Deriving Pre-conditions for Array Bound Check Elimination

Wei-Ngan Chin and Siau-Cheng Khoo and Dana N. Xu

School of Computing
National University of Singapore
{chinwn,khoosc,xuna}@comp.nus.edu.sg

Abstract. We present a high-level approach to array bound check optimization that is neither hampered by recursive functions, nor disabled by the presence of partially redundant checks. Our approach combines a *forward analysis* to infer precise contextual constraint at designated program points, and a *backward method* for deriving a safety pre-condition for each bound check. Both analyses are formulated with the help of a practical constraint solver based on Presburger formulae; resulting in an accurate and fully automatable optimization. The derived pre-conditions are also used to guide bound check specialization, for the purpose of eliminating partially redundant checks.

1 Introduction

Array bound check optimization has been extensively investigated over the last three decades [24, 12, 5, 17, 7, 8, 15], with renewed interests as recently as [3, 27, 23]. While successful bound check elimination can bring about measurable gains in performance, the importance of bound check optimization goes beyond these direct gains. In safety-oriented languages, such as Ada or Java, all bound violation must be faithfully reported through precise exception handling mechanism. With this, the presence of bound checks could potentially interfere with other program analyses. For example, data-flow based analysis must take into account potential loss in control flow should array bound violation occurs.

In this paper, we provide fresh insights into the problem of array bound check elimination, with the goal of coming up with a much more precise inter-procedural optimization.

Let us first review the key problem of identifying bound checks for elimination. In general, under a given context, a check can be classified as either:

- *unsafe*;
- *totally redundant*;
- *partially redundant*.

A check is classified as *unsafe* if either a bound violation is expected to occur, or its safety condition is unknown. As a result, we cannot eliminate such a check. A check is classified as *totally redundant* if it can be proven that no bound violation will occur[1]. Lastly, a check is said to be *partially redundant* if we can identify a pre-condition that could ensure that the check becomes redundant.

Note that the classification of a check depends upon a given context. Specifically, a partially redundant check under a given context can become totally redundant when the context becomes "stronger". (Contexts are expressed as predicate.)

To illustrate these three types of checks, consider the following two functions, expressed in a first-order functional language.

$$newsub(arr, i, j) \;=\; \textbf{if}\ (0 \;\leq\; i \;\leq\; j)\ \textbf{then}\ L1@H1@sub(arr, i)\ \textbf{else}\ -1$$
$$last(arr) = \ \textbf{let}\ \ v = length(arr)\ \ \textbf{in}\ L2@H2@sub(arr, v)$$

Arrays used in this paper are assumed to start at index 0, and can be accessed by primitive functions, such as sub. Furthermore, we annotate each array access sub call by some *check labels*, e.g. $L1, H1$, to identify the low and high bound checks respectively. The first function, $newsub$, accesses the element of an array after performing a test on its index parameter i. For the access of $sub(arr, i)$ to be safe, both a low bound check $L1 = i \geq 0$ and a high bound check $H1 = i \;<\; length(arr)$ must be satisfied.

Under the context $0 \;\leq\; i \;\leq\; j$ of the **if**-branch, we can prove that $L1$ is *totally redundant*, but the same cannot be said about $H1$. In fact, the $H1$ check is *partially redundant*, and could be made redundant under appropriate pre-conditions, for e.g. $j < length(arr)$.

The second function is meant to access the last element of a given array but it contains a bug. While the index of our array ranges from 0 to $length(arr) - 1$, this function used an index outside of this range. Hence, its upper bound check $H2 = v < length(arr)$ is *unsafe* as it contradicts with the assertion $v = length(arr)$ from the **let** statement.

Totally and partially redundant checks are traditionally identified by two separate techniques. As a matter of fact, forward data flow analysis[1, 15] which determines *available expressions* has been primarily used to identify totally redundant checks. An expression (or check) e is said to be *available* at program point p if some expression in an equivalence class of e has been computed on every path from entry to p, and the constituent of e has not been redefined in the control flow graph (CFG). Using this information, the computation of an expression (or check) e at point p is redundant if e is available at that point.

