# A Practical and Precise Inference and Specializer for Array Bound Checks Elimination

Corneliu Popeea
Department of Computer Science
National University of Singapore
corneliu@comp.nus.edu.sg

Dana N. Xu
Computer Laboratory
University of Cambridge
nx200@cam.ac.uk

Wei-Ngan Chin
Department of Computer Science
National University of Singapore
chinwn@comp.nus.edu.sg

## Abstract

Arrays are intensively used in many software programs, including those in the popular graphics and game programming domains. Although the problem of eliminating redundant array bound checks has been studied for a long time, there are few works that attempt to be both aggressively precise and practical. We propose an inference mechanism that achieves both aims by combining a forward relational analysis with a backward precondition derivation. Our inference algorithm works for a core imperative language with assignments, and analyses each method once through a summary-based approach. Our inference is *precise* as it is both path and context sensitive. Through a novel technique that can strengthen preconditions, we can selectively reduce the sizes of formulae to support a *practical* inference algorithm. Moreover, we subject each inferred program to a flexivariant specialization that can achieve good tradeoff between elimination of array checks and 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 desired system is viable.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

***General Terms*** Languages, Theory, Verification

***Keywords*** Safety verification, Dependent types, Size properties

## 1. Introduction

Array bound check optimization has been extensively investigated over the last three decades [40, 11, 19], with renewed interests as recently as [3, 45, 14, 41, 30]. While the successful elimination of bound checks can bring about measurable efficiency gain, the importance of check optimization goes beyond this direct gain. In safety-oriented languages, such as Java, all bound violation must be faithfully reported under precise exception handling mechanism. Thus, check optimization is even more important for run-time efficiency under such constraints. For example, the code motion technique is severely hindered by potential array bound violations.

Most array optimization techniques (*e.g.* [40, 11, 43]) 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)[4], 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 precise analyses.

In this paper, we propose a practical 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 form of dependent type [23, 7] that can capture symbolic program states using a relational analysis. For practical reasons, we currently make use of an existing Presburger arithmetic solver [35] that is quite efficient. Nevertheless, our proposal allows this solver to be replaced by a more appropriate one, if desired. The key contributions of this paper include:

- **Forward with Backward Combination :** We propose a novel combination of forward plus backward analysis that can be practical and precise. This combination performs the more expensive forward fix-point analysis only once per method, but proceeds to derive individual safety precondition for each check across procedural boundary. We provide the *first formalization and implementation* of this combination technique for an imperative language. (Sec 2, 4 and 5)
- **Smaller Preconditions :** To obtain a practical analysis, we devise a new technique to make *formulae smaller* by suitable strengthening of preconditions (Sec 6). This approach trades (some) precision for speed and has been vindicated by experiments with our prototype inference system.
- **Integration with Specializer:** 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 [5, 9, 44], their integration with program specializer is hardly investigated. We show how a *flexivariant* specializer could be used to insert runtime test for each array check that has been classified as unsafe (Sec 7).


*PEPM'08,* January 7–8, 2008, San Francisco, California, USA.

- **Indirection Arrays :** Our approach can analyse the *bounds of elements* inside an array. This is important for eliminating array checks for a class of programs where indexes are kept inside indirection arrays (Sec 8). Past techniques on array bound checks elimination have largely ignored this aspect.
- **Prototype :** To confirm the viability of our approach, we have built a prototype inference and specializer system (Sec 9).

## 2. Overview

A key feature of our approach is the three-way classification of checks. 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 the program state $s$ at the location of this check. 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:

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

  The precondition can be derived using $pre = (\forall L \cdot \neg s \vee c)$, where $L$ is the set of local variables, denoted by $vars(s, c) - V$. The function $vars$ returns the free variables used in $s$ and $c$.
- $c$ is *unsafe*, if `false` is the only precondition that can be found to satisfy (1). In this case, the analysis will (conservatively) conclude that the check $c$ may fail at runtime.

Partially-safe checks are special in that they can be propagated across methods from callees to callers. This mechanism can further exploit the program states at callers' sites for the elimination of checks. While the above classification is general and may be applicable to any kind of checks, in this paper we shall be focusing exclusively on array-related checks.

Let us highlight the above check classification using the `foo` example at the top of Figure 1. In this example, `randInt` returns a random integer, while `abs` converts each number into its positive counterpart. The set of parameters $V$ at method boundary is $\{\mathtt{a}, \mathtt{j}, \mathtt{n}\}$ where `a` is an array with indices from `0` to $\mathtt{len}(\mathtt{a}){-}\mathtt{1}$. The `foo` method contains two array accesses at locations $\ell_1$ and $\ell_2$. The symbolic program states (`sps`) at these sites may be affected by the type invariants [1], conditionals, imperative updates and by prior calls. Computing the states for the method entry $\ell_0$ and the locations $\ell_1$ and $\ell_2$, we get:

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

Based on the earlier classification of checks, we can establish that the low-bound checks (at $\ell_1$ and $\ell_2$) are safe, since:

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

For the high-bound checks (denoted by $\ell_1$.`H` and $\ell_2$.`H`), we derive (the weakest) preconditions through universal quantification of the local variables, as follows:

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

These derived preconditions may be the weakest, but they do not take into account the type invariant and thus are larger than needed. The type invariant $\mathtt{len}(\mathtt{a}) > \mathtt{0}$ can be used to simplify $\mathtt{pre}(\ell_2.\mathtt{H})$ to *False* and $\mathtt{pre}(\ell_1\mathtt{.H})$ to $(\mathtt{j<=len(a)-2} \vee \mathtt{n<=j} \wedge \mathtt{j+1>=len(a)})$. The last formula contains a disjunct `(j<=len(a)-2)` for satisfying the check, and a second disjunct $(\mathtt{n<=j} \wedge \mathtt{j+1>=len(a)})$ for avoiding the check (when the conditional test is unsatisfiable). In general, the simplification may drop disjuncts that violate the type invariant `(len(a)<=0)` or remove conditions already present in the type invariant `(len(a)>0)`. We perform each simplification of a formula $\phi_1$ under type invariant $\phi_2$ by the operation (*gist* $\phi_1$ *given* $\phi_2$). This *gist* operation yields a simplified term $\phi_3$ such that $\phi_3 \wedge \phi_2 \equiv \phi_1 \wedge \phi_2$ and was introduced in [36].

