, this information (Δ, Φ, Υ) can be derived via inference of the method body (since methods below `mn` in the calling hierarchy are already inferred.) To support recursive methods, we make use of *constraint abstractions* (adopted from [13]). For each mutual-recursive method, we first derive a (recursive) constraint abstraction Q of the form $q\langle n^* \rangle = \phi$. These abstractions are used by fix-point computation to provide a sound and precise analysis for recursive methods. An adaptation of the fix-point approximation from [7] is detailed via examples in Section 5. Besides constraint abstractions, our language of constraints contains conjunctions and disjunctions of linear (in)equalities. We make use of a Presburger calculator [21] (with support for universal and existential quantifications) to eliminate local variables or simplify formulae.

To support type inference and program specialization, we also require each primitive method (denoted by p in Fig 3) to be suitably annotated with a postcondition, a set of preconditions and a set of runtime tests. At a primitive call site, if some precondition is not satisfied, its corresponding runtime test is to be inserted by specialization. We provide a set of primitives for 1-dimensional array with element type τ, as shown below.

$$
\begin{array}{l}
\underline{\tau}[\texttt{Int}^n]\ \ \texttt{newArr(Int}^s\ \texttt{v}, \underline{\tau}\ \texttt{val)} \\
\quad \texttt{where}\ (0{<}s \wedge n{=}s \wedge s{=}s'); \{\texttt{S}\text{: } s{>}\ 0\}; \{\texttt{S: v>0}\} \\
\texttt{Int}^r\ \ \texttt{len}(\underline{\tau}[\texttt{Int}^n]\ a)\ \ \texttt{where}\ (r{=}n \wedge n{=}n'); \{\}; \{\} \\
\underline{\tau}\ \ \texttt{sub}(\underline{\tau}[\texttt{Int}^n]\ \texttt{a}, \texttt{Int}^s\ \texttt{v}) \\
\quad \texttt{where}\ \ (0{\leq}s{<}n \wedge no\mathcal{X}\{n,s\});\ \{\texttt{L}\text{: } 0{\leq}s, \texttt{H}\text{: } s{<}n\}; \\
\qquad \{\texttt{L: } 0{\leq}\texttt{v},\ \texttt{H: v<len(a)}\} \\
\texttt{Void}\ \ \texttt{assign}(\underline{\tau}[\texttt{Int}^n]\ \texttt{a}, \texttt{Int}^s\ \texttt{v}, \underline{\tau}\ \texttt{val}) \\
\quad \texttt{where}\ \ (0{\leq}s{<}n \wedge no\mathcal{X}\{n,s\});\ \{\texttt{L}\text{: } 0{\leq}s,\ \texttt{H}\text{: } s{<}n\}; \\
\qquad \{\texttt{L: } 0{\leq}\texttt{v}, \texttt{H: v<len(a)}\}
\end{array}
$$

Primitive `newArr` returns a new array with all elements initialized to value `v`, `len` returns the length at dimension `1`, `sub` returns an array element from the specified location, while `assign` updates the specified location with value `v`. The extension of these primitives to k-dimensional case is straightforward as shown in the technical report [29].

4. INFERENCE RULES FOR IMP_I

Our inference system analyses and propagates state information so as to determine if an array check is *safe* and if a *precondition* is to be propagated to the *method boundary*. The type judgment for the entire program is $P_m \vdash_I P \rightsquigarrow P_I$. It derives a program $P_I \in \text{IMP}_I$ from a program $P \in \text{IMP}$ and a set of primitive declarations P_m. The type judgement for expressions is specified as follows:

$$V;\Gamma;\Delta \vdash e \rightsquigarrow e_1 :: \tau, \Delta_1, \Phi, \Upsilon$$

Here V is a set of size variables available at the boundary of the method (called *boundary variables*) in which the expression e resides. Γ is a type environment mapping program variables to their annotated types; Δ (resp. Δ_1) is a size constraint over size variables occurring in Γ (resp. Γ and τ) before (resp. after) the evaluation of the expression e; τ is an annotated type.

The above judgement states that e will be transformed into e_1 with an annotated type τ during the inference. Both e and e_1 have the same underlying type. Furthermore, successful evaluation of e (and e_1) requires the validity of preconditions Φ, and the inclusion of the runtime tests Υ. Successful evaluation of e also changes the program state from Δ to Δ_1. Target expression e_1 contains both size-variable annotated types and *labels* that uniquely identify calls.

For convenience, our inference rules ensure that the size variables occurring in the annotated type τ are unique; *ie.,* $FSV(\tau) \cap FSV(\Gamma) = \emptyset$ where FSV returns a set of free size variables found. Some of the interesting inference rules are specified in Figure 4 (the entire set of rules is available in [29]). In the rules, we use $s = \mathit{fresh}()$ and $\ell = \mathit{fresh}()$ to generate a new size variable and a new label, respectively. We also extend it to annotated type, so that $\hat{\tau} = \mathit{fresh}(\texttt{t})$ (or $\hat{\tau} = \mathit{fresh}(\tau)$) returns a new type $\hat{\tau}$ with the same underlying type as `t` (or τ), but annotated with fresh size variables. The function $\mathit{equate}(\tau_1, \tau_2)$ generates equality constraints for the corresponding size variables of its two arguments, assuming both arguments share the same underlying type. For example, we have $\mathit{equate}(\texttt{Int}^n, \texttt{Int}^{m'}) = (n = m')$. The function $\mathit{rename}(\tau_1, \tau_2)$ returns a mapping instead, e.g. $\mathit{rename}(\texttt{Int}^n, \texttt{Int}^{m'}) = (n \mapsto m')$. Also, the conditional is expressed as $\zeta_1 \lhd b \rhd \zeta_2 =_{df}$ if b then ζ_1 else ζ_2. For the rest of this section, we highlight the important aspects of our inference system via examples.

4.1 Inferring Imperative Update

Consider an assignment $\texttt{v} = \texttt{v} + \texttt{u}$, with a pre-state $\Delta = (m'{=}2{+}n' \wedge n'{=}5)$ and $\Gamma = \{\texttt{u} :: \texttt{Int}^m, \texttt{v} :: \texttt{Int}^n, \ldots\}$. This example shows how the *prime* notation is used to capture the latest values of size variables at each symbolic state [14]. It also shows how updates are effected by a sequential composition operator, $\circ_X$, where X denotes a set of size variables that are being updated.