Partially redundant checks are more difficult to handle. Traditionally, a backward dataflow analysis [8] is used to determine the *anticipatability* of expressions. An expression (or check) e is anticipatable at program point p if e is computed on every path from p to the exit of the CFG before any of its constituents are redefined. By hoisting an anticipatable expression to its *safe earliest* program point, selected checks can be made totally redundant. Historically, hoisting of anticipatable check is deemed as crucial for eliminating checks from loop-based

[1] This includes the possibility that the check can be safely executed or it can be avoided.

programs. Unfortunately, hoisting of checks causes bound errors to be flagged at an earlier program point, creating problems for precise exception handling.

In this paper, we propose a new approach to eliminating array bound checks. Our approach begins with a forward *contextual-constraint analysis* that synthesize contexts for checks in a program. This is then followed by a backward derivation of *weakest pre-conditions* needed for checks to be eliminated.

For the example given above, our method determines that the lower bound check $L1$ in the function *newsub* is totally redundant; the upper bound check $H2$ in the function *last* is unsafe. Furthermore, the upper bound check $H1$ of *newsub* is determined to be partially redundant; the derived pre-condition being:

$$pre(H1) \equiv (i \leq -1) \vee (j < i \wedge 0 \leq i) \vee (i < length(arr))$$

To overcome the problem arising from hoisting of partially-redundant checks, we propose to use program specialization to selectively enforce contexts that are strong enough for eliminating partially redundant checks. We note that such specialization technique is also advocated by [18] in their bound check optimization of Java programs.

Our new approach is built on top of an earlier work on sized-type inference[4], where we are able to automatically infer *input/output size relation* and also determine *invariants* for parameters of recursive functions over the sizes of data structures used. The inference is performed accurately and efficiently with the help of a constraint-solver on Presburger form[22]. The presence of sized type greatly enhances inter-procedural analysis of *contextual constraints*, which are crucial for identifying both unsafe and totally redundant checks. More importantly, accurate contextual constraint also helps in the derivation of safety pre-conditions for partially-redundant checks. With the derived pre-condition, we can provide specialized code to selectively eliminate partially-redundant checks based on the available contexts. The specialization process can be further tuned to provide a range of time/space tradeoff.

Our main contributions are:

1. To the best of our knowledge, we are the first to handle *partially redundant* checks through the backward derivation of *safety pre-condition* after contextual constraint has been gathered in a separate forward phase. This gives very accurate result for eliminating partially-redundant checks.
2. We deal directly with recursive functions, and the invariant synthesis is performed only once for each recursive function, instead of for every occurrence of checks within the function. Except for [26, 23] whose methods are restricted to totally redundant checks, almost all previous work for bounds check elimination deal with only loop-based programs.
3. We design a simple yet elegant approach to derive the weakest pre-condition (with respect to a given contextual constraint) for check elimination from the context of the check and the synthesized invariant. Our approach works seamlessly across recursive procedures.
4. We support inter-procedural optimization through backward propagation of a function's pre-condition to its callers to become a check.

5. We introduce three forms of bound check specialization: *polyvariant* for maximal specialization, *monovariant* for minimal code duplication, and *duovariant* specialization for a space/time tradeoff. While the idea of using context-based program specialization [16, 6] is not new, our work is novel in its use of pre-condition for guiding effective specialization.

Section 2 gives an overview of our method by introducing sized types and the main steps towards bound check specialization. Section 3 formalizes the context synthesis as a forward analysis method. It also illustrates how invariants on recursive functions can be synthesized, so as to provide informative contexts for recursive functions. Section 4 describes the key steps for classifying checks, and the inter-procedural mechanism for deriving pre-conditions for each partially redundant check. Section 5 shows how the derived pre-conditions can be used to guide bounds check specialization; while Section 6 shows that the cost of analysis is within acceptable limit. Related work is compared in Section 7, before we discuss some future directions of research in the Section 8.