While a goal of our analysis is to obtain weaker preconditions for precision, this might impact the scalability of our analysis. To obtain smaller (but stronger) preconditions, we apply a similar simplification based on the *gist* operation, but more aggressive. For example, simplifying $\mathtt{pre}(\ell_1\mathtt{.H})$ with respect to the program state of the check $\exists\, i \cdot \mathtt{sps}(\ell_1)$ yields a smaller precondition `(j<=len(a)-2)` without the disjunct that allows avoiding the check. Our proposal trades off precision for performance and is crucial for overcoming the intractability of solving large Presburger arithmetic formulae.

```
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 (j≤s−2) ∨ (n≤j ∧ j+1≥s) ;
    {ℓ1.H : (j≤s−2) ∨ (n≤j ∧ j+1≥s)}; {ℓ2.H} {···}

              ⇓ Specialization

Float foo(Float[Int^s] a, Int^j j, Int^n n)
 where (j≤s−2) ∨ (n≤j∧j+1≥s); (j≤s−2) ∨ (n≤j∧j+1≥s)
 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.** Inference and Specialization : An Example

One feature of our 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: a postcondition $\Delta$, a set of preconditions $\Phi$, a set of unsafe checks $\Upsilon$, and annotated types $\tau_0, .., \tau_k$.

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

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 $\{\ldots\}$. During the actual

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

$$
\begin{array}{rcl}
\mathtt{P} & ::= & meth^* \\
meth & ::= & \mathtt{t}\ mn\ (\mathtt{ref}\ \mathtt{t}\ v)^*\ \{e\} \\
e & ::= & c \mid v \mid \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid v\ =\ e \\
 & & \mid \mathtt{t}\ v{=}e_1\ ;\ e_2 \mid mn(v^*) \\
\mathtt{t} & ::= & \underline{\mathtt{t}} \mid \underline{\mathtt{t}}[\mathtt{Int},\ldots,\mathtt{Int}] \\
\underline{\mathtt{t}} & ::= & \mathtt{Void} \mid \mathtt{Int} \mid \mathtt{Bool} \mid \mathtt{Float}
\end{array}
$$

**Figure 2.** Source IMP language

$$
\begin{array}{rcll}
\mathtt{P} & ::= & meth^* & \\
meth & ::= & \tau\ mn\ ((\mathtt{ref}\ \tau\ v)^*)\ \mathtt{where}\ \Delta;\Phi;\Upsilon\ \{e\} & \\
prim & ::= & \tau\ mn\ ((\tau\ v)^*)\ \mathtt{where}\ \Delta;\Phi;\mathtt{C} & \\
e & ::= & \cdots \mid \ell : mn\ (v^*) & \\
\tau,\hat{\tau} & ::= & \underline{\tau} \mid \underline{\tau}[\mathtt{Int}^{s_1},\ldots,\mathtt{Int}^{s_k}] & \\
\underline{\tau} & \in & \mathbf{PrimAnnType} & \\
 & ::= & \mathtt{Void} \mid \mathtt{Int}^s \mid \mathtt{Bool}^s \mid \mathtt{Float} & \\
\Phi & ::= & \{\ (l^+ : \phi)^*\ \} & \text{(Labelled Preconditions)} \\
\Upsilon & ::= & \{\ (l^+)^*\ \} & \text{(Unsafe Checks)} \\
\mathtt{C} & ::= & \{\ (l^+ : e)^*\ \} & \text{(Labelled Runtime Checks)} \\
\ell & \in & \mathbf{Label} & \\
\ell^+ & ::= & \ell \mid \ell_1.\cdots.\ell_n & \text{(Label Sequences)} \\
\phi,\Delta & ::= & a_1{=}a_2 \mid a_1{\leq}a_2 \mid \phi_1{\wedge}\phi_2 \mid \phi_1{\vee}\phi_2 \mid q\langle s^*\rangle & \\
a & ::= & c \mid s \mid s' \mid c{*}a \mid a_1{+}a_2 & \\
 & & \text{where } s \text{ denotes a size variable} & \\
\mathcal{Q} & ::= & \{(q\langle s^*\rangle = \phi)^*\} &
\end{array}
$$

**Figure 3.** Inferred $\text{IMP}_I$ Language

inference, we use size variables instead of program variables. For example, size variables $s$, $j$ and $n$ denote `len(a)`, `j` and `n` respectively.

The inference result is 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 $\Delta$ and a precondition $\phi_{pre}$:

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

The precondition $\phi_{pre}$ is a conjunction of checks from $\Phi$ that are guaranteed safe at each call site. For example, if `pre(`$\ell_1$`.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. Note that $\Delta \equiv \phi_{pre}$ holds for this particular example, but in general the two formulae may be different. This is so as postcondition is computed using over-approximation, while precondition is computed using under-approximation. Moreover, postcondition may capture its method's result(s), but not so for precondition.

Well-typed programs are safe in that no array bound errors are ever encountered by any array access during program execution. This safety property is guaranteed by either the program context (for array checks $\ell_1$.L and $\ell_2$.L), or the precondition of each method (for array check $\ell_1$.H) or the inserted runtime test (for $\ell_2$.H). In the rest of this paper, we shall formalise a type inference system to derive well-typed programs for a core imperative language.

## 3. An Imperative Language

To formalise our type inference we first introduce a source language IMP (see Figure 2), where types, denoted by $\mathtt{t}$ and $\underline{\mathtt{t}}$, 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 with the help of pass-by-reference parameters.

### 3.1 Target Language

The target of our inference system is a corresponding imperative language with dependent types where types may be annotated with size variables. For example, a boolean value can be denoted by $\mathtt{Bool}^b$ where $b = 0$ represents *False* and $b = 1$ represents *True*; an integer value can be denoted by $\mathtt{Int}^n$ with $n$ to denote its integer value, while $\mathtt{Float}[\mathtt{Int}^s]$ can denote an array of floats with $s$ elements. Input-output relation between size variables from method parameters and result is captured after the `where` keyword:

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

Note that *True* for `randInt` signifies that $r$ is unbounded. Also, non-trivial size relations can be supported through disjunctive formulae. 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 the notation $no\mathcal{X}\{a,b\}{\equiv}(a'{=}a{\wedge}b'{=}b)$ as a shorthand for "no change in state". This no-change in state occurs mostly for parameters that are passed by value. Pass-by-reference parameters are also supported in our language using the `ref` keyword.

