A Modular Type Inference and Specializer for Array Bound Checks Elimination

Dana N. Xu\textsuperscript{1}, Corneliu Popeea\textsuperscript{2}, Siau-Cheng Khoo\textsuperscript{2}, and Wei-Ngan Chin\textsuperscript{2}\textsuperscript{2}

\textsuperscript{1}Computer Laboratory, University of Cambridge
\textsuperscript{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 formulæ. 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 pre-condition 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 formulæ 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 formulæ 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 \( \text{pre} \) (expressed in terms of variables from only \( V \)) such that the following is valid.
  \[
  (\text{pre} \land s \Rightarrow c)
  \]
  Such a precondition can be generated using
  \[
  \text{pre} = (\forall L \cdot _{-j}^n \text{pre}(c)) \text{ where } L \text{ is the set of local variables, denoted by } \text{vars}(s,c) - V.
  \]
- \( c \) 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 \( \text{randInt} \) returns a random integer, while \( \text{abs} \) 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 \( \text{len}(a) - 1 \). This method contains two array accesses at locations \( \ell_1 \) and \( \ell_2 \). 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 \( \ell_0 \), are:

\[
\begin{align*}
\text{sps}(\ell_0) &= \text{len}(a) > 0 \\
\text{sps}(\ell_1) &= \text{sps}(\ell_0) \land i = j + 1 \land (0 < i < n) \\
\text{sps}(\ell_2) &= \text{sps}(\ell_0) \land i = j + 1 \land m > 0
\end{align*}
\]

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

\[
\begin{align*}
\text{sps}(\ell_1) &= (i > 0) \text{ and } \text{sps}(\ell_2) = (m > 0)
\end{align*}
\]

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:

\[
\begin{align*}
\text{pre}(\ell_1.H) &= \forall i. \text{sps}(\ell_1) \Rightarrow i < \text{len}(a) \\
&= \forall i. (\text{len}(a) > 0 \land i = j + 1 \land (0 < i < n) \Rightarrow i < \text{len}(a)) \\
&= \text{len}(a) < 0 \lor (j < \text{len}(a) - 2 \land 1 < \text{len}(a)) \\
&\lor (1 < \text{len}(a) - 2 < j + 1 \land n = j)
\end{align*}
\]

\[
\begin{align*}
\text{pre}(\ell_2.H) &= \forall i. m. \text{sps}(\ell_2) \Rightarrow m < \text{len}(a) \\
&= \forall i. m. (\text{len}(a) > 0 \land i = j + 1 \land m > 0 \Rightarrow m < \text{len}(a)) \\
&= \text{len}(a) < 0
\end{align*}
\]

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, \( \text{len}(a) > 0 \) is a type invariant and can be used to simplify the precondition of \( \text{pre}(\ell_2.H) \) to false. Similarly, \( \text{pre}(\ell_1.H) \) can be simplified to \( j < \text{len}(a) - 2 \lor n = j \) using the same type invariant. This simplified formula contains a condition \( j < \text{len}(a) - 2 \) 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 \( \text{len}(a) \), is positive (a design decision we took for our language).

We perform each simplification of a formula \( \phi_1 \) under type invariant \( \phi_2 \) by the operation \( \text{gist } \phi_2 \text{ given } \phi_2 \). This yields a simplified term \( \phi_2 \) such that \( \phi_1 \land \phi_2 \equiv \phi_1 \land \phi_2 \) [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 \( \Delta \), a set of preconditions \( \Phi \), a set of unsafe checks \( \Upsilon \), and size-annotated types \( \tau_0, ..., \tau_k \).

\[
\begin{align*}
\tau_m & (\tau_1 v_1, ..., \tau_k v_k) \text{ where } \Delta : \Phi; \Upsilon \{\text{body}\} \\
\end{align*}
\]

For example, after type inference on the \( \text{foo} \) 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 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 \( \Phi_{\text{pre}} \) as part of its method header. The precondition \( \Phi_{\text{pre}} \) is a conjunction of checks from \( \Phi \) that are guaranteed safe from each call site, as outlined below.

\[
\begin{align*}
\tau_m & (\tau_1 v_1, ..., \tau_k v_k) \text{ where } \Delta : \Phi_{\text{pre}} \{\text{body}\} \\
\end{align*}
\]

For example, if \( \text{pre}(\ell_1.H) \) is found to be safe when analyzing the call sites of method \( \text{foo} \), we can generate the specialised (and well-typed) method at the bottom of Figure 1.