$$
\begin{array}{llll}
x \in \mathbf{Var} & \langle\text{Variables}\rangle & a \in \mathbf{Arr} & \langle\text{Array Names}\rangle \\
f \in \mathbf{Fname} & \langle\text{Function Names}\rangle & n \in \mathbf{Int} & \langle\text{Integer Constants}\rangle \\
L \in \mathbf{Label} & \langle\text{Labels for checks}\rangle & & \\
p \in \mathbf{Prim} & \langle\text{Primitives}\rangle & & \\
\quad p ::= + \mid - \mid * \mid / \mid > \mid = \mid != \mid < \mid >= \mid & & & \\
\qquad <= \mid not \mid or \mid and \mid length \mid newArr & & & \\
\kappa \in \mathbf{Call} & \langle\text{Calls}\rangle & & \\
\quad \kappa ::= L@\kappa \mid f\ (x_1, \ldots, x_n) \mid sub(a, x) \mid update(a, x_1, x_2) & & & \\
e \in \mathbf{Exp} & \langle\text{Expressions}\rangle & & \\
\quad e ::= x \mid n \mid p\ (x_1, \ldots, x_n) \mid \kappa \mid \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 & & & \\
d \in \mathbf{Def} & \langle\text{Function Definition}\rangle & & \\
\quad d ::= f\ (x_1, \ldots, x_n) = e & & &
\end{array}
$$

Fig. 1. The Language Syntax

2 Overview

We apply our technique to first-order typed functional language with strict semantics. Recursive functions in the language are confined to self-recursion. Currently, mutual recursion are encoded into self-recursion by appropriate tagging of input and output. The language is defined in Fig. 1. Note that the language syntax includes *check labels* (also called *labels* for brevity) that identify bound checks (*ie.*, array bound checks or checks that originated from these bound checks). Check labels appears syntactically at calls to functions/operations that involve bound checks. However, We do not label self-recursive calls, as we provide slightly different treatment to recursive function definitions (as explained in Section 3.2). Lastly, check labels are automatically inserted into programs by our analysis.

We restrict the arguments to a function to be just variables. This simplifies presentation, without loss of generality.

Sized Type = (**AnnType**, **F**)

Annotated Type Expressions:

$v \in \mathbf{V}$ ⟨Size Variables⟩ $t \in \mathbf{TVar}$ ⟨Type Variables⟩

$\sigma \in \mathbf{AnnType}$ ⟨Annotated Types⟩

$\sigma ::= \forall\, t\,.\,\sigma \mid \tau \mid \tau \rightarrow \tau$

$\tau \in \mathbf{Basic}$ ⟨Basic Type⟩

$\tau ::= t \mid (\tau_1, .., \tau_n) \mid \mathtt{Arr}^v\, \tau \mid \mathtt{Int}^v \mid \mathtt{Bool}^v$

Presburger Formulae:

$n \in \mathcal{Z}$ ⟨Integer constants⟩ $v \in \mathbf{V}$ ⟨Variable⟩

$\phi \in \mathbf{F}$ ⟨Presburger Formulae⟩

$\phi ::= b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\, \phi \mid \exists\, v\,.\,\phi \mid \forall\, v.\,\phi$

$b \in \mathbf{BExp}$ ⟨Boolean Expression⟩

$b ::= \mathit{True} \mid \mathit{False} \mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 < a_2$
$\mid a_1 > a_2 \mid a_1 \leq a_2 \mid a_1 \geq a_2$

$a \in \mathbf{AExp}$ ⟨Arithmetic Expression⟩

$a ::= n \mid v \mid n * a \mid a_1 + a_2 \mid -\, a$

Fig. 2. Syntax of Sized Types

We only consider *well-typed* programs. We enhance the type system with the notion of *sized types*, which captures the size information about the underlying expressions/values. For a function, sized type reveals size relationships amongst the parameters and results of that function. The syntax of sized type is depicted in Fig. 2. It is a pair containing an annotated type and a Presburger formula. An annotated type expression augments an ordinary type expression with size variables; the relationship among these variables are expressed in the associated formula. In this paper, we consider only three basic types: Arrays, integers, and booleans. The annotated type for arrays is $\mathtt{Arr}^v\, \tau$, where v captures the array size; for integers, it is $\mathtt{Int}^v$, where v captures the integer value; for booleans, it is $\mathtt{Bool}^v$, where v can be either 0 or 1, representing the values *False* and *True* respectively. Occasionally, we omit writing size variables in the annotated type when these variables are unconstrained.