Figure 3 summarises a language with dependent type, called $\text{IMP}_I$, which is designed to be the target of our inference. Each method declaration captures three information: an input-output relation (postcondition) $\Delta$, a set $\Phi$ that contains a precondition for each partially-safe check, and a set of label sequences $\Upsilon$, each sequence representing the location of an unsafe check. The labels from $\Phi$ and $\Upsilon$ 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`, the triple $(\Delta,\Phi,\Upsilon)$ can be derived via inference of the method body (since the triple for each method called in `mn` are already inferred.) To support recursive methods, we make use of *constraint abstractions* (adopted from [20]). For each mutual-recursive method, we first derive a (recursive) constraint abstraction $\mathcal{Q}$ 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 [11] 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 solver [35] (with support for universal and existential quantifications) to eliminate local variables or simplify formulae.

Primitive methods (denoted by *prim* in Figure 3) lack a method body and are instead annotated with a postcondition and a set of preconditions to support type inference. A primitive is also annotated with a set of runtime tests `C` for use by the specializer : if some precondition is not satisfied at a primitive call site, its corresponding runtime test is to be inserted. Array operations are implemented as calls to primitive methods. For example, 1-dimensional array operations with element type $\tau$ are shown below:

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

The primitive `newArr` returns a new array with all elements initialized to the value `v`, `len` returns the length of the array, `sub` returns an array element from the specified index `i`, while `assign` updates the specified array element with the value `v`. For example, an array access $\mathtt{a}[\mathtt{i}]$ is (automatically) converted to $\mathtt{sub}(\mathtt{a},\mathtt{i})$, while an array update $\mathtt{a}[\mathtt{i}] = \mathtt{v}$ is converted to the primitive call $\mathtt{assign}(\mathtt{a},\mathtt{i},\mathtt{v})$.

## 4. Type Inference Rules

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 $P_m \vdash_I P \leadsto P_I$. 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:

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

Here $V$ is a set of size variables (called *boundary variables*) available at the boundary of the method in which the expression $e$ resides. $\Gamma$ is a type environment mapping program variables to their annotated types. The above judgement states that $e$ will be transformed into $e_1$ during the inference: the target expression $e_1$ will contain types annotated with fresh size-variables and labels that uniquely identify method calls. Both $e$ and $e_1$ have the same underlying type. Furthermore, successful evaluation of $e$ (and $e_1$) requires the validity of preconditions $\Phi$, and the inclusion of the runtime tests $\Upsilon$. Successful evaluation of $e$ also changes the program state from $\Delta$ to $\Delta_1$.

For convenience, our inference rules ensure that the size variables occurring in the annotated type $\tau$ are unique; *ie.,* $FSV(\tau) \cap FSV(\Gamma) = \emptyset$ where $FSV$ returns the set of free size variables found. Some of the interesting inference rules are specified in Figure 4. In these rules, we use $s = \mathit{fresh}()$ and $\ell = \mathit{fresh}()$ to generate a new size variable and a new label, respectively. For annotated types, $\hat{\tau} = \mathit{fresh}(\mathtt{t})$ (or $\hat{\tau} = \mathit{fresh}(\tau)$) returns a new type $\hat{\tau}$ with the same underlying type as $\mathtt{t}$ (or $\tau$), but annotated with fresh size variables. The function $equate(\tau_1, \tau_2)$ generates equality constraints for the corresponding size variables of its two arguments, assuming both arguments share the same underlying type. For example, we have $equate(\mathtt{Int}^n, \mathtt{Int}^{m'}) = (n = m')$. The function $rename(\tau_1, \tau_2)$ returns a mapping instead, e.g. $rename(\mathtt{Int}^n, \mathtt{Int}^{m'}) = (n \mapsto m')$. A conditional constraint is expressed as $\zeta_1 \lhd b \rhd \zeta_2 =_{df}$ if $b$ then $\zeta_1$ else $\zeta_2$. 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 expression $\mathtt{v} = \mathtt{v} + \mathtt{u}$, with a pre-state formula $\Delta = (m'{=}2{+}n' \wedge n'{=}5)$ and $\Gamma = \{\mathtt{u} :: \mathtt{Int}^m, \mathtt{v} :: \mathtt{Int}^n, \ldots\}$. This example shows how the *prime* notation is used to capture the latest values of size variables at each symbolic state [22]. 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:

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

The function $assign$ performs the necessary sequential composition:

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

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:

$$\begin{array}{ll} \multicolumn{2}{l}{\phi_1 \circ_X \phi_2 =_{def} \exists R \cdot \rho_1(\phi_1) \wedge \rho_2(\phi_2)} \\ \text{where} & X = \{s_1, \ldots, s_n\} \text{ are size variables being updated} \\ & R = \{r_1, \ldots, r_n\} \text{ are fresh size variables} \\ & \rho_1 = \{s_i' \mapsto r_i\}_{i=1}^n \qquad \rho_2 = \{s_i \mapsto r_i\}_{i=1}^n \end{array}$$

### 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 $\mathtt{e} = \mathtt{if}\ \mathtt{u}\ \mathtt{then}\ \mathtt{v}\ \mathtt{else}\ 5$ and $\Gamma = \{\mathtt{v} :: \mathtt{Int}^n, \mathtt{u} :: \mathtt{Bool}^b\}$, the rule derives $\Delta_3$ 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 $r$.

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

### 4.3 Precondition for Safety of Check

Precondition derivation is essential for the detection of safe checks across method boundaries. A check is proved safe when a call context implies the call's preconditions. Otherwise, the preconditions associated with a call are replaced by preconditions associated with its caller. The generated preconditions are expressed in terms of the boundary variables. The [Call] rule formalizes this process.