![Figure 1: Infer and Specialize: An Example](image-url)

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 \( \ell_1.L \) and \( \ell_2.L \),

---

\[\text{Float} \text{ foo}(\text{Float}[\text{Int}] \ a, \text{Int} \ j, \text{Int} \ n) \]

\[
\begin{align*}
\ell_0: & \text{ Float } v = 0.0; \text{ Int } i = j + 1; \\
& \text{ if } (0 < i < n) \text{ then } v = \text{foo}(\ell_1: \text{a}[i]) \text{ else } (); \\
& \text{Int } m = \text{abs}(\text{randInt}()); \\
& v = (\ell_2: \text{a}[m]) \}
\end{align*}
\]

\[\downarrow\]

**Inference**

\[\text{Float} \text{ foo}(\text{Float}[\text{Int}] \ a, \text{Int}^1 \ j, \text{Int}^n \ n) \text{ where } (n < j) \lor (j < s - 2 \land j < n - 1); \]

\[
\{\ell_1.H : j < s - 2\}; \{\ell_2.H \} \}
\]

\[\downarrow\]

**Specialization**

\[\text{Float} \text{ foo}(\text{Float}[\text{Int}] \ a, \text{Int}^1 \ j, \text{Int}^n \ n) \text{ where } (n < j) \lor (j < s - 2 \land j < n - 1); \]

\[
\ell_0: \text{ Float } v = 0.0; \text{ Int } i = j + 1; \\
& \text{ if } (0 < i < n) \text{ then } v = \text{foo}(\ell_1: \text{a}[i]) \text{ else } (); \\
& \text{Int } m = \text{abs}(\text{randInt}()); \\
& v = (\text{if } m < \text{len}(a) \text{ then } \ell_2: \text{a}[m] \text{ else error } ) \}
\]
or the precondition of each method (for array check $t_1, \ldots , t)$ or by inserted runtime test (for array check $t_2, \ldots , t$). 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 $\text{Bool} \, \tau$ with $\tau \equiv 0 \, \text{false} \, 1 \, \text{true}$ and $\tau \equiv 1 \, \text{false} \, 2 \, \text{true}$ can denote an array of floats with $s$ elements. Input-output relation between size variables from method parameters and result is capture after the run-time test.

3.1 Core Language

To formalise our type inference we first introduce a source language $\text{Imp}$ (see Fig 2), where types, denoted by $\tau$ and $\xi$, do not have annotations. $\text{Imp}$ has support for assignments, conditions, 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 $\text{Imp}$. 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. $\text{newArr}$, $\text{len}$, $\text{sub}$, $\text{assign}$).

In Figure 3 we introduce a language with dependent type, $\text{Imp}_D$, which is designed to capture the result of our inference. Each method declaration captures three information:

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 $\text{Bool} \, \tau$ with $\tau \equiv 0 \, \text{false} \, 1 \, \text{true}$ and $\tau \equiv 1 \, \text{false} \, 2 \, \text{true}$ can denote an array of floats with $s$ elements. Input-output relation between size variables from method parameters and result is capture after the run-time test.

3.1 Core Language

To formalise our type inference we first introduce a source language $\text{Imp}$ (see Fig 2), where types, denoted by $\tau$ and $\xi$, do not have annotations. $\text{Imp}$ has support for assignments, conditions, 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 $\text{Imp}$. 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. $\text{newArr}$, $\text{len}$, $\text{sub}$, $\text{assign}$).