A sample program for our language is shown in Fig. 3. This program contains four functions that implement binary search. The main function *bsearch* takes

```
getmid :: (Arr^a Int, Int^l, Int^h) → (Int^m, Int)
        Size  a ≥ 0 ∧ 2m ≤ l + h ∧ l + h ≤ 1 + 2m
getmid(arr, lo, hi) =  let m = (lo + hi)/2
                         in   let  x = L3@H3@sub arr m  in (m, x)
cmp :: (Int^i, Int^j) → Int^r
        Size  (i < j ∧ r = −1) ∨ (i = j ∧ r = 0) ∨ (i > j ∧ r = 1)
cmp(k, x)  =  if k < x then − 1 else if k = x then 0 else 1
look :: (Arr^a Int, Int^l, Int^h, Int) → Int^r
        Size (a ≥ 0) ∧ ((l ≤ h) ∨ (l > h ∧ r = −1))
        Inv  a* = a ∧ l ≤ h, l* ∧ h* ≤ h ∧
             2 + 2l + 2h* ≤ h + 3l* ∧ l + 2h* < h + 2l*
look(arr, lo, hi, key) =
        if (lo <= hi) then
           let (m, x) = L4@H4@getmid(arr, lo, hi)
           in   let t = cmp(key, x)
                in   if t < 0 then look(arr, lo, m − 1, key)
                     else if (t == 0) then m else look(arr, m + 1, hi, key)
        else    − 1
bsearch :: (Arr^a Int, Int) → Int
        Size (a ≥ 0)
bsearch(arr, key)  =  let  v = length(arr)  in  L5@H5@look(arr, 0, v − 1, key)
```

Fig. 3. Binary Search Program

an array and a key in order to search for an element in the array. If found, the corresponding array index is returned, otherwise -1 is returned. The recursive invocation of binary search is carried out by the function *look*.

2.1 Use of Sized Types

Sized type of a function captures the relationship between sizes of the function's input and output. For instance, the annotated type for function *cmp* is $(\mathtt{Int}^i, \mathtt{Int}^j) \rightarrow \mathtt{Int}^r$, where i, j are the respective input values, and r is its output. The size constraint (identified by the keyword **Size**) states three possible outputs for calling *cmp*, depending on whether the argument k is less than, equal to, or greater than the argument x.

Even more importantly, through sized-type inference [4], we can synthesize, for a recursive function, an invariant that describes changes in size of input arguments of the function during its nested recursive-call invocations. For example, an accurate invariant relationship between the (first three) argument sizes of any nested recursive calls to *look*, $a^\star, l^\star, h^\star$, and the (first three) parameter sizes of the initial first call to *look*, namely a, l, h, has been captured as the following Presburger formula :

$$inv(look) \;=\; a^\star \;=\; a \;\wedge\; l \;\leq\; h,\; l^\star \;\wedge\; h^\star \;\leq\; h \;\wedge$$
$$2 + 2l + 2h^\star \;\leq\; h + 3l^\star \;\wedge\; l + 2h^\star \;<\; h + 2l^\star$$

This invariant tells us that, in the successive recursive invocations of *look*, the size of the first argument remains unchanged. Also, the values of the argument *lo* ($l^\star$) in the successive calls never get smaller than its initial value l, while those of *hi* ($h^\star$) never get bigger than the initial value h. For example, if the first call to *look* is $look(2, 10)$, we know that successive recursive calls $look(lo, hi)$ will satisfy the relationship $lo \geq 2 \wedge hi \leq 10$. Terminology-wise, we call the initial set of arguments/size variables (eg., a, l, and h) the *initial arguments/sizes*, and that of an arbitrarily nested recursive call (eg., $a^\star$, $l^\star$, and $h^\star$) the *recursive arguments/sizes*.

$$\begin{array}{ll}
+ \;::\; (\texttt{Int}^i,\, \texttt{Int}^j) \to \texttt{Int}^k & - \;::\; (\texttt{Int}^i,\, \texttt{Int}^j) \to \texttt{Int}^k \\
\quad \textbf{Size}\; k = i + j & \quad \textbf{Size}\; k = i - j \\
= \;::\; (\texttt{Int}^i,\, \texttt{Int}^j) \to \texttt{Bool}^b & \\
\quad \textbf{Size}\; (0 \leq b \leq 1) \wedge ((i = j \wedge b = 1) \vee (i \neq j \wedge b = 0)) & \\
sub \;::\; ((\texttt{Arr}^a\, \tau),\, \texttt{Int}^i) \to \tau & update \;::\; ((\texttt{Arr}^a\, \tau),\, \texttt{Int}^i,\, \tau) \to () \\
\quad \textbf{Size}\; a \geq 0 & \quad \textbf{Size}\; a \geq 0 \\
\quad \textbf{Req}\; L : i \geq 0 \;;\; H : i < a & \quad \textbf{Req}\; L : i \geq 0 \;;\; H : i < a \\
length \;::\; (\texttt{Arr}^a\, \tau) \to \texttt{Int}^i & newArr \;::\; (\texttt{Int}^i, \tau) \to \texttt{Arr}^a\, \tau \\
\quad \textbf{Size}\; a \geq 0 \wedge a = i & \quad \textbf{Size}\; (i \geq 0 \wedge a = i)
\end{array}$$