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

$$\begin{array}{rcl} pre_1 & = & (\Delta \gtrapprox \rho(0 \leq i))\downarrow_V \equiv_s \mathit{True} \\ pre_2 & = & (\Delta \gtrapprox \rho(i < s))\downarrow_V \equiv_s (v < 7) \end{array}$$

where $\rho = \{s \mapsto m, s' \mapsto m', i \mapsto j, i' \mapsto j'\}$. The substitution $\rho$ replaces the size variables associated with the formal parameters of

$$\frac{\Gamma(v) = \tau \quad \tau_1 = \mathit{fresh}(\tau) \quad \phi = \mathit{equate}(\mathit{prime}(\tau), \tau_1)}{V;\Gamma;\Delta \vdash v \leadsto v :: \tau_1, \Delta \wedge \phi, \emptyset, \emptyset}\ [\mathbf{Var}]$$

$$\frac{V;\Gamma;\Delta \vdash e \leadsto e_1 :: \tau_1, \Delta_1, \Phi, \Upsilon \quad \Gamma(v) = \tau \quad \Delta_2 = \mathit{assign}(\Delta_1, \tau, \tau_1)}{V;\Gamma;\Delta \vdash v{=}e \leadsto v{=}e_1 :: \mathtt{Void}, \Delta_2, \Phi, \Upsilon}\ [\mathbf{Var{-}Assign}]$$

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

$$\frac{\begin{array}{c}\Gamma(v_i) = \tau_i\ \forall i \in 1..n \quad \tau = \mathit{fresh}(\hat{\tau}) \quad U = \bigcup_{i=1}^{k} \mathit{FSV}(\tau_i) \quad \ell = \mathit{fresh}() \\ (\hat{\tau}\ m(\hat{\tau}_1\ x_1, \ldots, \hat{\tau}_n\ x_n)\ \mathtt{where}\ \Delta_m;\ \Phi_m; \cdots) \in P \cup P_m \qquad \Phi_m = \{(\ell_1^+ : \phi_1), .., (\ell_k^+ : \phi_k)\} \\ \rho = \mathit{rename}(\hat{\tau}, \tau) \uplus \Sigma_{i=1}^{n}\{\mathit{rename}(\hat{\tau}_i, \tau_i)\} \qquad \mathit{pre}_i \equiv_s (\Delta \approx\!\!> \rho\phi_i){\downarrow_V} \\ \mathit{mkChk}(\mathit{pre}_i, \ell.\ell_i^+, \hat{\Phi}_i, \Upsilon_i)\ \forall i \in 1..k \qquad \hat{\Phi} = \bigcup_{i=1}^{k} \hat{\Phi}_i \quad \Upsilon = \bigcup_{i=1}^{k} \Upsilon_i\end{array}}{V;\Gamma;\Delta \vdash m(v_{1..n}) \leadsto \ell : m(v_{1..n}) :: \tau, \Delta \circ_U \rho(\Delta_m), \hat{\Phi}, \Upsilon}\ [\mathbf{Call}]$$

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

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

**Figure 4.** Type Inference Rules

sub with those from the actual parameters of the call. The new preconditions are obtained by simplifying ($\equiv_s$) the result of the operations ($\approx\!\!>$) and $\downarrow_V$. The operator $\approx\!\!>$ formulates the implication of an array-bound check by the corresponding calling context. It ensures 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:

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

The operator $\downarrow_V$ projects a constraint to the boundary variable set $V$ through quantification of (size variables from) the local variables. These variables are universally quantified, so that the resulting precondition is strengthened (weakening via $\exists$ quantifier is unsound in this case):

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

After its derivation, each precondition is classified 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 decided at a later stage (a partially-safe check). Here, $A$ is a label sequence leading to the specific bound-check, $B$ outputs the check if it is partially-safe, and $C$ outputs the label sequence identifying the check if it should be left at runtime. For the example above, we have $\mathit{mkChk}(\mathit{pre}_1, \ell.\mathtt{L}, \emptyset, \emptyset)$ and $\mathit{mkChk}(\mathit{pre}_2, \ell.\mathtt{H}, \{\ell.\mathtt{H} : \mathit{pre}_2\}, \emptyset)$, where $\ell$ is a new label associated with the call $\mathtt{sub(z, j)}$. These $\mathit{mkChk}$ clauses indicate that the low-bound check is safe, while the upper-bound check is partially safe.

For recursive methods, we first employ a fixed-point 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 next.

## 5. Recursion Analysis

Our type inference rules effectively determine both a postcondition and a set of preconditions for non-recursive methods. For recursive methods, these rules derive a (recursive) constraint abstraction that can be analyzed via fix-point analysis. The analysis steps are: (i) determine a fix-point for the constraint abstraction, and derive the method postcondition, (ii) determine an invariant for the recursive calls, and (iii) derive preconditions for checks inside recursion.

### 5.1 Deriving Postcondition

The postcondition can be derived from a recursive constraint via a fix-point approximation procedure pioneered in [11] and adapted for a disjunctive domain in [1, 18, 39, 33]. 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 $\phi_0$ is the base case. The fix-point of such an abstraction can be formalised by the following series:

$$q_0\langle n^*, r\rangle = \mathit{False}$$
$$q_{i+1}\langle n^*, r\rangle = \phi_0 \vee \phi_1[q_i\langle s^*, r_1\rangle, q_i\langle t^*, r_2\rangle]$$

For the above fix-point series to converge, we perform approximations via two techniques, known as *hulling* and *widening*.

Hulling approximates a set of disjuncts $\bigvee \phi_i$ with a conjunct $\phi$ such that $(\bigvee \phi_i) \Rightarrow \phi$. This process can be refined by hulling selectively a subset of closely-related disjuncts. We use the notion of affinity to characterize how closely related is a pair of disjuncts [33]. This selective hulling process is denoted by $\bigvee \phi_i \equiv_h \phi$.

Conjunctive widening takes a formula $\bigwedge \phi_i$ and drops (by replacing with *True*) those constraints $\phi_i$ that are changed compared to the previous step. To apply the widening operator to a disjunctive formula, we first look for pairs of disjuncts (from the current and the previous step) to widen and then apply the conjunctive widening on these pairs [33]. 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 post fix-point $q_p\langle n^*, r\rangle$ has been reached.

**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 Abstraction :**
$sumvec\langle s,i,j\rangle \equiv (i>j)\vee(i\leq j \wedge 0\leq i<s \wedge sumvec\langle s,i+1,j\rangle)$