The following depicts the inference step for assignment:

$$\frac{\begin{array}{c}\Gamma(\texttt{v}) = \texttt{Int}^n \quad \Gamma(\texttt{u}) = \texttt{Int}^m \\ V;\Gamma;\Delta \vdash \texttt{v}+\texttt{u} \rightsquigarrow \texttt{v}+\texttt{u} :: \texttt{Int}^r, \Delta \wedge r = n' + m', \emptyset, \emptyset \\ \Delta_2 = \mathit{assign}(\Delta \wedge r = n' + m', \texttt{Int}^n, \texttt{Int}^r)\end{array}}{V;\Gamma;\Delta \vdash \texttt{v} = \texttt{v}+\texttt{u} \rightsquigarrow \texttt{v} = \texttt{v}+\texttt{u} :: \texttt{void}, \Delta_2, \emptyset, \emptyset}$$

The function assign performs the necessary sequential composition:

$$\begin{array}{ll} \mathit{assign}(\Delta, \tau, \tau_1) =_{def} & \mathit{let}\ X = FSV(\tau)\ ;\ Y = FSV(\tau_1) \\ & \mathit{in}\ \exists Y.(\Delta \circ_X \mathit{equate}(\mathit{prime}(\tau), \tau_1)) \end{array}$$

For our example, the correct post-state of the assignment can be computed as follows:

$$\begin{array}{rl} \Delta_2 = & \exists r \cdot ((\Delta \wedge r{=}n'{+}m') \circ_{\{n\}} (n'{=}r)) \\ = & \exists r \cdot ((m'{=}2{+}n' \wedge n'{=}5 \wedge r{=}n'{+}m') \circ_{\{n\}} (n'{=}r)) \\ = & \exists r \cdot (\exists n_0 \cdot m'{=}2{+}n_0 \wedge n_0{=}5 \wedge r{=}n_0{+}m' \wedge n'{=}r) \\ = & (m'{=}7 \wedge n'{=}m'{+}5) \end{array}$$

More formally, sequential composition is defined as:

$$\begin{array}{l} \phi_1 \circ_X \phi_2 =_{def} \exists R \cdot \rho_1(\phi_1) \wedge \rho_2(\phi_2) \\ \text{where} \quad X = \{s_1, \ldots, s_n\} \text{ are size variables being updated} \\ \qquad\quad R = \{r_1, \ldots, r_n\} \text{ are fresh size variables} \\ \qquad\quad \rho_1 = \{s'_i \mapsto r_i\}_{i=1}^n \qquad \rho_2 = \{s_i \mapsto r_i\}_{i=1}^n \end{array}$$

4.2 Path Sensitive Inference

The [If] rule attempts to track the size constraint of conditionals with path sensitivity. The two conditional branches are distinguished by assuming the conditional-test result to be either 1 or 0, representing the `true` or the `false` value, respectively. Given $\texttt{e} = \texttt{if u then v else 5}$ and $\Gamma = \{\texttt{v} :: \texttt{Int}^n, \texttt{u} :: \texttt{Bool}^b\}$, we have the following derivation, where Δ_3 is obtained by combining, via disjunction, the inference results of both branches. We replace both r_1 and r_2 (the resulting sizes from both branches) by the final resulting size r.

$$\frac{\begin{array}{c}\Delta_1 = \Delta \wedge (b' = 1) \quad \Delta_2 = \Delta \wedge (b' = 0) \\ V;\Gamma;\Delta_1 \vdash \texttt{v} \rightsquigarrow \texttt{v} :: \texttt{Int}^{r_1}, \Delta_1 \wedge (r_1 = n'), \emptyset, \emptyset \\ V;\Gamma;\Delta_2 \vdash \texttt{5} \rightsquigarrow \texttt{5} :: \texttt{Int}^{r_2}, \Delta_2 \wedge (r_2 = 5), \emptyset, \emptyset \\ \Delta_3 = \Delta \wedge ((b' = 1 \wedge r = n') \vee (b' = 0 \wedge r = 5))\end{array}}{V;\Gamma;\Delta \vdash \texttt{e} \rightsquigarrow \texttt{e} :: \texttt{Int}^r, \Delta_3, \emptyset, \emptyset}$$

4.3 Precondition for Safety of Check

Precondition derivation is essential for the detection of safe array-bound checks across method boundaries. The idea is to replace the preconditions associated with a call by preconditions associated with its caller. It also determines whether a call context can imply the call's preconditions. It is described in the [Call] rule. The generated preconditions are expressed in terms of the boundary variables.

As an example, consider inferring a primitive call $\texttt{sub}(\texttt{z}, \texttt{i})$ under the type assumption $\Gamma = \{\texttt{v} :: \texttt{Int}^v, \texttt{z} :: \texttt{Float}[\texttt{Int}^m], \texttt{i} :: \texttt{Int}^i\}$ and pre-state $\Delta = (m'{=}m) \wedge (m'{=}10) \wedge (i'{=}v'{+}2) \wedge (v'{=}v{+}1) \wedge (v'{>}5)$. Furthermore, let the set of boundary variables V be $\{v, m\}$ (thus `i` is a local). The two array-bound checks of `sub` (declared in Section 3.1), $0{\leq}s$ and $s{<}n$, are transformed into the following preconditions:

$$\begin{array}{rcl} \mathit{pre}_1 & = & (\Delta \not\approx \rho(0 \leq s))\downarrow_V \equiv_s \texttt{true} \\ \mathit{pre}_2 & = & (\Delta \not\approx \rho(s < n))\downarrow_V \equiv_s (v < 7) \end{array}$$

where $\rho = \{n \mapsto m, n' \mapsto m', s \mapsto i, s' \mapsto i'\}$. The substitution ρ replaces caller's size variables for those associated with the formal parameters of `sub`. The new preconditions are obtained by simplifying ($\equiv_s$) the result of the operations ($\not\approx$) and $\downarrow_V$. $\not\approx$ formulates the implication of an array-bound check by the corresponding calling context. It ensures

$$\frac{\begin{array}{c}\Gamma(v)=\tau \quad \tau_1=\mathit{fresh}(\tau)\\ \phi=\mathit{equate}(\mathit{prime}(\tau),\tau_1)\end{array}}{V;\Gamma;\Delta\vdash v \leadsto v :: \tau_1, \Delta\wedge\phi, \emptyset, \emptyset}\;[\mathbf{Var}] \qquad \frac{\begin{array}{c}V;\Gamma;\Delta\vdash e\leadsto e_1 :: \tau_1,\Delta_1,\Phi,\Upsilon\\ \Gamma(v)=\tau \quad \Delta_2=\mathit{assign}(\Delta_1,\tau,\tau_1)\end{array}}{V;\Gamma;\Delta\vdash v\texttt{=}e\leadsto v\texttt{=}e_1 :: \texttt{Void},\Delta_2,\Phi,\Upsilon}\;[\mathbf{Var{-}Assign}]$$