Fig. 4. Sized Types of Some Primitives

Fig. 4 depicts the sized types of a collection of primitive functions used in the rest of this paper. For array-access operations (*sub* and *update*), we also include their respective pre-conditions, which must be satisfied for the operations to be safe. These pre-conditions are identified by the keyword **Req**.

Once the sized types of related functions have been inferred, we proceed to handle bound check optimization. The process works in a bottom-up fashion, starting with functions at the bottom of the call hierarchy. We list the steps involved below. Throughout the rest of the paper, we use the binary search program depicted in Fig. 3 as the running example.

Step 1 Forward contextual-constraint analysis.
Step 2 Backward pre-condition derivation.
Step 3 Bound check specialization.

These steps are described in details in the following sections.

3 Context Synthesis

We begin by determining the context within which a check occurs. Contextual information is described in Presburger form. It is called *contextual constraint*, and is gathered by traversing the syntax tree of the function body, beginning from the root of the tree to a check-labelled call. Constraints gathered during traversal include constraint for selecting a branch (of **if**-expression), assertion about the sizes of local variables, and post-conditions of function calls.

$$
\begin{array}{l}
\mathcal{C} \ :: \ \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{F} \rightarrow (\mathbf{AnnType} \times \mathcal{P}((\mathbf{Label}, \mathbf{F})) \times \mathbf{F}) \\
\qquad\quad \textit{where}\ \mathbf{Env} = \mathbf{Var} \rightarrow \mathbf{AnnType} \times \mathbf{F} \\
\mathcal{C}[\![x]\!]\ \Gamma\, \psi = \ \textit{let}\ (\tau, \phi) = \Gamma[\![x]\!] \ \ \textit{in}\ (\tau, \emptyset, \phi) \\
\mathcal{C}[\![n]\!]\ \Gamma\, \psi = \textit{let}\ v = \textit{newVar} \ \ \textit{in}\ (\mathtt{Int}^v, \emptyset, (v = n)) \\
\mathcal{C}[\![f\ (x_1, \ldots, x_n)]\!]\ \Gamma\, \psi = \textit{let}\ ((\tau_1, \ldots, \tau_n) \rightarrow \tau, \phi_f) = \alpha\,(\Gamma[\![f]\!]) \\
\qquad\qquad\qquad\qquad\qquad\qquad X = \cup_{i=1}^{n}\ \{fv(\tau_i)\} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\tau_i', \phi_i) = \Gamma[\![x_i]\!]\, \forall\, i \in \{1, \ldots, n\} \\
\qquad\qquad\qquad\qquad\qquad\qquad \phi = \exists\, X\, .\ \phi_f \wedge (\wedge_{i=1}^{n}\, (\phi_i \wedge (eq\ \tau_i'\ \tau_i))) \\
\qquad\qquad\qquad\qquad\qquad\quad \textit{in}\ (\tau, \emptyset, \phi)
\end{array}
$$