**Figure 5.** Sum Vector Program

Consider the simple summation program from Figure 5, where the constraint abstraction obtained from our inference rules is also given. To obtain a closed-form postcondition, we apply fix-point analysis starting with *False*, the least element of the disjunctive polyhedron domain. Due to the use of widening, such fix-point approximation *always terminates*. For brevity, we display related constraints like $(j{-}1\leq i \wedge 0\leq i \wedge i\leq j)$ using the abbreviated form $(j{-}1, 0\leq i\leq j)$.

$$
\begin{aligned}
sumvec_0\langle s,i,j\rangle &= False\\
sumvec_1\langle s,i,j\rangle &= (i>j)\vee(i\leq j \wedge 0\leq i<s \wedge (\exists i_1\cdot i_1{=}i{+}1\wedge False))\\
&= (i>j)\\
sumvec_2\langle s,i,j\rangle &= (i>j)\vee(i\leq j \wedge 0\leq i<s \wedge (\exists i_1\cdot i_1{=}i{+}1\wedge i_1>j))\\
&= (i>j)\vee(0\leq i<s \wedge i{=}j)\\
sumvec_3\langle s,i,j\rangle &= (i>j)\vee(i\leq j \wedge 0\leq i<s \wedge (\exists i_1\cdot i_1{=}i{+}1\\
&\quad \wedge\ (i_1>j \vee (0\leq i_1<s \wedge i_1 = j))))\\
&= (i>j)\vee(0\leq i<s{-}1 \wedge j{=}i{+}1)\vee(0\leq i\leq s\wedge i{=}j)\\
&\equiv_h (i>j)\vee(j{-}1, 0\leq i\leq j<s)\\
sumvec_4\langle s,i,j\rangle &= (i>j)\vee(i\leq j \wedge 0\leq i<s \wedge (\exists i_1\cdot i_1{=}i{+}1\\
&\wedge\ (i_1>j \vee (0\leq i_1<s{-}1 \wedge j{=}i_1{+}1)\vee(0\leq i_1\leq s\wedge i_1{=}j)))))\\
&\equiv_h (i>j)\vee(j{-}2, 0\leq i\leq j<s)\\
&\equiv_w (i>j)\vee(0\leq i\leq j<s)\\
sumvec_5\langle s,i,j\rangle &= (i>j)\vee(i\leq j \wedge 0\leq i<s \wedge (\exists i_1\cdot i_1{=}i{+}1\\
&\quad \wedge\ (i_1>j \vee (0\leq i_1\leq j<s))))\\
&= (i>j)\vee(0\leq i\leq j<s)
\end{aligned}
$$

*Fix-Point Detected:* $sumvec_5\langle s,i,j\rangle \Rightarrow sumvec_4\langle s,i,j\rangle$

We reach the following fix-point in five iterations:

$$sumvec\langle s,i,j\rangle = (i>j)\vee(0\leq i\leq j<s)$$

### 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.

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

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

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

We derive the following recursive invariant via fix-point analysis:

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

The 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 $\ell$ at program context $s$ in a recursive method $m$ with invariant $i$. Its two preconditions can be derived as follows:

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

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

$$
\begin{aligned}
&\texttt{preFst}(\ell_1.L) = (j<i)\vee(0\leq i)\\
&\texttt{preFst}(\ell_1.\texttt{H}) = (j<i)\vee(i<s)\\
&\texttt{preRec}(\ell_1.L) = True\\
&\texttt{preRec}(\ell_1.\texttt{H}) = (j<s)\vee(s\leq j\wedge i\leq -1)\vee(s\leq j, i)
\end{aligned}
$$

These preconditions are propagated to the caller of each $sumvec$ call. Note that the precondition for (rest of the) recursive checks for $\ell_1.L$ is totally safe, but the first check of $\ell_1.L$ can be guarded by a condition $(j<i)\vee(0\leq i)$. 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 in Section 7.

## 6. 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 and quantifiers) but has double-exponential complexity, namely $2^{2^{cn}}$ where $n$ is the size of its formulae. A summary-based analysis like ours brings about a smaller number of size variables at each method boundary than a global analysis approach. With this decrease, the main proviso for efficiency is to ensure that the pre and postconditions are kept small in size.

A major reason for large formulae is the presence of disjuncts related to the specification aggregation problem observed in [25]. To counter this effect, a derived postcondition can be weakened through the *hulling* of its disjuncts. However, applying a weakening process is unsound for preconditions! For preconditions, it is only safe to strengthen and we propose a new technique that improves the analysis efficiency at a low cost in precision. We perform the strengthening of the precondition $\phi_{pre}$ using the *gist* operation from the Omega library [36].

Given a check $c$ which occurs at a location with program state $s$ and local variables $V_L$, we have earlier derived the weakest precondition using $\texttt{pre} = (\forall V_L \cdot \neg s \vee c)$. This derived precondition is unsuitable due to the negation of a (possibly very large) program state formula $s$. To derive smaller preconditions, we may simplify `pre` using a valid state $s_1$ for which $(\exists V_L \cdot s) Rightarrow s_1$ holds.

- One such $s_1$ that can be used is the *type invariant* `inv` at method entry. Let us refer to this technique of using (*gist* `pre` *given* `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* `pre` *given* $\exists V_L \cdot s$) as *strong pre-derivation*. 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 obtained by both selective and strong prederivations. 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.

In our experiments (see Section 9), we tested the three prederivation techniques. 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. We found the selective prederivation to have a reasonable compromise between efficiency and precision. Furthermore, we achieved a significant reduction in the inference times needed by some larger programs which fail to complete in reasonable (allotted) time, otherwise!

## 7. 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 $\ell_1$ | | from $\ell_2$ | |
| $\ell_1.L$ | $\ell_1.H$ | $\ell_2.L$ | $\ell_2.H$ |
| *True* | $\phi_1$ | *True* | $\phi_2$ |
| **Pre-Conditions for `p`** | | | |
| from $\ell_3$ | | from $\ell_4$ | |
| $\ell_3.\ell_1.H$ | $\ell_3.\ell_2.H$ | $\ell_4.\ell_1.H$ | $\ell_4.\ell_2.H$ |
| *True* | $\phi_3$ | $\phi_4$ | *False* |
| **Pre-Conditions for `main`** | | | |
| from $\ell_5$ | | from $\ell_6$ | |
| $\ell_5.\ell_3.\ell_2.H$ | $\ell_5.\ell_4.\ell_1.H$ | $\ell_6.\ell_1.H$ | $\ell_6.\ell_2.H$ |
| *True* | *True* | *False* | *False* |

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