$$\frac{\begin{array}{c}\Gamma(v)=\texttt{Bool}^b \quad V;\Gamma;\Delta\wedge(b'=1)\vdash e_1\leadsto e_3 :: \tau_1,\Delta_1,\Phi_1,\Upsilon_1\\ \tau=\mathit{fresh}(\tau_1) \quad V;\Gamma;\Delta\wedge(b'=0)\vdash e_2\leadsto e_4 :: \tau_2,\Delta_2,\Phi_2,\Upsilon_2\\ \rho_i=\mathit{rename}(\tau_i,\tau)\;\;\forall i\in\{1,2\} \quad \Delta_3=\rho_1\Delta_1\vee\rho_2\Delta_2 \quad e_5=\texttt{if}\;v\;\texttt{then}\;e_3\;\texttt{else}\;e_4\end{array}}{V;\Gamma;\Delta\vdash\texttt{if}\;v\;\texttt{then}\;e_1\;\texttt{else}\;e_2\leadsto e_5 :: \tau,\Delta_3,\Phi_1\cup\Phi_2,\Upsilon_1\cup\Upsilon_2}\;[\mathbf{If}]$$

$$\frac{\begin{array}{c}\Gamma(v_i)=\tau_i\;\forall i\in 1..n \quad \tau=\;\mathit{fresh}(\hat\tau) \quad U=\bigcup_{i=1}^{k}\mathit{FSV}(\tau_i) \quad \ell=\mathit{fresh}()\\ (\hat\tau\;m(\hat\tau_1\;x_1,\ldots,\hat\tau_n\;x_n)\;\texttt{where}\;\Delta_m;\;\Phi_m;\cdots)\in P\cup P_m \qquad \Phi_m=\{(\ell_1^+:\phi_1),..,(\ell_k^+:\phi_k)\}\\ \rho=\mathit{rename}(\hat\tau,\tau)\uplus\Sigma_{i=1}^{n}\{\mathit{rename}(\hat\tau_i,\tau_i)\} \qquad \mathit{pre}_i\equiv_s(\Delta\mathrel{\approx\!>}\rho\phi_i)\!\downarrow_V\\ \mathit{mkChk}(\mathit{pre}_i,\ell.\ell_i^+,\hat\Phi_i,\Upsilon_i)\;\forall i\in 1..k \qquad \hat\Phi=\bigcup_{i=1}^{k}\hat\Phi_i \quad \Upsilon=\bigcup_{i=1}^{k}\Upsilon_i\end{array}}{V;\Gamma;\Delta\vdash m(v_{1..n})\leadsto\ell:m(v_{1..n}) :: \tau,\Delta\circ_U\rho(\Delta_m),\hat\Phi,\Upsilon}\;[\mathbf{Call}]$$

$$\frac{\begin{array}{c}\mathit{md}=\texttt{t}\;m(\texttt{t}_1\;v_1,\ldots,\texttt{t}_k\;v_k)\;\{e\} \qquad \tau_i=\mathit{fresh}(\texttt{t}_i)\;\forall\;i=1..n \quad \tau=\mathit{fresh}(\texttt{t})\\ V=\bigcup_{i=1}^{k}\mathit{FSV}(\tau_i) \quad W=V\cup\mathit{FSV}(\tau) \qquad \Gamma=\{v_1:\tau_1,\ldots,v_k:\tau_k\}\\ \Delta_{init}=\mathit{init}(\Gamma) \quad V;\Gamma;\Delta_{init}\vdash e\;\leadsto\;e_1 :: \tau,\Delta,\Phi,\Upsilon \quad \mathcal{Q}=\{m\langle W\rangle=\Delta\}\end{array}}{\vdash_I\;\mathit{md}\leadsto\tau\;m(\tau_1\;v_1,\ldots\tau_k v_k)\;\texttt{where}\;m\langle W\rangle;\Phi;\Upsilon\{e_1\}\mid\mathcal{Q}}\;[\mathbf{Mtd{-}Declare}]$$

$$\frac{\mathit{pre}\equiv\;\texttt{true}}{\mathit{mkChk}(\mathit{pre},\ell^+,\emptyset,\emptyset)}\;[\mathbf{MkChk{-}1}] \qquad \frac{\mathit{pre}\equiv\;\texttt{false}}{\mathit{mkChk}(\mathit{pre},\ell^+,\emptyset,\{\ell^+\})}\;[\mathbf{MkChk{-}2}] \qquad \frac{\neg(\mathit{pre}\equiv\texttt{true}\vee\mathit{pre}\equiv\texttt{false})}{\mathit{mkChk}(\mathit{pre},\ell^+,\{\ell^+:\mathit{pre}\},\emptyset)}\;[\mathbf{MkChk{-}3}]$$

Figure 4: Type Inference Rules

that all size variables are expressed in terms of those of the call arguments, and primed variables are used in the post-state of the caller:

$$\Delta \mathrel{\approx\!>} \phi =_{def} (\Delta \Rightarrow \rho(\phi)) \text{ where } \begin{array}{l}\rho=\{s_1\mapsto s_1',\ldots,s_n\mapsto s_n'\};\\ \{s_1,\ldots,s_n\}=\mathit{FSV}(\phi)\end{array}$$

The quantification operator $\downarrow_V$ projects a constraint to the boundary variable set V through universal quantification of the local variables:

$$\phi\!\downarrow_V =_{def} \forall W\cdot\phi \quad \text{where } W=\mathit{FSV}(\phi)-V.$$

Each derived precondition is then used by the relation

$$\mathit{mkChk}(\mathit{pre}, A, B, C)$$

to determine if the corresponding array bound check can be eliminated safely, be left as runtime check, or be decided at a later stage (a partially-safe check). Here, A is a label sequence leading to the specific bound-check, B outputs at most one partially-safe check, and C outputs at most one label sequence identifying a runtime check. Continuing with the above example, we have $\mathit{mkChk}(\mathit{pre}_1, \ell.\texttt{L}, \emptyset, \emptyset)$ and $\mathit{mkChk}(\mathit{pre}_2, \ell.\texttt{H}, \{\ell.\texttt{H} : \mathit{pre}_2\}, \emptyset)$, where ℓ is a new label associated with the call $\texttt{sub(z, i)}$. The above *mkChk* clauses indicate that the low-bound check is safe, and the upper-bound check is partially safe.

For recursive methods, we first employ a fixpoint computation to derive both the method postcondition and a recursive invariant. The invariant captures a size relation to relate the parameters of an arbitrary-nested recursive call with those of the first call. Once the postcondition and the invariant are determined, we can compute the program state at each program point and derive preconditions similarly to the non-recursive case. Details are given in the Section 5.

5. RECURSION ANALYSIS

Our type inference rules effectively determine both a post-condition for each given expression and a set of preconditions for its partially-safe checks. The steps for inferring the body of a recursive method are: (i) determine a fix-point for the postcondition, (ii) determine an invariant on its recursive calls, and (iii) derive preconditions for checks inside recursion. These steps are performed with the help of recursive constraint abstractions.