In Figure 3 we introduce a language with dependent type, $\text{Imp}_D$, which is designed to capture the result of our inference. Each method declaration captures three information:

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 $\text{Bool} \, \tau$ with $\tau \equiv 0 \, \text{false} \, 1 \, \text{true}$ and $\tau \equiv 1 \, \text{false} \, 2 \, \text{true}$ can denote an array of floats with $s$ elements. Input-output relation between size variables from method parameters and result is capture after the run-time test.

3.1 Core Language

To formalise our type inference we first introduce a source language $\text{Imp}$ (see Fig 2), where types, denoted by $\tau$ and $\xi$, do not have annotations. $\text{Imp}$ has support for assignments, conditions, 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 $\text{Imp}$. 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. $\text{newArr}$, $\text{len}$, $\text{sub}$, $\text{assign}$).

In Figure 3 we introduce a language with dependent type, $\text{Imp}_D$, which is designed to capture the result of our inference. Each method declaration captures three information:
Primitive \texttt{newarr} returns a new array with all elements initialized to value \texttt{v}, \texttt{len} returns the length at dimension \texttt{1}, \texttt{sub} returns an array element from the specified location, while \texttt{assign} updates the specified location with value \texttt{v}. The extension of these primitives to \(k\)-dimensional case is straightforward as shown in the technical report [29].

4. Inference Rules for \(\text{IMP}_I\)

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 \(P_0 \vdash_I P \leadsto P_I\). 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:

\[ V; \Gamma; \Delta \vdash e \sim e_1 :: \tau, \Delta_1, \Phi, \Upsilon \]

Here \(V\) is a set of size variables available at the boundary of the method \((\text{called boundary variables})\) in which the expression \(e\) resides. \(\Gamma\) is a type environment mapping program variables to their annotated types; \(\Delta\) (resp. \(\Delta_1\)) is a size constraint over size variables occurring in \(\Gamma\) (resp. \(\Gamma\) and \(\tau\)) before (resp. after) the evaluation of the expression \(e\); \(\tau\) is an annotated type.

The above judgement states that \(e\) will be transformed into \(e_1\) with an annotated type \(\tau\) 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 \(\Phi\), and the inclusion of the runtime tests \(\Upsilon\). Successful evaluation of \(e\) also changes the program state from \(\Delta\) to \(\Delta_1\). 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 \(\tau\) are unique; i.e., \(\text{FSV}(\tau) \cap \text{FSV}(\Gamma) = \emptyset\) where \(\text{FSV}\) 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 \(s = \text{fresh}(i)\) and \(t = \text{fresh}(\tau)\) to generate a new size variable and a new label, respectively. We also extend it to annotated type, so that \(\tau = \text{fresh}(\tau)\) or \(\tau = \text{fresh}(\tau)\) returns a new type \(\tau\) with the same underlying type as \(\tau\) (or \(\tau\)), but annotated with fresh size variables. The function \(\text{equate}(\tau_1, \tau_2)\) generates equality constraints for the corresponding size variables of its two arguments, assuming both arguments share the same underlying type. For example, we have \(\text{equate}([\text{Int}^n, \text{Int}^m]) = (n = m)\). The function \(\text{rename}(\tau_1, \tau_2)\) returns a mapping instead, e.g. \(\text{rename}([\text{Int}^n, \text{Int}^m]) = (n \rightarrow m)\). Also, the conditional is expressed as \(\zeta_1 < b \Rightarrow \zeta_2 = q\) 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 \(v = v + u\), with a pre-state \(\Delta = (m = 2+n'/m+n' = 5)\) and \(\Gamma = \{u :: \text{Int}^m, v :: \text{Int}^n, \ldots\}\). This example shows how the \textit{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, \(\bigtriangleup\), where \(X\) denotes a set of size variables that are being updated. The following depicts the inference step for assignment:

\[
\begin{align*}
\Gamma(v) &= \text{Int}^n \\
\Gamma(u) &= \text{Int}^m \\
V; \Gamma; \Delta \vdash v + u \sim v + u :: \text{Int}^n, \Delta \wedge n' = m', \emptyset, \emptyset \\
\Delta_2 &= \text{assign}(\Delta \wedge r = n' + m', \text{Int}^n, \text{Int}^m) \\
\Big&\; \vdash v + u \sim v + u :: \text{void}, \Delta_2, \emptyset, \emptyset
\end{align*}
\]