```
t1 q(···) where ···  {ℓ1.H : φ1, ℓ2.H : φ2}, {}
t2 p(···) where ···  {ℓ3.ℓ2.H : φ3, ℓ4.ℓ1.H : φ4}, {ℓ4.ℓ2.H}
void main() where ···  {}, {ℓ6.ℓ1.H, ℓ6.ℓ2.H}
```

Thus, there are three unsafe checks that must be residualized at runtime, 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 using the context of the caller. 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.

We propose a *flexivariant* program specialization scheme in this paper. As special cases, we can either support polyvariant or monovariant specializations. For polyvariance, we can achieve it by never attempting to weaken any of the configurations encountered. For monovariance, we can achieve it by weakening each configuration encountered to its most conservative variant with maximal unsafe checks. For this example, the monovariant case will weaken the configurations of both `q_1` and `q_2` to `q_3`. Even though `q_3` is the weakest configuration, it still has two low bound checks eliminated.

A key feature of our 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).

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

The specialization of an expression is defined by:

$$\mathtt{P},\ \sigma,\ \varsigma \rhd^e_{\mathit{flex}}\ e\ \rightharpoonup\ e_1,\ \sigma_N$$

The specialization cache $\sigma$ drives the process, while $\varsigma$ contains the checks to be residualized. New specialization points created during specialization are stored in $\sigma_N$. We highlight the most important specialization rules below.

An array operation is specialized in $[\mathbf{Spec{-}Prim}]$ by calling the respective primitive method without array checks under the condition that the combined runtime checks for this operation, $e_1$, is true.

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

Here, a label sequence of the form $\ell.c$ occurring in the set $\varsigma$ 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 ($[\mathbf{Spec{-}Call_1}]$). 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 $[\mathbf{Spec{-}Call_2}]$.

$$\frac{\begin{array}{c} (\tau\ m(\tau_1\ x_1, \ldots \tau_k x_k)\ \mathtt{where}\ \Delta, \Phi, \Upsilon\ \{e\}) \in \mathtt{P} \\ \varsigma_2 = \mathcal{W}(m, \varsigma_1) \quad \varsigma_1 = \{\ell^+ \mid \ell_1.\ell^+ \in \varsigma\} \cup \Upsilon \\ (m, \varsigma_2, \_) \notin \sigma \qquad m_s = \mathit{genName}(m, \varsigma_2) \end{array}}{\begin{array}{c}\mathtt{P},\ \sigma,\ \varsigma \rhd^e_{\mathit{flex}}\ (\ell_1 : m(v_1, \ldots, v_k)) \\ \rightharpoonup\ m_s(v_1, \ldots, v_k),\ \{(m, \varsigma_1, m_s)\}\end{array}}\ [\mathbf{Spec{-}Call_1}]$$

$$\frac{\begin{array}{c} (\tau\ m(\tau_1\ x_1, \ldots \tau_k x_k)\ \mathtt{where}\ \Delta, \Phi, \Upsilon\{e\}) \in \mathtt{P} \\ \varsigma_1 = \{\ell^+ \mid \ell_1.\ell^+ \in \varsigma\} \cup \Upsilon \qquad (m, \mathcal{W}(m, \varsigma_1), m_s) \in \sigma \end{array}}{\mathtt{P},\ \sigma,\ \varsigma \rhd^e_{\mathit{flex}}\ (\ell_1 : m(v_1, \ldots, v_k)) \rightharpoonup\ m_s(v_1, \ldots, v_k),\ \emptyset}\ [\mathbf{Spec{-}Call_2}]$$

## 8. Array Indirections

There is a class of programs which has been largely ignored in past work on array bound checks elimination. This class of programs uses indexes that are stored in another array (*indirection array*). Array indirections are used intensively for implementing sparse matrix operations. For such matrices, only nonzero elements are stored; Additionally, the indices of these elements are kept inside an indirection array. Lujan *et al* [27] proposed a solution to handle indirection arrays via a runtime mechanism. Our system handles indirection arrays and relies entirely on compile-time analysis.

To support programs with indirection arrays, the bounds of their elements will have to be captured using an additional size variable $a$ via a new annotated type for integer array $\mathtt{Int}^a[\mathtt{Int}^s]$. Precise tracking will allow us to analyse the indexes retrieved from such integer arrays. As the array elements are being changed by the `assign` primitive, their bounds may also change during program execution. Such size properties are therefore *mutable*. To handle them safely, we require the support of an alias analysis, such as the one proposed in [21], that could be used to identify may-aliases amongst the integer arrays.

In addition to alias annotation, the main extra machinery is a set of enhanced primitive declarations (preconditions and runtime tests are unchanged, so we replace them for brevity with $\ldots$).

$$\begin{array}{l} \mathtt{Int}^a[\mathtt{Int}^r]\ \mathtt{newArr}(\mathtt{Int}^s\ \mathtt{s}, \mathtt{Int}^v\ \mathtt{v}) \\ \quad \mathtt{where}\ (0{<}s \wedge r{=}s \wedge a{=}v \wedge no\mathcal{X}\{s,v\}); \ldots \\ \mathtt{Int}^r\ \mathtt{sub}(\mathtt{Int}^a[\mathtt{Int}^s]\ \mathtt{a}, \mathtt{Int}^i\ \mathtt{i}) \\ \quad \mathtt{where}\ \ (0{\le}i{<}s \wedge r{=}a \wedge no\mathcal{X}\{i,s,a\}); \ldots \\ \mathtt{Void\ assign}(\mathtt{Int}^a[\mathtt{Int}^s]\ \mathtt{a}, \mathtt{Int}^i\ \mathtt{i}, \mathtt{Int}^v\ \mathtt{v}) \\ \quad\ \mathtt{where}\ \ (0{\le}i{<}s \wedge (a'{=}v \vee a'{=}a) \wedge no\mathcal{X}\{i,s,v\}); \ldots \end{array}$$

The array elements are updated by the `newArr` and `assign` primitives, and read by the `sub` primitive. In particular, the formula $(a'{=}v \vee a'{=}a)$ captures a weak update operation with a new approximation to the state of elements in the array. Furthermore, we may even track the relation between array indexes and their elements by using the annotated type $\mathtt{Int}^{(i,a)}[\mathtt{Int}^s]$ with a new size variable $i$ to denote index positions. By using primitives with such type declarations, we can selectively support increased precision for our analysis. Note that both the inference and the specializer work with the above indirection array primitives as well as with the array primitives without indirection from Section 3.1.

Let us illustrate how array indirections are analyzed via a simple example that initializes an array with a range of integer values:

$$\begin{array}{l} \mathtt{Void\ initArr\ (Int}^a\mathtt{[Int}^s\mathtt{]\ a,\ Int}^i\mathtt{\ i,\ Int}^j\mathtt{\ j,\ Int}^n\mathtt{\ n)} \\ \quad\ \ \mathtt{where}\ \ \mathit{initArr}\langle a,s,i,j,n\rangle \\ \{\quad \mathtt{if\ i>j\ then\ ()\ else\ \{a[i]=n;\ initArr(a,i+1,j,n+1)\}\ \}} \end{array}$$

Using the fix-point analysis described in Sec 5, we can obtain the following post-condition which captures the initialization of the array elements:

$$\begin{array}{l} \mathit{initArr}\langle a,s,i,j,n\rangle\ \equiv \\ \quad \equiv (i{>}j \wedge a'{=}a)\ \ \vee\ \ (0{\le}i{\le}j{<}s \wedge (a'{=}a \vee n{\le}a'{\le}n{+}j{-}i)) \end{array}$$

## 9. Implementation

We have constructed the proposed modular inference system together with a program specializer. Our implementation includes a pre-processing phase to convert a C-like input program to IMP. The output from our system was validated by a separate checking system that we have also built. The entire prototype system was written in Haskell and compiled using Glasgow Haskell compiler[32]. For constraint solving in the Presburger arithmetic domain, we used the Omega library [35]. A web-demo of our system can be found at http://loris-7.ddns.comp.nus.edu.sg/~popeeaco/imp/.

We evaluated our prototype using small programs with challenging recursion and two numerical-intensive benchmarks: SciMark (`Fast Fourier Transform, LU decomposition, Successive Over-Relaxation`) [31] and Linpack [13]. Our test platform was a

| Benchmark Programs | Source (lines) | Static Checks | Checking (secs) | Inference (secs) | | | Static Checks Eliminated |
|---|---|---|---|---|---|---|---|
| | | | | Weak | Selective | Strong | |