⟨Treatment of primitive operations is the same as that of function application.⟩

$$
\begin{array}{l}
\mathcal{C}[\![L@e]\!]\ \Gamma\, \psi = \textit{let}\ (\tau, \beta, \phi) = \mathcal{C}[\![e]\!]\ \Gamma\, \psi \\
\qquad\qquad\qquad\quad \textit{in}\ (\tau, \{(L, \mathcal{F}_{\Gamma,\psi})\} \cup \beta, \phi) \\
\mathcal{C}[\![\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]\ \Gamma\, \psi = \\
\qquad\quad \textit{let}\ (\mathtt{Bool}^v, \beta_0, \phi) = \mathcal{C}[\![e_0]\!]\ \Gamma\, \psi \\
\qquad\qquad (\tau_1, \beta_1, \phi_1) = \mathcal{C}[\![e_1]\!]\ \Gamma\ (\psi \wedge \phi \wedge (v = 1)) \\
\qquad\qquad (\tau_2, \beta_2, \phi_2) = \mathcal{C}[\![e_2]\!]\ \Gamma\ (\psi \wedge \phi \wedge (v = 0)) \\
\qquad\qquad \tau_3 = \alpha\,(\tau_1) \\
\qquad\qquad Y = \{\, v \,\} \cup fv(\tau_1) \cup fv(\tau_2) \\
\qquad\qquad \phi_3 = \exists\, Y\, .\, \phi \wedge (((eq\ \tau_1\ \tau_3) \wedge (v = 1) \wedge \phi_1) \\
\qquad\qquad\qquad\qquad \vee\ ((eq\ \tau_2\ \tau_3) \wedge (v = 0) \wedge \phi_2)) \\
\qquad\quad \textit{in}\ (\tau_3, \beta_0 \cup \beta_1 \cup \beta_2, \phi_3) \\
\mathcal{C}[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!]\ \Gamma\, \psi = \\
\qquad\quad \textit{let}\ (\tau_1, \beta_1, \phi_1) = \mathcal{C}[\![e_1]\!]\ \Gamma\ \psi \\
\qquad\qquad (\tau, \beta, \phi) = \mathcal{C}[\![e_2]\!]\ \Gamma[x :: (\tau_1, \phi_1)]\ \psi \\
\qquad\qquad Y = fv(\tau_1) \\
\qquad\qquad \phi_2 = \exists\, Y\, .\, (\phi_1 \wedge \phi) \\
\qquad\quad \textit{in}\ (\tau, \beta_1 \cup \beta, \phi_2)
\end{array}
$$

Fig. 5. Definition of the Context-Derivation Function $\mathcal{C}$

Forward analysis $\mathcal{C}$ is employed to synthesize contextual constraints. This is depicted in Fig. 5. We first explain how this is done for non-recursive function, and describe the recursive case in the following section.

$\mathcal{C}$ operates on expressions. It takes in a sized-type environment Γ which binds the program variables, primitives, and user-defined functions to their respective sized types. It produces a triple consisting of: the annotated type of the subject expression, a set of bindings between call labels appearing in the expression and their contextual constraints, and the size constraint of the subject expression.

During the traversal of the syntax tree, $\mathcal{C}$ updates Γ with sized types of locally defined variables. It also maintains a constraint ψ that captures the context of the subject expression. Initially, ψ is set to the value *True*. When a branch of an **if**-expression is chosen, the constraint leading to this decision is captured in ψ. When a labelled call is encountered, its contextual constraint is derived by combining (via conjunction) ψ with related constraints kept in the environment ρ. The result is expressed as $\mathcal{F}_{\Gamma,\psi}$. Formally, it is defined as follows:

$$\begin{array}{lll} \mathcal{F}_{\Gamma,\psi} & = \wedge\,(\cup_{i\,\geq\,0}\,\Phi_i) & \textbf{where} \\ \quad \Phi_0 & = \{\psi\} & \\ \quad \Phi_{i+1} & = \{\,\phi \mid \exists\, x,\, \tau\,.\, \Gamma\,[\![\, x \,]\!] \,=\, (\tau,\phi)\,;\, fv(\phi) \,\cap\, fv(\Phi_i) \,\neq\, \emptyset\,;\, \phi \,\notin\, \cup_{j\,\leq\,i}\, \Phi_j \,\} & \end{array}$$

As the environment Γ is finite, computation of $\mathcal{F}_{\Gamma,\phi}$ always terminates.

Notation-wise, in Fig. 5, function *newVar* returns a new variable. Function α performs renaming of size variables (like α-conversion). It is overloaded so that it can take in either an annotated type or a sized type (which is a pair). It consistently renames all size variables occurring in its argument. Lastly, operation $eq\ \tau_1\ \tau_2$ produces a conjunction that equates the corresponding size variables of two annotated types. For instance, $(eq\ \mathtt{Int}^v\ \mathtt{Int}^w)$ produces the constraint $(v\,=\,w)$. Lastly, $\Gamma[x\ ::\ (\tau,\phi)]$ denotes updating of the environment Γ by a new binding of x to a sized type (τ,ϕ).