The function \(\text{assign}\) performs the necessary sequential composition:

\[
\text{assign}(\Delta, \tau_1, \tau_2) = \text{id}_{\Delta} \quad \text{let } X = \text{FSV}(\tau); Y = \text{FSV}(\tau_2) \quad \text{in } \exists Y(V; \Delta \not\vdash \text{equate}(\text{prime}(\tau), \tau_1))
\]

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

\[
\begin{align*}
\Delta_2 &= \exists r \cdot (\exists (\text{Int}^n \wedge n' = m') \cdot (n \sim r)) \\
&= \exists r \cdot (\exists (\text{Int}^n \wedge n' = m') \cdot (n \sim r)) \\
&= \exists r \cdot (\exists n_0 \cdot m_2 = 2 + n_0 \wedge n_0 = 5 \wedge r = n_0 + n_0 / n' = r) \\
&= (m_2 = 7 \wedge n = n' = 5)
\end{align*}
\]

More formally, sequential composition is defined as:

\[
\phi_1 \circ \phi_2 = \text{id}_{\Delta} \quad \exists R \cdot (\rho_1(\phi_1) \wedge \rho_2(\phi_2))
\]

where \(X = \{s_1, \ldots, s_n\}\) are size variables being updated

\[
R = \{r_1, \ldots, r_n\}\}
\]

are fresh size variables

\[
\rho_1 = \{s_1 \mapsto r_1, n_1\}
\]

\[
\rho_2 = \{s_2 \mapsto r_2, n_2\}
\]

4.2 Path Sensitive Inference

The \(\text{tr}\) rule attempts to track the size constraint of conditions with path sensitivity. The two conditional branches are distinguished by assuming the conditional-test result to be either \(0\) or \(1\), representing the \texttt{true} or \texttt{false} value, respectively. Given \(e = \text{if } u \text{ then } v \text{ else } 5\) and \(\Gamma = \{v :: \text{Int}^n, u :: \text{Bool}\}\), we have the following derivation, where \(\Delta_3\) 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 \(r\).

\[
\begin{align*}
\Delta_1 &= \Delta \wedge (b' = 1) \\
V; \Gamma; \Delta \vdash v \sim v :: \text{Int}^n, \Delta_1 \wedge r_1 = n', \emptyset, \emptyset \\
V; \Gamma; \Delta_2 \vdash v + 5 :: \text{Int}^n, \Delta_2 \wedge r_2 = n', \emptyset, \emptyset \\
\Delta_3 &= \Delta \wedge (b' = 1 \wedge n = n') \vee (b' = 0 \wedge r = 5) \\
V; \Gamma; \Delta \vdash e \sim e :: \text{Int}^n, \Delta_3, \emptyset, \emptyset
\end{align*}
\]

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 \(\text{call}\) rule. The generated preconditions are expressed in terms of the boundary variables.

As an example, consider inferring a primitive call \texttt{sub}(z, i) under the type assumption \(\Gamma = \{v :: \text{Int}^n, z :: \text{Float}([\text{Int}^m]), i :: \text{Int}\})\) and pre-state \(\Delta = (m = 11) \wedge (n' = 10) \wedge (i' = 2) \wedge (v' = 1) \wedge (v' > 5)\). Furthermore, let the set of boundary variables \(V = \{v, m\}\) (thus \(i\) is a local). The two array-bound checks of \texttt{sub} (declared in Section 3.1), \(0 \leq s < n\) and \(s < n\), are transformed into the following preconditions:

\[
\begin{align*}
\text{pre}_1 &= (s \geq 0 \\
\text{pre}_2 &= (s < n) \wedge (s < n) \wedge (s < n)
\end{align*}
\]