| binary search | 31 | 2 | 0.17 | 1.84 | 1.81 | 1.79 | 100% |
| bubble sort | 39 | 12 | 0.43 | 1.55 | 1.51 | 1.47 | 100% |
| foo | 12 | 4 | 0.39 | 0.66 | 0.67 | 0.87 | 50%/75% |
| hanoi tower | 38 | 16 | 3.73 | 11.74 | 11.53 | 11.47 | 100% |
| merge sort | 58 | 24 | 7.70 | 11.21 | 16.01 | 13.07 | 100% |
| queens | 39 | 8 | 0.52 | 2.13 | 2.11 | 2.10 | 100% |
| quick sort | 43 | 20 | 0.38 | 1.92 | 1.92 | 1.76 | 100% |
| sentinel | 26 | 4 | 0.05 | 0.18 | 0.16 | 0.15 | 75% |
| sparse multiply | 46 | 12 | 3.27 | 22.61 | 17.37 | 7.09 | 100% |
| sumvec | 33 | 2 | 0.11 | 0.51 | 0.48 | 0.47 | 100% |
| FFT | 336 | 62 | 9.58 | * | 58.02 | 28.74 | 100% |
| LU Decomp. | 191 | 82 | 13.10 | 137.1 | 93.31 | 72.91 | 100% |
| SOR | 84 | 32 | 1.15 | 7.18 | 4.67 | 3.8 | 100% |
| Linpack | 903 | 166 | 42.26 | * | 360.1 | 162.2 | 100% |

**Figure 6.** Statistics for Array Bound Checks Elimination

Pentium 2.8 GHz system with 1GBytes main memory, running Red Hat Linux 9.0.

Our main objective was to show the viability and the precision of the system. Figure 6 summarises the statistics obtained for each program that we inferred. 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 each program (column 3). The time taken for inference (columns 5-7) includes parsing, preprocessing, modular type inference and specialization. For comparison, we present the time taken for checking pre-annotated programs (column 4), composed from parsing and dependent type checking. The size of the method constraints (preconditions, postconditions and recursive invariants) is on average around 15% of the size of the source program. Thus, our inference eliminates the effort to annotate methods required of programmers with access to only a dependent type checker.

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` (column 8). The `sentinel` example illustrates a pattern where some checks cannot be eliminated by our method, since it makes use of a sentinel/guard against falling off one end of the array. Like [45, 43], we were unable to capture the existential property that is required for check elimination. For the `foo` example, strong prederivation and selective prederivation eliminate 50% and 75%, respectively, of the static checks.

We can compare our experimental results to other analyses that are based on disjunctive domains similar to ours, but employ only forward derivation [39, 33]. For the benchmark set used in our previous work [33], a forward derivation and a fixed-point analysis with Hausdorff affinity akin to [39] led to 76% check elimination, while a forward analysis using planar affinity introduced in [33] was able to eliminate 84% of the checks. Compared to these two previous analyses, our current techniques achieve 100% check elimination. We can attribute this improvement to the combination of the forward derivation of postconditions with the backward derivation of preconditions. Another reason for our improved results was the handling of array indirections present in the `sparse multiply` and `Linpack` benchmarks.

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 those larger programs we found it crucial to use either selective or strong prederivation; weak prederivation does not scale up as inference fails to complete in reasonable time (cases denoted by * signify over an hour inference time).

To summarize our experiences, we observe that our initial goal was to build a precise inference system and make it practical by employing a modular analysis that computes method summaries. However, the small number of size variables at each method boundary was not enough to ensure the efficiency of our system. The backward component of our system proved to be expensive mostly due to two reasons. Firstly, precondition derivation was done via negation of a (possibly very large) program state formula. Secondly, array bound checks were specialized by deriving individual preconditions, one for each check. This was our intention in order to enable aggressive program optimization. Note that proving program safety does not necessarily require individual precondition derivation (and, in our setting, can be less expensive). To cope with these additional difficulties, we employed additional approximations to reduce the size of method summaries: weakening of postconditions via selective hulling and strengthening of preconditions via gisting. With these techniques, both the inference and the specializer were integrated into a system that was shown to be practical and precise enough for our purposes.

## 10. 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 fixed-point 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 system below and refer the reader to the technical report [34] for details on the proof.

THEOREM 1 (Soundness). *Let $P$ be a program and a type inference judgement such that ($P_m \vdash_I P \leadsto P_I$). 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.

## 11. Related Works

Traditionally, data-flow analysis techniques have been employed to gather information for the purpose of identifying redundant array checks [19]. 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 [24], to the use of difference constraints in [3].

To identify redundant checks more accurately, verification-based methods have been used by Suzuki and Ishihata [40], Necula and Lee [29] and Xu *et al* [45]. Xi and Pfenning have advocated the use of dependent types for array bound check elimination [43]. 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* [16, 17]. 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 deals with inter-procedural analysis over recursive methods, whereas they focus on intra-procedural analysis and loops. Recently, Flanagan [15] introduced the idea of inserting assertions that cannot be proven during type checking as run-time checks. Our use of a flexivariant specializer to insert runtime checks (after inference) shares a similar flavour. However, our proposal is based on inference, while his is formalised for a type-checker.

Identifying redundant array bound checks can also be done using abstract interpretation techniques over numerical domains. In a seminal paper, Cousot and Halbwachs [11] introduced the polyhedra abstract domain and defined convex-hull and widening operators for this domain. Subsequently, various other abstract domains have been proposed, varying from conjunctive domains like octagons [28], pentagons [26] or symbolic ranges [38] to disjunctive domains [39, 33]. In fact, safety analyzers that scale to large critical programs like ASTRÉE [2] or C Global Surveyor [41] use elaborate combinations of abstract domains to achieve maximum efficiency. For example, the static analyzer that has been described by Cousot *et al* [2, 10] 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. However, 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 [9] and our proposal is a solution in this direction.

To avoid fix-point iteration, Rugina and Rinard [37] proposed an analysis method (using linear programming) to synthesize polynomial symbolic bounds. While efficient, fixing a target form (without disjunction) for the symbolic bound may result in loss of precision. Dor *et al* advocated for linear constraints, expressed using pre/post conditions, to help determine the safety of C pointers to string buffers [14]. For their experiments, the inference result is, however, less precise than user-supplied annotations. This is likely due to the absence of disjunction and path-sensitivity during inference.

The idea of deriving preconditions for partially redundant checks was first proposed in [8] to complement postcondition inference on sized types [7] for a first-order functional language. However, this early work was mostly informal and had no implementation. We formalize this early idea by inferring a sound dependent-type annotation for an imperative language, and integrating its results with a program specializer. Moreover, we now have a practical and precise implementation.