5.1 Deriving Postcondition

The postcondition in a recursive constraint can be derived via a fix-point approximation procedure pioneered in [7]. Let us consider a constraint abstraction of the form $q\langle n^*, r\rangle$ where n^* denote inputs, while r denotes its output. For simplicity and without loss of generality, let us assume we have a constraint abstraction with two recursive invocations of the following form.

$$q\langle n^*, r\rangle = \phi_0 \vee \phi_1[q\langle s^*, r_1\rangle, q\langle t^*, r_2\rangle]$$

Note that $\phi_1[\_, \_]$ is a formula with two holes containing the two recursive invocations, while ϕ_0 is the base case. According to [7], the fix-point of such an abstraction can be formalised by a series of formulae of the form.

$$\begin{array}{c} q_0\langle n^*, r\rangle = \texttt{false} \\ q_{i+1}\langle n^*, r\rangle = \phi_0 \vee \phi_1[q_i\langle s^*, r_1\rangle, q_i\langle t^*, r_2\rangle] \end{array}$$

For the above fix-point series to converge, we perform approximations via two techniques, known as *hulling* and *widening* from [7]. Hulling combines a set of disjuncts $\bigvee \phi_i$

Methods with Postconditions:

```
Float sumvec(Float[Int^s] a,Int^i i,Int^j j)
  where sumvec⟨s,i,j⟩,...
{ if i>j then 0.0 else {Int v= ℓ1:sub(a,i);
                v+ℓ2:sumvec(a,i+1,j) } }
Float sum(Float[Int^s] a) where sum⟨s⟩,...
{ Int l=ℓ3:len(a); ℓ4:sumvec(a,0,l-1)   }
```

Constraint Abstractions :

$sumvec\langle s,i,j\rangle \;\equiv\; (i{>}j)\vee(i{\le}j\wedge(0{\le}i{<}s)\wedge sumvec\langle s,i{+}1,j\rangle)$
$\quad\equiv_{fix}\; (i{>}j)\vee(0{\le}i{\le}s{-}1,j)$
$sum\langle s\rangle \;\equiv\; s{>}0$

Figure 5: Sum Vector Program

into a conjunct ϕ such that $(\bigvee\phi_i)\Rightarrow\phi$. We refine this process by allowing hulling to be selectively applied to a subset of disjuncts. We denote this process by $\bigvee\phi_i\equiv_h\phi$. Widening takes a formula $\bigvee\bigwedge\phi_i$ and then drops (by replacing with `true`) one or more ϕ_i from it. Let us denote widening by $\equiv_w$. We shall apply each fix-point approximation until we obtain a formula $q_p\langle n^*,r\rangle$ such that $q_{p+1}\langle n^*,r\rangle\Rightarrow q_p\langle n^*,r\rangle$. This test indicates that a fix-point $q_p\langle n^*,r\rangle$ has been reached.

Consider a simple summation program in Figure 5, where the constraint abstractions obtained from our inference rules are also given. To obtain a closed-form postcondition, we apply fix-point analysis to its constraint abstraction, starting with $sumvec_0$. We reach the following fix-point in four iterations:

$$sumvec\langle s,i,j\rangle = (i{>}j)\vee(0{\le}i{\le}j,s{-}1)$$

The detailed process is depicted below. Due to the use of widening, such fix-point approximation *always terminates*.

$$\begin{array}{l}
sumvec_0\langle s,i,j\rangle \;=\; \texttt{false}\\
sumvec_1\langle s,i,j\rangle \;=\; (i{>}j)\vee((0{\le}i{<}s)\wedge(\exists i_1\cdot i_1{=}i{+}1\wedge\texttt{false}))\\
\quad=\;\; (i{>}j)\\
sumvec_2\langle s,i,j\rangle \;=\; (i{>}j)\vee((0{\le}i{<}s)\wedge(\exists i_1\cdot i_1{=}i{+}1\wedge i_1{>}j))\\
\quad=\;\; (i{>}j)\vee(0{\le}i{\le}s)\\
sumvec_3\langle s,i,j\rangle \;=\; (i{>}j)\vee((0{\le}i{<}s)\wedge(\exists i_1\cdot i_1{=}i{+}1\\
\qquad\qquad\wedge((i_1{>}j)\vee(0{\le}i_1{\le}s)))\\
\quad=\;\; (i{>}j)\vee(0{\le}i{\le}s{-}2,j)\vee(0{\le}i{\le}s\wedge i{=}j)\\
\quad\equiv_h\;\; (i{>}j)\vee(0{\le}i{\le}j,s{-}1)\\
sumvec_4\langle s,i,j\rangle \;=\; (i{>}j)\vee((0{\le}i{<}s)\wedge(\exists i_1\cdot i_1{=}i{+}1\\
\qquad\qquad\wedge((i_1{>}j)\vee(0{\le}i_1{\le}j,s{-}1)))\\
\quad\equiv_h\;\; (i{>}j)\vee(0{\le}i{\le}j,s{-}1)\\
\textit{Fix-Point Detected:}\;\; sumvec_4\langle s,i,j\rangle\Rightarrow sumvec_3\langle s,i,j\rangle
\end{array}$$

5.2 Deriving Recursive Invariant

Within each recursive method, we may have checks that must be optimized. To deal with this, we compute another constraint, but this time, for just the input parameters (excluding the results of method). More specifically, we build a one-step size relation to relate the parameters of the next recursive calls with those of the first call. This relation is then analysed via fix-point analysis to derive a multi-steps relation, known as *recursive invariant*. The latter can relate the parameters of an arbitrary recursive call with those of the first call.

One-step relation can be directly extracted from each recursive constraint abstraction. Given the earlier abstraction (of two recursive invocations), $q\langle n^*,r\rangle=\phi_0\vee\phi_1[q\langle s^*,r_1\rangle,q\langle t^*,r_2\rangle]$. We can obtain a one-step relation, named I, that attempts to relate the input n^* with that of its recursive call, $\hat{n}^*$, as shown below.

$$I\langle n^*,\hat{n}^*\rangle=\phi_1[\bigwedge(s=\hat{n})^*,q\langle t^*,r_2\rangle])\vee\phi_1[q\langle s^*,r_1\rangle,\bigwedge(t=\hat{n})^*]$$

With this relation, we can now apply fix-point analysis to obtain:

$$I_1\langle n^*,\hat{n}^*\rangle = I\langle n^*,\hat{n}^*\rangle$$
$$I_{i+1}\langle n^*,\hat{n}^*\rangle = I_i\langle n^*,\hat{n}^*\rangle\vee(\exists z^*\cdot I_i\langle n^*,z^*\rangle\wedge I\langle z^*,\hat{n}^*\rangle)$$

Through a fix-point analysis, we derive the following recursive invariant:

$$sumvecI\langle s,i,j,\hat{s},\hat{i},\hat{j}\rangle=(\hat{s}{=}s)\wedge(\hat{j}{=}j)\wedge(0{\le}i{<}\hat{i}{\le}s,j{+}1)$$

Recursive invariant is important for deriving safety preconditions of checks inside recursive methods, as elaborated next.