where \(\rho = \{n \mapsto m, n' \mapsto m', s \mapsto i', s' \mapsto i'\}\). The substitution \(\rho\) replaces caller’s size variables for those associated with the formal parameters of \texttt{sub}. The new preconditions are obtained by simplifying \((\equiv_1)\) the result of the operations \(\bowtie\) and \(\gg\). \(\bowtie\) 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:
\[ \Delta \equiv \phi \equiv_{df} (\Delta \Rightarrow \rho(\phi)) \] where \( \phi = \{ s_1 \mapsto s'_1, \ldots, s_m \mapsto s'_m \}; \{ v_1, \ldots, v_k \} = FSV(\phi) \)
The quantification operator \( \forall V \) projects a constraint to the boundary variable set \( V \) through universal quantification of the local variables:
\[ \phi | V \equiv_{dfj} \forall W. \phi \quad \text{where} \quad W = FSV(\phi) \setminus V. \]
Each derived precondition is then used by the relation
\[ \text{mkChk}(\text{pre}, A, B, C) \]
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 \( \text{mkChk}(\text{pre}_2, \ell \cdot \{ \ell : \text{pre}_2 \}, \emptyset) \) and \( \text{mkChk}(\text{pre}_2, \ell \cdot \{ \ell : \text{pre}_2 \}, \emptyset) \), where \( \ell \) is a new label associated with the call \( \text{sub}(\cdot, 1) \). The above \( \text{mkChk} \) 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 precondition 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 postcondition 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 \( q(n^*, r) \) where \( n^* \) denotes 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.

\[ q(n^*, r) = \phi_0 \lor q(s^*, r_1), q(t^*, r_2) \]

Note that \( \phi_1[i, \ldots, i] \) is a formula with two holes containing the two recursive invocations, while \( \phi_0 \) 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.

\[ q_0(n^*, r) = \text{false} \quad q_{i+1}(n^*, r) = \phi_0 \lor q_i(s^*, r_1), q_i(t^*, r_2) \]

For the above fix-point series to converge, we perform approximations via two techniques, known as tailing and widening from [7]. Hulling combines a set of disjuncts \( \forall \phi_i \)
Methods with Postconditions:

```c
Float sumvec(Float[Int] a, Int i, Int j)
where sumvec(s,i,j)=...;
{ if i>j then 0.0 else {Int v = f1(sub(a,i));
                   v+f2:sumvec(a,i+1,j) } }
Float sum(Float[Int] a) where sum(s,...);
{ Int l=e3:llen(a); l:f4:sumvec(a,0,1) }
```

Figure 5: Sum Vector Program

into a conjunct \( \phi \) such that \( \forall \phi_i \Rightarrow \phi \). We refine this process by allowing hulling to be selectively applied to a subset of disjuncts. We denote this process by \( \forall \phi_i \equiv \forall \phi \). Widening takes a formula \( \forall \phi \land \exists \phi \) and then drops (by replacing with \( \text{true} \)) one or more \( \phi \) from it. Let us denote widening by \( \equiv_w \). We shall apply each fix-point approximation until we obtain a formula \( q_n(n',r) \) such that \( q_{n+1}(n',r) = q_n(n',r) \). This test indicates that a fix-point \( q_{n}(n',r) \) 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 \( \text{sumvec} \). We reach the following fix-point in four iterations:

\[
\text{sumvec}(s,i,j) = (i>j)\lor(0 \leq i \leq j, s-1)
\]