As an example, for the following function definition,

$$\begin{array}{l} newsub\ ::\ (\mathtt{Arr}^a\ \mathtt{Int},\ \mathtt{Int}^i,\mathtt{Int}^j)\ \rightarrow\ \mathtt{Int} \\ newsub(arr,i,j)\ =\ \textbf{if}\ 0\ \leq\ i\ \leq\ j\ \textbf{then}\ L1@H1@sub(arr,i)\ \textbf{else}\ -1 \end{array}$$

$\mathcal{C}$ determines the context for the labelled call to be:

$$ctx(L1)\ =\ ctx(H1)\ =\ a\ \geq\ 0\ \wedge\ 0\ \leq\ i\ \leq\ j$$

3.1 Recursive-Call-Invariant Synthesis

Invariant synthesis is in general a hard problem for recursive function definitions. Two pieces of invariant information are useful. First, invariant describing input/output size relation (ie., its sized type) of a recursive function can be propagated across functions to achieve better context synthesis. Computing such invariant is not always possible, however, as sometimes the precise relation between input- and output-size is beyond Presburger formulation.

Second, *recursive-call invariant* captures the argument-size relationship between an initial call and an arbitrarily nested recursive call of the same function. This relation is needed for synthesizing the contextual constraint of a labelled call invoked at arbitrarily nested depth. Fortunately, such relation can often be formulated precisely using Presburger formula.

Computation of recursive-call invariant proceeds as follows: We first compute the constraint relating the parameter sizes of a function and the argument sizes of all recursive calls textually occurring in the function body. Conceptually, this constraint spells out the change in argument size during *one unfolding* of the recursive call. It can be captured by a procedure similar to the context computation $\mathcal{C}$. For example, consider the function *look* defined in the binary search program. We obtain the following constraint, which we call U:

$$\begin{aligned} U \;:=\; [a,l,h] \;\rightarrow\; & [a^\star, l^\star, h^\star] \;:\; h \;\geq\; l \wedge \\ & \exists\,(m\,.\,2m \;\leq\; h+l \;\wedge\; h+l \;\leq\; 1 \;+\; 2m \wedge \\ & \;((l^\star = l \;\wedge\; h^\star = m-1) \;\vee\; (l^\star = m+1 \;\wedge\; h^\star = h))) \end{aligned}$$

The notation used here is adapted from literature work in Omega Calculator [22]. It specifies the constraint between two sets of variables: $[a,l,h]$ and $[a^\star, l^\star, h^\star]$. The former is the set of initial parameter sizes (they are called the *source*), and the latter being the set of argument sizes of a recursive call (they are called the *target*).

Next, we perform inductive computation to infer the change in argument size resulting from arbitrary number of recursive-call unfolding. The result is the recursive-call invariant. In the case of *look*, we have:

$$\begin{aligned} \mathit{inv}(\mathit{look}) \;=\; & a^\star \;=\; a \;\wedge\; l \;\leq\; h,\; l^\star \;\wedge\; h^\star \leq\; h \;\wedge \\ & 2+2l+2h^\star \;\leq\; h+3l^\star \;\wedge\; l+2h^\star \;<\; h+2l^\star \end{aligned}$$

Several researchers, including the present authors, have proposed different techniques for synthesizing invariants. As these techniques are complementary in power and efficiency, we believe a collection of these techniques is needed to do a decent job. This includes:

Polyhedra analysis. This is proposed and developed by Cousot and Halbwachs [5, 11, 10], as well as King and his co-workers [2, 14]. It is an abstract interpretation approach to finding the input/output size relation through fixed-point computation over linear constraint. Both convex-hull operation (to eliminate multiple disjuncts) and widening operation (to generalize a constraint by dropping some conjuncts that cannot be subsumed by others in an ascending chain of constraints) are used as generalization techniques to ensure termination of the analysis. For recursive-call invariant computation, we modify this analysis by ignoring the degenerated case of a recursive definition from our computation. As an example, Fig. 6 illustrates a trace of such computation for the function *look* with the aid of the Omega calculator.