5.3 Deriving Precondition

Our inference can derive preconditions for checks inside recursion. Due to recursion, such checks may be encountered multiple times. We propose to separate out the check of the *first* recursive call from the checks of the *rest* of the recursive calls. The reason for this is that recursive invariant that we derive is applicable to all recursive calls, except the first. Consequently, the program state for the first check and the program state for the recursive checks are different. More specifically, consider a check c labelled as ℓ at program context s in a recursive method m with invariant i. Its two preconditions can be derived as follows:

$$\begin{array}{lcl}
\texttt{preFst}(\ell) &=& \forall L\cdot(s\Rightarrow c)\;\texttt{where}\;L=vars(s,c)-V\\
\texttt{preRec}(\ell) &=& \forall L\cdot(s\wedge i\Rightarrow c)\;\texttt{where}\;L=vars(s,c,i)-V
\end{array}$$

For the *sumvec* example, we would derive two sets of preconditions, namely:

$$\begin{array}{l}
\texttt{preFst}(\ell_1.\texttt{L})=(j{<}i)\vee(0{\le}i)\\
\texttt{preFst}(\ell_1.\texttt{H})=(j{<}i)\vee(i{<}s)\\
\texttt{preRec}(\ell_1.\texttt{L})=\texttt{true}\\
\texttt{preRec}(\ell_1.\texttt{H})=(j{<}s)\vee(s{\le}j\wedge i{\le}{-}1)\vee(s{\le}j,i)
\end{array}$$

These preconditins are propagated to the caller of each *sumvec* call. Note that the precondition for (rest of the) recursive checks for ℓ_1.L is totally safe, but the first check of ℓ_1.L can be guarded by a condition $(j{<}i)\vee(0{\le}i)$. These different scenarios of array checks can be exploited by program specialization, so as to maximise the elimination of redundant checks whilst being mindful of the potential for code explosion. We describe such a specialization process next.

6. FLEXIVARIANT SPECIALIZATION

The objective of specialization is to place run-time tests (for unsafe checks) at their respective primitive operations with the objective that array operations become safe, *and* the array checks are done minimally. To this end, we specialize the existing method definitions with information about run-time tests.

To understand the effectiveness of various approaches to specializing method definitions, we examine the following example program:

```
void main()        t2 p(···)          t1 q(···)
{ ···              { ···              { ···
  ℓ5 : p(···);       ℓ3 : q(···);       v1=(ℓ1: sub(a1,i1));
  ···                ···                ℓ2: assign(a2,i2,v1)}
  ℓ6 : q(···)}       ℓ4 : q(···)}
```

Let us assume that the results of inference are as follows:

Pre-Conditions for q			
from ℓ_1		from ℓ_2	
ℓ_1.L	ℓ_1.H	ℓ_2.L	ℓ_2.H
true	ϕ_1	true	ϕ_2

Pre-Conditions for p			
from ℓ_3		from ℓ_4	
$\ell_3.\ell_1$.H	$\ell_3.\ell_2$.H	$\ell_4.\ell_1$.H	$\ell_4.\ell_2$.H
true	ϕ_3	ϕ_4	false

Pre-Conditions for main			
from ℓ_5		from ℓ_6	
$\ell_5.\ell_3.\ell_1$.H	$\ell_5.\ell_4.\ell_2$.H	$\ell_6.\ell_1$.H	$\ell_6.\ell_2$.H
true	true	false	false

This corresponds to the following inferred method headers with partially-safe and unsafe checks.

$$\begin{array}{ll}\texttt{t1 q}(\cdots)\ \texttt{where}\ \cdots & \{\ell_1.\texttt{H}:\phi_1,\ \ell_2.\texttt{H}:\phi_2\},\{\}\\ \texttt{t2 p}(\cdots)\ \texttt{where}\ \cdots & \{\ell_3.\ell_2.\texttt{H}:\phi_3,\ \ell_4.\ell_1.\texttt{H}:\phi_4\},\{\ell_4.\ell_2.\texttt{H}\}\\ \texttt{void main}()\ \texttt{where}\ \cdots & \{\},\{\ell_6.\ell_1.\texttt{H},\ell_6.\ell_2.\texttt{H}\}\end{array}$$

Thus, there are three unsafe checks that must be residualized at run-time, namely $\ell_4.\ell_2$.H, $\ell_6.\ell_1$.H and $\ell_6.\ell_2$.H. The other checks are either safe or partially-safe with the possibility of becoming safe at its caller's context. An aggressive approach to eliminating checks is *polyvariant specialization*. This aims at creating *multiple* specialized methods for each method definition, such that each specialized version of a method has a different set of array checks being eliminated. Its application on our example program yields the following result:

```
void main()      t2 p(···)               t1 q_1(···) where .., φ1∧φ2
{ ···            where .., φ3∧φ4         { ···
  p(···);          { ···                   v1 = (sub(a1, i1));
  ···                q_1(···);             assign(a2, i2, v1) }
  q_3(···)}          ···
                     q_2(···)}           t1 q_3(···)  where .., true
                                         { ···
t1 q_2(···)  where .., φ1                  v1 = (if (i1 < len(a1))
{ ···                                           then sub(a1, i1)
  v1 = sub(a1, i1);                             else error);
  if (i2 < len(a2)) then                   if (i2 < len(a2)) then
    assign(a2, i2, v1)                       assign(a2, i2, v1)
  else error }                             else error }
```

Note that three versions of q have been created to handle its three calls under different calling contexts.

Our *flexivariant* program specializer allows such full polyvariance to be achieved by never attempting to weaken any of the configurations encountered. It is also possible to obtain monovariant specialization. This can be obtained by weakening each configuration encountered to its most conservative variant with maximal unsafe checks. For this example, our monovariant specializer will weaken the configurations of both q_1 and q_2 to q_3. Though q_3 is the weakest specialised variant, it still has two low bound checks eliminated.

Another feature of this flexivariant specialization scheme is its ability to trade-off optimization for a reduction in code size. Furthermore, it is possible to achieve such trade-offs with minimal loss in performance. For example, if it can be determined that q_1 configuration occurs *infrequently*, we may weaken it into q_2 to save on code size with little loss in performance.