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

\[
\begin{align*}
\text{sumvec}_0(s,i,j) & = \text{false} \\
\text{sumvec}_1(s,i,j) & = (i>j)\lor((0 \leq i < s)\land(\exists i_1:i_1=i+1\lor\text{false})) \\
\text{sumvec}_2(s,i,j) & = (i>j)\lor((0 \leq i < s)\land(\exists i_1:i_1=i+1\land i_1 > j)) \\
\text{sumvec}_3(s,i,j) & = (i>j)\lor((0 \leq i < s)\land(\exists i_1:i_1=i+1) \\
& \quad \land((i_1 > j)\lor(0 \leq i \leq s-2)\lor(0 \leq s \land j = j)) \\
& \equiv (i > j)\lor(0 \leq i \leq j, s-1) \\
\text{sumvec}_4(s,i,j) & = (i > j)\lor((0 \leq i < s)\land(\exists i_1:i_1=i+1) \\
& \quad \land((i_1 > j)\lor(0 \leq i_1 \leq j, s-1))) \\
& \equiv (i > j)\lor(0 \leq i \leq j, s-1)
\end{align*}
\]

**Fix-Point Detected.** \( \text{sumvec}_4(s,i,j) \Rightarrow \text{sumvec}_3(s,i,j) \)

### 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 a recursive invocation, \( q(n', r) = q_0 \lor q_1 [q(s', r_1), q(t', r_2)] \). We can obtain a one-step relation, named \( I \), that attempts to relate the input \( n^* \) with that of its recursive call, \( n^* \), as shown below.

\[
I(n^*, n^*) = q_0 [\bigwedge (s = n^*), q(t', r_2)] \lor q_0 [q(s', r_1), \bigwedge (t = n^*)]
\]

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

\[
I_{i+1}(n^*, n^*) = I_i(n^*, n^*) \lor (\exists z : I_i(n^*, z) \land I_i(z, n^*))
\]

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

\[
\text{sumvec}(s,i,j) = (k=s)\land(j=j)\lor(0 \leq i < k \leq j)
\]

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:

\[
\begin{align*}
\text{preFst}(\ell) & = \forall L \cdot (s \Rightarrow c) \land L = \text{vars}(s, c) - V \\
\text{preRec}(\ell) & = \forall L \cdot (s \land c) \land L = \text{vars}(s, c) - V
\end{align*}
\]

For the \( \text{sumvec} \) example, we would derive two sets of preconditions, namely:

\[
\begin{align*}
\text{preFst}(f_1, L) & = (j < i) \lor (0 \leq i) \\
\text{preFst}(f_1, H) & = (j < i) \lor (i < s) \\
\text{preRec}(f_1, H) & = (i < s) \lor (s < j) \lor (s < j, i)
\end{align*}
\]

These preconditions are propagated to the caller of each \( \text{sumvec} \) call. Note that the precondition for (rest of the) recursive checks for \( f_1 \) is totally safe, but the first check of \( f_1 \) can be guarded by a condition \( (j < i) \lor (0 \leq i) \). 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 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:

```c
void main() { ... t2 p(...); ... l5 := p(...); ... l6 := q(...); ... v1=(l5: sub(a1,11)); ... l5:=assign(a2,12,v1)}
```
Let us assume that the results of inference are as follows:

<table>
<thead>
<tr>
<th>Pre-Conditions for q</th>
<th>from ( \ell_1 )</th>
<th>from ( \ell_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ell_1, L )</td>
<td>( \phi_1 )</td>
<td>( \phi_2 )</td>
</tr>
<tr>
<td>( \ell_1, H )</td>
<td>( \phi_1 )</td>
<td>( \phi_2 )</td>
</tr>
<tr>
<td>( \ell_2, L )</td>
<td>( \phi_1 )</td>
<td>( \phi_2 )</td>
</tr>
<tr>
<td>( \ell_2, H )</td>
<td>( \phi_1 )</td>
<td>( \phi_2 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pre-Conditions for p</th>
<th>from ( \ell_3 )</th>
<th>from ( \ell_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ell_3, \ell_1, H )</td>
<td>( \phi_1 )</td>
<td>( \phi_2 )</td>
</tr>
<tr>
<td>( \ell_3, \ell_2, H )</td>
<td>( \phi_1 )</td>
<td>( \phi_2 )</td>
</tr>
<tr>
<td>( \ell_4, \ell_1, H )</td>
<td>( \phi_1 )</td>
<td>( \phi_2 )</td>
</tr>
<tr>
<td>( \ell_4, \ell_2, H )</td>
<td>( \phi_1 )</td>
<td>( \phi_2 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pre-Conditions for main</th>
<th>from ( \ell_5 )</th>
<th>from ( \ell_6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ell_5, \ell_3, \ell_1, H )</td>
<td>( \phi_1 )</td>
<td>( \phi_2 )</td>
</tr>
<tr>
<td>( \ell_5, \ell_4, \ell_2, H )</td>
<td>( \phi_1 )</td>
<td>( \phi_2 )</td>
</tr>
<tr>
<td>( \ell_6, \ell_1, H )</td>
<td>( \phi_1 )</td>
<td>( \phi_2 )</td>
</tr>
<tr>
<td>( \ell_6, \ell_2, H )</td>
<td>( \phi_1 )</td>
<td>( \phi_2 )</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
& \text{t1 q(...) where} \cdots \{ \ell_1, H : \phi_1, \ell_2, H : \phi_2 \}, \{ \} \\
& \text{t2 p(...) where} \cdots \{ \ell_3, \ell_2, H : \phi_3, \ell_4, \ell_1, H : \phi_4 \}, \{ \ell_4, \ell_2, H \} \\
& \text{void main()} \text{where} \cdots \{ \}, \{ \ell_5, \ell_3, \ell_1, H, \ell_6, \ell_2, H \}
\end{align*}
\]

Thus, there are three unsafe checks that must be respecialized at run-time, namely \( \ell_5, \ell_3, \ell_1, H, \ell_6, \ell_2, H \) and \( \ell_5, \ell_4, \ell_2, H \). 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:

\[
\begin{align*}
& \text{void main()} \text{t2 p(...) where} \cdots \{ \ell_1, \ell_2, H : \phi_3, \phi_4 \}, \{ \} \\
& \text{t1 q1(...) where} \cdots \{ \ell_1, L : \phi_1 \}, \{ \} \\
& \text{p(...) where} \cdots \{ \ell_3, \} \\
& \{ \} \quad \text{v1 = (sub(a1,11));} \\
& \quad \text{q1(...) where} \cdots \{ \} \\
& \text{q3(...) where} \cdots \{ \} \\
& \end{align*}
\]

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 q1 and q2 to q3. Though q3 is the weakest specialized 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 q1 configuration occurs infrequently, we may weaken it into q2 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:

\[
P_{\text{flex}} \xrightarrow{\rho} S
\]

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).

\[
\begin{align*}
& (m, \varsigma, m_r) \in \text{SSig = MName \times LSet \times MName} \\
& (\sigma, \sigma_y, \sigma_n, \sigma_{fn}) \in \text{SCache = } P^{\text{SSig}} \\
& \varsigma \in \text{LSet = } P^{\text{Label}^+}
\end{align*}
\]

The specialization of an expression is defined by:

\[
P, \sigma, \varsigma \xrightarrow{\rho_{\text{flex}}} e \xrightarrow{e_1}, \sigma_n
\]

The specialization cache \( \varsigma \) drives the process, while \( \varsigma \) contains the checks to be respecialized. New specialization points created during specialization are stored in \( \sigma_n \). We highlight the most important specialization rules below.

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

\[
\begin{align*}
& \{\text{Spec-Prim}\} \\
& \tau m(x_1, \ldots, x_n) \text{ where } \Delta, \Phi, C \in P_m \\
& \rho \in [x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n] \\
& e_1 = \{ p \in \ell.c \in \varsigma \land (c : e) \in C \} \\
& e_2 = \begin{cases} e_1 \text{ if } e_1 \text{ then } m(v_1, \ldots, v_k) \text{ else error } \\
& e_3 = m(v_1, \ldots, v_k) \land (e_1 = \text{true}) \lor e_2 \\
& \end{cases}
\end{align*}
\]

Here, a label sequence of the form \( \ell.c \) occurring in the set \( \varsigma \) represents an array check to be respecialized. Its code is available at the corresponding primitive method declaration. Variable substitution is needed to respecialize the code. All codes thus generated are combined as a conjunction, 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 \([\text{Spec-Call1}]\). Weakening of configurations by \( \tau \) 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 \([\text{Spec-Call2}]\).