Unlike the work in [6] which uses a separate set-based analysis for properties of elements in a collection, the current paper uses arithmetic constraints to represent such properties directly for indirection arrays. This decision reduces the burden of using two different analyses. On the other hand, the set-based analysis approach [6] may give more precise results via universal and existential properties, and deal with elements which may not be integers.

Flexivariant specialization scheme enables a trade-off to be made, that can give up some array check optimization for a reduction in code size. Such trade-off can be guided with the help of suitable path-profiling techniques[42]. Such a compromise was originally pioneered in a technique, called *selective specialization* [12], to convert expensive dynamic method dispatches for OO programs into static counterparts, where possible. Our flexivariant scheme supports the proposed inference with a family of specializers, with selective specialization as a possible option.

## 12. Conclusion

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.

## Acknowledgments

This work was supported by NUS grant R252-000-213-112 and A*STAR grant R-252-000-233-305. It was also supported in part by Microsoft Research through its Ph.D. Scholarship program for the second author. We thank Siau-Cheng Khoo for his profound and sound advices. We also thank anonymous referees for their careful comments.

## References

[1] R. Bagnara, P.M. Hill, and E. Zaffanella. Widening operators for powerset domains. In *Verification, Model Checking and Abstract Interpretation*, pages 135–148, 2004.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM Conference on Programming Language Design and Implementation*, pages 196–207, 2003.

[3] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *ACM Conference on Programming Language Design and Implementation*, pages 321–333, 2000.

[4] R. Bodik, R. Gupta, and M.L. Soffa. Complete removal of redundant expressions. In *ACM Conference on Programming Language Design and Implementation*, pages 1–14, June 1998.

[5] R. Chatterjee, B. Ryder, and W. Landi. Relevant context inference. In *ACM Symposium on Principles of Programming Languages*, 1999.

[6] W. N. Chin, S. C. Khoo, and D. N. Xu. Extending sized type with collection analysis. In *Proceedings of the ACM SIGPLAN Workshop on Partial evaluation and semantics-based program manipulation*, pages 75–84. ACM Press, 2003.

[7] W.N. Chin and S.C. Khoo. Calculating sized types. In *ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation*, pages 62–72, Boston, Massachusetts, January 2000.

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

[9] P. Cousot and R. Cousot. Modular static program analysis. In *International Conference on Compiler Construction*, 2002.

[10] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *European Symposium on Programming*, pages 21–30, 2005.

[11] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[12] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *ACM Conference on Programming Language Design and Implementation*, pages 93–102, 1995.

[13] 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.

[14] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *ACM Conference on Programming Language Design and Implementation*, pages 155–167, 2003.

[15] C. Flanagan. Hybrid type checking. In *ACM Symposium on Principles of Programming Languages*, pages 245–256, 2006.

[16] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *ACM Symposium on Principles of Programming Languages*, 2002.

[17] C. Flanagan and J.B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *ACM Symposium on Principles of Programming Languages*, 2001.

[18] B.S. Gulavani and S.K. Rajamani. Counterexample driven refinement for abstract interpretation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2006.

[19] R. Gupta. A fresh look at optimizing array bound checking. In *ACM Conference on Programming Language Design and Implementation*, pages 272–282, New York, June 1990.

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

[21] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *ACM Conference on Programming Language Design and Implementation*, 2001.

[22] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[23] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *ACM Symposium on Principles of Programming Languages*, pages 410–423. ACM Press, January 1996.

[24] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. In *ACM Conference on Programming Language Design and Implementation*, pages 270–278. ACM Press, June 1995.

[25] P. Lam, V. Kuncak, and M. Rinard. Cross-cutting techniques in program specification and analysis. In *International Conference on Aspect-Oriented Software Development*, March 2005.

[26] F. Logozzo and M. Fahndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In *ACM Symposium on Applied Computing*, 2008.

[27] Mikel Luján, John R. Gurd, T. L. Freeman, and José Miguel. Elimination of Java array bounds checks in the presence of indirection. In *ACM Joint Java Grande-IScope Conf.*, pages 76–85, 2002.

[28] A. Mine. The octagon abstract domain. In *the Eighth Working Conference on Reverse Engineering*, 2001.

[29] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *ACM Conference on Programming Language Design and Implementation*, pages 333–344, 1998.

[30] T.V.N. Nguyen and F. Irigoin. Efficient and effective array bound checking. *ACM Trans. Program. Lang. Syst.*, 27(3):527–570, 2005.

[31] National Institute of Standards and Technology. Java SciMark benchmark for scientific computing. http://math.nist.gov/scimark2.

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

[33] C. Popeea and W.N. Chin. Inferring disjunctive postconditions. In *ASIAN CS Conference*, 2006.

[34] C. Popeea, D.N. Xu, and W.N. Chin. A practical and precise inference and specializer for array bound checks elimination. Technical report. http://www.comp.nus.edu.sg/~corneliu/research/array.tr.pdf.

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

[36] W. Pugh. Counting solutions to Presburger formulas: how and why. In *ACM Conference on Programming Language Design and Implementation*, 1994.

[37] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM Conference on Programming Language Design and Implementation*, pages 182–195. ACM Press, June 2000.

[38] S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program analysis using symbolic ranges. In *International Static Analysis Symposium*, pages 366–383, 2007.

[39] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *International Static Analysis Symposium*, Springer LNCS, Seoul, Korea, August 2006.

[40] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *ACM Symposium on Principles of Programming Languages*, pages 132–143, 1977.

[41] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *ACM Conference on Programming Language Design and Implementation*, pages 231–242, 2004.

[42] Y. Wu and J.R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 1–11. ACM Press, 1994.

[43] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM Conference on Programming Language Design and Implementation*. ACM Press, June 1998.

[44] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *ACM Symposium on Principles of Programming Languages*, pages 351–363, 2005.

[45] Z. Xu, B.P. Miller, and T. Reps. Safety checking of machine code. In *ACM Conference on Programming Language Design and Implementation*, pages 70–82. ACM Press, June 2000.