Flexivariant specialization of a program P into an optimized program S is declared as follows:

$$\triangleright_{flex}\ \texttt{P} \rightharpoonup \texttt{S}$$

Specializing a method requires information about the set of runtime tests to which calls in the method body may lead. Thus, a specialized method can be identified by a triple comprising the original method name, a set of label sequences associated with the relevant runtime tests, and a new method name uniquely defined by the first two components of the triple. We call such a triple a *specialization signature* (or *signature* in short), and a set containing such signatures a *specialization cache* (or *cache* in short).

$$\begin{array}{rcl}(m,\varsigma,\hat{m}) & \in & \textbf{SSig} \ = \ \textbf{MName}\times\textbf{LSet}\times\textbf{MName}\\ \sigma,\sigma_Y,\sigma_N,\hat{\sigma_N} & \in & \textbf{SCache} \ = \ \mathcal{P}(\textbf{SSig})\\ \varsigma & \in & \textbf{LSet} \ = \ \mathcal{P}(\textbf{Label}^+)\end{array}$$

The specialization of an expression is defined by:

$$\texttt{P},\ \sigma,\ \varsigma\ \triangleright^e_{flex}\ e\ \rightharpoonup\ e_1,\ \sigma_N$$

The specialization cache σ drives the process, while ς contains the checks to be residualized. New specialization points created during specialization are stored in σ_N. We highlight the most important specialization rules below.

Array operation is specialized in [Spec−Prim] by calling the respective primitive method without array checks under the condition that the combined runtime checks for this operation, e_1, is true.

$$\frac{\begin{array}{c}\boxed{\textbf{Spec}-\textbf{Prim}}\\ \tau\ m(\tau_1\ x_1,\ldots,\tau_n\ x_n)\ \ \text{where}\ \Delta,\Phi,C\ \in\ P_m\\ \rho=[x_1\mapsto v_1,\ldots,x_n\mapsto v_n]\\ e_1=\bigwedge\{\rho\ e\mid\ \ell.c\in\varsigma\wedge(c:e)\in C\}\\ e_2=\texttt{if}\ e_1\ \texttt{then}\ m(v_1,\ldots,v_k)\ \texttt{else error}\\ e_3=m(v_1,\ldots,v_k)\lhd(e_1=\texttt{true})\rhd e_2\end{array}}{\texttt{P},\ \sigma,\ \varsigma\ \triangleright^e_{flex}\ (\ell:m(v_1,\ldots,v_k))\rightharpoonup e_3,\ \emptyset}$$

Here, a label sequence of the form $\ell.c$ occurring in the set ς represents an array check to be residualized. Its code is available at the corresponding primitive method declaration. Variable substitution is needed to residualize the code. All codes thus generated are combined as a conjunct, named e_1, which is then wrapped as a runtime test for the primitive call to m. If the runtime set is empty – signified by e_1 being true – the m call will not be wrapped by a conditional.

Similarly, user-defined methods are specialized with respect to the set of runtime tests ([Spec−Call₁]). Weakening of configurations by $\mathcal{W}$ may enlarge this set of runtime tests. Specialization produces a signature for this specialized method if the latter has not been recorded in the current cache. Otherwise, it reuses the specialised method that has been recorded previously, as specified in [Spec−Call₂].

$$\frac{\begin{array}{c}\boxed{\textbf{Spec}-\textbf{Call}_\textbf{1}}\\ (\tau\ m(\tau_1\ x_1,\ldots\tau_k x_k)\ \texttt{where}\ \Delta,\Phi,\Upsilon\ \{e\})\in\texttt{P}\\ \varsigma_2=\mathcal{W}(m,\varsigma_1)\quad \varsigma_1=\{\ell^+\mid\ell_1.\ell^+\in\varsigma\}\cup\Upsilon\\ (m,\varsigma_2,\_)\notin\sigma\qquad m_s=genName(m,\varsigma_2)\end{array}}{\begin{array}{c}\texttt{P},\ \sigma,\ \varsigma\ \triangleright^e_{flex}\ (\ell_1:m(v_1,\ldots,v_k))\\ \rightharpoonup\ m_s(v_1,\ldots,v_k),\ \{(m,\varsigma_1,m_s)\}\end{array}}$$

$$\frac{\begin{array}{c}\boxed{\textbf{Spec}-\textbf{Call}_\textbf{2}}\\ (\tau\ m(\tau_1\ x_1,\ldots\tau_k x_k)\ \texttt{where}\ \Delta,\Phi,\Upsilon\{e\})\in\texttt{P}\\ \varsigma_1=\{\ell^+\mid\ell_1.\ell^+\in\varsigma\}\cup\Upsilon\qquad (m,\mathcal{W}(m,\varsigma_1),m_s)\in\sigma\end{array}}{\texttt{P},\ \sigma,\ \varsigma\ \triangleright^e_{flex}\ (\ell_1:m(v_1,\ldots,v_k))\ \rightharpoonup\ m_s(v_1,\ldots,v_k),\ \emptyset}$$

The complete formalization of flexivariant specializer, as well as its correctness, can be found in the technical report [29].

7. SOUNDNESS OF INFERENCE SYSTEM

The soundness of our type inference is defined with respect to a type checking system and a specialization process. After type inference (that includes fixpoint analysis), the inferred program must be *specialized* to include the runtime tests discovered during inference, before it becomes well-typed.

We state the soundness of our inference system below, and refer te reader to the technical report [29] for details on the proof and the specialization process.

THEOREM 1 (SOUNDNESS). *Let P be a program such that $(P_m \vdash_I P \rightsquigarrow P_I)$ under type inference. Let $(\triangleright_{flex} P_I \rightharpoonup P_T)$ be the specialization of P_I to P_T guided by the inferred runtime tests. Then P_T is well-typed.*

As a special case, if no unsafe check is discovered during inference then P_I is well-typed. However, if unsafe checks are discovered, the use of label sequences (*eg.*, $\ell_6.\ell_1.\mathtt{H}$) to identify array checks also enables *debugging feedback*. Specifically, our analysis can pin-point the exact location of each unsafe check based on the calling hierarchy up until an unsatisfied precondition.

8. DERIVING SMALLER FORMULAE

An important property of program analysis is efficiency, and this is particularly so for an inference system based on Presburger arithmetic. Presburger arithmetic can give highly accurate analysis (with disjunctions, quantifiers and logical connectives) but has double-exponential complexity, namely $2^{2^{cn}}$ where n is the size of its formulae. Being a summary-based approach, we use a bounded number of size variables at each method boundary. With this bound, the main proviso for efficieny is to ensure that the pre and post-conditions are kept small in size.

A major reason for large formulae is the presence of disjuncts. A derived postcondition can be weakened through the *hulling* of each set of selected disjuncts. For the derivation of a precondition, it is only safe to strengthen and this can be done by *simplifying* it (via *gist* [21]) with respect to the symbolic program state of the location where its check originated.

Given a check c which occurs at a location with program state s and local variables V_L, we have derived earlier the weakest precondition using $\mathtt{pre} = (\forall V_L \cdot \neg s \vee c)$. This derived precondition may be large due to the presence of disjunctive formulae in $\neg s$. To derive smaller preconditions, we may simplify $\mathtt{pre}$ using a valid state s_1 for which $(\exists V_L \cdot s) \Rightarrow s_1$ holds.