\[
\begin{align*}
& \{\text{Spec-Call1}\} \\
& \tau m(x_1, \ldots, x_n) \text{ where } \Delta, \Phi, \Sigma \{ e \} \in P \\
& \varsigma_2 = W(m, \varsigma_1), \varsigma_1 = \{ e' | \ell_1, e' \in \varsigma \lor \exists \} \\
& (m, \varsigma_2, \varsigma_2) \notin \sigma \xrightarrow{\rho} m_\varsigma(v_1, \ldots, v_k), c_\varsigma = \text{genName}(m_\varsigma, \varsigma_2) \\
& \xrightarrow{P, \sigma, \varsigma \xrightarrow{\rho_{\text{flex}}} (e' : m(v_1, \ldots, v_k)) \rightarrow m_\varsigma(v_1, \ldots, v_k), \emptyset}
\end{align*}
\]

\[
\begin{align*}
& \{\text{Spec-Call2}\} \\
& \tau m(x_1, \ldots, x_n) \text{ where } \Delta, \Phi, \Sigma \{ e \} \in P \\
& \varsigma_1 = \{ e' | \ell_1, e' \in \varsigma \lor \exists \} \\
& (m, W(m, \varsigma_1), m_s) \in \sigma \\
& \xrightarrow{P, \sigma, \varsigma \xrightarrow{\rho_{\text{flex}}} (e' : m(v_1, \ldots, v_k)) \rightarrow m_\varsigma(v_1, \ldots, v_k), \emptyset}
\end{align*}
\]
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 to the technical report [29] for details on the proof and the specialization process.

**Theorem 1 (Soundness).** Let $\Pi$ be a program such that $(P_0 \rightarrow P \sim P_1)$ under type inference. Let $(P_{pre}, P_1 \rightarrow P_{post})$ be the specialization of $P_1$ to $P_{post}$ guided by the inferred runtime tests. Then $P_{post}$ is well-typed.

As a special case, if no unsafe check is discovered during inference then $P_{post}$ is well-typed. However, if unsafe checks are discovered, the use of label sequences $(\ldots, l_4, l_3, \ldots)$ 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 $2^{2^n}$ 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 efficiency 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 pre-condition, it is only safe to strengthen this can be done by simplifying it (via gss [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 $V_s$, we have derived earlier the weakest pre-condition using $\text{pre} = (\forall V_s. \neg \neg \neg c)$. This derived pre-condition may be large due to the presence of disjunctive formulae in $\neg s$. To derive smaller preconditions, we may simplify $\text{pre}$ using a valid state $s_1$ for which $(\exists V_s. s \Rightarrow s_1)$ holds.

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

A second technique is to use $\exists V_s \cdot s$ itself. Let us refer to this technique using $(\text{gst} \ \text{pre} \ \text{given} \ \exists V_s \cdot s)$ 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 $\exists V_s \cdot s$ that is weakened by removing conditional tests from $s$. For example, consider a symbolic program state from the recursive sumvec method:

$$\exists i \cdot s \Rightarrow (0 \leq i < s, \ i + 1)$$

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

$$\exists i \cdot s \Rightarrow (0 \leq i < s, j + 1)$$

Simplifying the precondition of $(j < s) \lor (s \leq i \leq -1) \lor (0 \leq i < s, j, i)$ 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 $(j < s) \lor (s < j \land i < -1) \lor (s \leq j, i)$ 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 1GiBytes 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...
Limpack 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 predervivation takes less time than selective predervivation, followed by weak predervivation. As an exception, the increased precision of weak predervivation allows a faster analysis of mergesort, since some bound checks are proved redundant at an earlier point than the other two predervivation methods. On the other hand, for medium to large-sized programs we found that it is crucial to use either selective or strong predervivation; weak predervivation 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 Rinaidi [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 Pfennig 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 prederviation. 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 formalizes 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