One such s_1 that we have already used is the *type invariant* $\mathtt{inv}$ at method entry. Let us refer to this technique using (*gist* $\mathtt{pre}$ *given* $\mathtt{inv}$) as *weak pre-derivation.*

A second technique is to use $\exists V_L \cdot s$ itself. Let us refer to this technique using (*gist* $\mathtt{pre}$ *given* $\exists V_L \cdot s$) as *strong prederivation*. This technique would strip off all the *avoidance conditions* from the derived precondition, which may result in some loss of precision. To recover this loss of precision, we also propose a third technique, called *selective prederivation*, which would first obtain a variant of $\exists V_L \cdot s$ that is weakened by removing conditional tests from s. For example, consider a symbolic program state from the recursive `sumvec` method:

$$\exists\ \hat{i} \cdot s{>}0 \wedge \hat{i}{\leq}j \wedge (0{\leq}i{<}\hat{i}{\leq}s, j{+}1)$$

After stripping off its conditional test, $\hat{i}{\leq}j$, we would obtain a weaker state:

$$\exists\ \hat{i} \cdot s{>}0 \wedge (0{\leq}i{<}\hat{i}{\leq}s, j{+}1)$$

Simplifying the precondition of $(j{<}s) \vee (s{\leq}j \wedge i{\leq}{-1}) \vee (0{\leq}i \wedge s{\leq}j, i)$ with this program state results in a much smaller precondition, namely $j{<}s$, that is also obtained by strong prederivation. This is in contrast to $(j{<}s) \vee (s{\leq}j \wedge i{\leq}{-1}) \vee (s{\leq}j, i)$ that is obtained by weak prederivation. We found this third technique to have a reasonable compromise between efficiency and precision. When compared to the weak prederivation technique, we were able to reduce the size of preconditions on average by 63.4% for selective prederivation and by 81.8% for strong prederivation (for the set of benchmark programs we evaluated). Furthermore, we achieved a significant reduction in the inference times for larger programs; some programs failing to complete otherwise!

9. IMPLEMENTATION

We have constructed the proposed modular inference system together with a program specializer. The output from our specializer can be validated by a separate type checking system that we have also built. Our implementation includes a pre-processing phase that allows us to use some syntactic sugars for IMP. The entire prototype system was built using Glasgow Haskell compiler[20] extended with the Omega[21] constraint solving library.

Our main objective for building this prototype is to show that such an advanced form of inference mechanism is indeed viable. We tested our system on a micro-benchmark suite (small programs with challenging recursion), and also the SciMark (`FFT`,`LU`,`SOR`) and Linpack benchmark suites[19, 8]). Figure 6 summarises the statistics obtained for each program that we inferred. Our test platform is a Pentium 2.8 GHz system with 1GBytes main memory, running Red Hat Linux 9.0.

To quantify the analysis complexity of the benchmark programs, we counted the program size (column 2) and also the number of static checks present in the original program (column 3). Since a partially-safe check is re-analyzed after propagating up its calling hierarchy, we also present the number of checks encountered during inference (column 4). Note that the latter is always higher than the number of static checks, and the difference represents the price to pay for being able to reason locally about checks.

Due to the precision of our inference system, we were able to eliminate 100% of array checks for all the programs we tested, except for `sentinel` and `foo`. The `sentinel` example illustrates a pattern where some checks cannot be eliminated by our method, since it make use of a sentinel/guard against falling off one end of the array. Like [30, 28], we are unable to capture the existential property that is required for check elimination. For the `foo` example, strong prederivation and selective prederivation eliminate 54% and 69%, respectively, of the dynamic checks. Our method is able to eliminate accesses through array indirections that account for 22% of the checks for `sparse multiply` and 0.06% of the checks for

Benchmark Programs	Source (lines)	Static Checks Prim.	Static Checks Inferred	Checking (secs)	Inference (secs) Weak	Inference (secs) Selective	Inference (secs) Strong	Dynamic Checks Elim.
binary search	50	2	5	0.17	1.84	1.79	1.81	100%
bubble sort	59	12	21	0.43	1.55	1.47	1.51	100%
foo	28	5	8	0.39	0.66	0.67	0.87	54%/69%
hanoi tower	38	16	31	3.73	11.74	11.53	11.47	100%
merge sort	80	16	64	7.70	11.21	16.01	13.07	100%
queen	45	8	13	0.52	2.13	2.11	2.10	100%
quick sort	67	12	30	0.38	1.92	1.92	1.76	100%
sentinel	26	4	9	0.05	0.18	0.15	0.16	50%
sparse multiply	46	12	20	3.27	22.61	17.37	7.09	100%
sumvec	33	2	4	0.11	0.51	0.48	0.47	100%
FFT	336	62	278	9.58	*	58.02	28.74	100%
LU Decomp.	191	80	160	13.10	137.1	93.31	72.91	100%
SOR	84	24	44	1.15	7.18	4.67	3.8	100%
Linpack	903	166	368	42.26	*	360.1	162.2	100%

Figure 6: Comparative Statistics

`Linpack` benchmark program. The size of constraint annotations are on average around 15% of the size of the source program. Thus, our inference eliminates sizeable mental effort required from programmers with only a type checker.

The time taken for inference includes parsing, preprocessing (to eliminate syntactic sugars), modular type inference and specialization, while the time taken for checking involves parsing and type checking. In almost all cases, strong prederivation takes less time than selective prederivation, followed by weak prederivation. As an exception, the increased precision of weak prederivation allows a faster analysis of `mergesort`, since some bound checks are proved redundant at an earlier point than the other two prederivation methods. On the other hand, for medium to large-sized programs we found that it is crucial to use either selective or strong prederivation; weak prederivation does not scale up as some programs cannot be inferred due to an out-of-memory error (denoted by *).

10. RELATED WORKS

Traditionally, data-flow analysis techniques have been employed to gather information for the purpose of identifying redundant array checks. Within the scope of intra-procedural analysis, these techniques are also used to gather anticipatable information for the purpose of hoisting partially-redundant checks to more profitable locations. The techniques have gradually evolved in sophistication, from the use of family of checks in [17], to the use of difference constraints in [2]. Most recently, Venet and Brat advocated for an abstract interpretation with difference constraints to eliminate about 80% of bound checks for large scale C programs [27]. Also, Dor *et al* advocated for linear constraints, expressed using pre/post conditions, to help determine the safety of C pointers to string buffers [9]. Their inference result is, however, less precise than user-supplied annotations. This is likely due to the absence of disjunction and path-sensitivity during inference.

To identify redundant checks more accurately, verification-based methods have been advocated by Suzuki and Ishihata [25], Necula and Lee [18] and Xu *et al* [30]; whilst Cousot and Halbwachs [7] have advocated the use of abstract interpretation techniques. Furthermore, Rugina and Rinard [24] proposed an analysis method (using linear programming) to synthesize polynomial symbolic bounds without fix-point iteration. While efficient, fixing a target form (without disjunction) for the symbolic bound results in loss of precision. Lastly, Xi and Pfenning have advocated the use of dependent types for array bound check elimination [28]. Their approach is limited to totally redundant checks. Moreover, the onus for supplying suitable dependent types rests squarely on the programmers, as only a type checker is available.

Precondition derivation with respect to a postcondition (or check) has been formulated via generating its *Verification Condition* (VC) by Flanagan *et al* [10, 11]. Their focus was to obtain compact VCs whose size is worst-case quadratic to the size of the source. However, they do not attempt to make preconditions and postconditions any *smaller* through strengthening and weakening, respectively. Furthermore, these VCs are for totally-redundant checks. In contrast, our technique stresses on modularity and deal with inter-procedural analysis over recursive methods, whereas they focus on intra-procedural analysis and loops.

The static analyzer that has been described by Cousot *et al* [1] succeeds in analyzing a program of 75 kloc with no false alarm. It achieves this by varying the precision of arithmetic abstract domains from interval domain to ellipsoid domain. It also uses a decision tree abstract domain and trace partitioning for path-sensitivity. These relational domains operate on packs of variables for efficiency reasons. On the other hand, our analysis maintains path-sensitivity and the same level of precision over the entire program by exploiting modularity. Being a summary-based approach, we have a bounded number of variables at method boundary and we further ensure that preconditions are kept small via suitable prederivation. Modularity has also been recognized as an important step for static program analyses to scale up to precise analysis of large programs [6] and our proposal is a solution in this direction.

The initial idea for deriving preconditions for partially redundant checks was informally described by the authors[5] for a first-order functional language. The current paper formalises modular inference and specialization processes for an imperative language setting. Most importantly, we have also engineered a system that trade-off precision for scalability and have proven its soundness.

11. CONCLUDING REMARKS

We have proposed a new inference mechanism for a dependent type system with size relations. Our approach captures postcondition in the presence of imperative updates, and derives safety preconditions for each check encountered. Both the postcondition and safety precondition are propagated interprocedurally, though in opposite directions. Recursive methods are also handled through a fix-point analysis on constraint abstraction derived via inference. The resulting analysis is not only flow and context-sensitive, but is also path-sensitive. It can capture symbolic program states between local variables, inputs and outputs. Initial experiences with a prototype implementation suggest that such an advanced form of type inference is both precise and efficient.

Just as the present analysis is empowered by the use of Presburger arithmetic, it is inevitably limited by the linearity of expressible constraints. However, by first subjecting the original program to pre-processing such as partial evaluation (using constant propagation and loop unrolling), our analysis can discover more linear constraints, and thus further improve its effectiveness. Moreover, our semantics-based type system ensures that the analysis outcome is not affected by syntactic differences of subject programs. For example, meaning-preserving transformations (e.g. $a + b = b + a$) and occurrences of intermediate functions do not alter the result of our analysis.

12. REFERENCES

[1] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN PLDI*, 2003.

[2] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *ACM SIGPLAN PLDI*, pages 321–333, 2000.

[3] R. Bodik, R. Gupta, and M.L. Soffa. Complete removal of redundant expressions. In *ACM SIGPLAN PLDI*, pages 1–14, June 1998.

[4] W.N. Chin and S.C. Khoo. Calculating sized types. In *ACM SIGPLAN PEPM*, pages 62–72, Boston, Massachusetts, United States, January 2000.

[5] W.N. Chin, S.C. Khoo, and Dana N. Xu. Deriving pre-conditions for array bound check elimination. In *Programs as Data Objects II*, pages 2–24, Aarhus, Denmark, May 2001. Springer Verlag.

[6] P. Cousot and R. Cousot. Modular static program analysis. In *Intl Symp. on Compiler Construction*. ACM Press, 2002.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM SIGPLAN/SIGACT POPL*, pages 84–96. ACM Press, 1978.

[8] J.J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

[9] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *ACM SIGPLAN PLDI*, pages 155–167. ACM Press, 2003.

[10] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *ACM SIGPLAN/SIGACT POPL*, pages 191–202. ACM Press, 2002.

[11] C. Flanagan and J.B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *ACM SIGPLAN PLDI*, pages 193–205, 2001.

[12] R. Gupta. A fresh look at optimizing array bound checking. In *ACM SIGPLAN PLDI*, pages 272–282, New York, June 1990.

[13] J Gustavsson and J Svenningsson. Constraint abstractions. In *Programs as Data Objects II*, pages 63–83, Aarhus, Denmark, May 2001.

[14] E.C.R. Hehner. *A Practical Theory of Programming*. Springer-Verlag, 1993.

[15] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12:576–580, 1969.

[16] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *23rd ACM Principles of Programming Languages Conference*, pages 410–423, St. Petersburg Beach, Florida, January 1996. ACM Press.

[17] P Kolte and M Wolfe. Elimination of redundant array subscript range checks. In *ACM SIGPLAN PLDI*, pages 270–278. ACM Press, June 1995.

[18] G Necula and P. Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN PLDI*, pages 333–344, 1998.

[19] National Institue of Standards and Technology. Java scimark benchmark for scientific computing. `http://math.nist.gov/scimark2/`.

[20] S Peyton-Jones and et al. Glasgow Haskell Compiler. http://www.haskell.org/ghc.

[21] W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of ACM*, 8:102–114, 1992.

[22] A Rountev and BG. Ryder. Points-to and side-effect analyses for programs built from precompiled libraries. In *Intl Symp. on Compiler Construction*. LNCS, 2001.

[23] Eric Ruf. Effective synchronization removal for java. In *ACM SIGPLAN PLDI*. ACM Press, June 2000.

[24] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM SIGPLAN PLDI*, pages 182–195. ACM Press, June 2000.

[25] N. Suzuki and K. Ishihata. Implementation of array bound checker. In *ACM SIGPLAN/SIGACT POPL*, pages 132–143, 1977.

[26] J. Cocke V. Markstein and P. Markstein. Optimization of range checking. In *Proceedings of a Symposium on Compiler Optimization*, pages 114–119, June 1982.

[27] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *ACM SIGPLAN PLDI*, pages 231–242. ACM Press, 2004.

[28] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM SIGPLAN PLDI*, pages 249–257. ACM Press, June 1998.

[29] D.N. Xu, C. Popeea, S.C. Khoo, and W.N. Chin. Scalable and precise inference for array bound checks elimination. Technical report, National University of Singapore, 2004. `http://www.comp.nus.edu.sg/~khoosc/research/array.ps`.

[30] Z. Xu, B.P. Miller, and T. Reps. Safety checking of machine code. In *ACM SIGPLAN PLDI*, pages 70–82. ACM Press, June 2000.

[31] J Zhu. Symbolic pointer analysis. In *IEEE ICCAD*, 2002.