In the above, lines begin with # are comments; lines end with ; are commands to the Omega Calculator [13]; outputs from the Calculator are indented rightward. (*hull* U) computes the convex hull of U (viewed as a relation) and *widen*(U_2, U_3) generalizes U_2 to yield a constraint W_2 such that both U_2 and U_3 are instances of W_2. (We refer the reader to the work of Halbwachs [9, 10] for detail description of these two operations.) *union* signifies disjunction, *compose*

$$
\begin{array}{l}
\#\ \text{First Approximation} \\
U_1 \ := \ \mathit{hull}\ U\ ; \\
\quad U_1 \ = \ [a, l, h] \ \rightarrow \ [a^\star, l^\star, h^\star] \ : \ a \ = \ a^\star \ \wedge \ l \ \leq \ h, \ l^\star \ \wedge \ h^\star \ \leq \ h \wedge \\
\qquad\qquad l + 2h^\star \ < \ h + 2l^\star \ \wedge \ 2 + 2l + 2h^\star \ \leq \ h + 3l^\star \ \wedge \\
\qquad\qquad h + 2l^\star \ \leq \ 3 + l + 2h^\star \ \wedge \ h + 4l^\star \ \leq \ 4 + 2l + 3h^\star \\
\#\ \text{Second Approximation} \\
U_2 \ := \ \mathit{hull}\ (U_1\ \mathit{union}\ (U_1\ \mathit{compose}\ U))\ ; \\
\quad U_2 \ = \ [a, l, h] \ \rightarrow \ [a^\star, l^\star, h^\star] \ : \ a \ = \ a^\star \ \wedge \ l \ \leq \ h, \ l^\star \ \wedge h^\star \ \leq \ h \wedge \\
\qquad\qquad h + 4l^\star \ \leq \ 9 + l + 4h^\star \ \wedge \ l + 2h^\star \ < \ h + 2l^\star \wedge \\
\qquad\qquad 2 + 2l + 2h^\star \ \leq \ h + 3l^\star \\
\#\ \text{Third Approximation} \\
U_3 \ := \ \mathit{hull}\ (U_2\ \mathit{union}\ (U_2\ \mathit{compose}\ U))\ ; \\
\quad U_3 \ = \ [a, l, h] \ \rightarrow \ [a^\star, l^\star, h^\star] \ : \ a \ = \ a^\star \ \wedge \ l \ \leq \ h, \ l^\star \ \wedge \ h^\star \ \leq \ h \wedge \\
\qquad\qquad h + 8l^\star \ \leq \ 21 + l + 8h^\star \ \wedge \ 2 + 2l + 2h^\star \ < \ h + 3l^\star \ \wedge \\
\qquad\qquad l + 2h^\star \ \leq \ h + 2l^\star \\
\#\ \text{Apply Generalization by Widening} \\
W_2 \ := \ \mathit{widen}(U_2, U_3)\ ; \\
\quad W_2 \ = \ [a, l, h] \ \rightarrow \ [a^\star, l^\star, h^\star] \ : \ a \ = \ a^\star \ \wedge \ l \ \leq \ h, \ l^\star \ \wedge \ h^\star \ \leq \ h \wedge \\
\qquad\qquad 2 + 2l + 2h^\star \ \leq \ h + 3l^\star \ \wedge \ l + 2h^\star \ < \ h + 2l^\star \\
\#\ \text{Is the result a fixed point?} \\
(W_2\ \mathit{compose}\ U)\ \mathit{subset}\ W_2; \\
\quad \mathit{True}
\end{array}
$$

Fig. 6. A trace of Omega Calculation of Recursive-call Invariant

combines two constraints by matching (and eliminating) the target of the former with the source of the latter. The second and third steps of the above trace above are iterative computation of fixed-point computation. The last command checks if a fixed-point is reached.

Transitive-closure operation. This is a fixed-point operation provided in the Omega Calculator [22]. Given a linear constraint expressed in the form of relation (such as U above), the transitive-closure operation aims to compute its *least* fixed point. A least fixed-point of U is defined as $\bigvee_{i>0} (U^i)$, where $U^1 = U$, and $U^{i+1} = U^i \ \mathit{compose}\ U$. A "shortcoming" in this operation is that it does not support generalization to give an approximate fixed point, if the least fixed point cannot be found.

Generalized transitive closure. To overcome the limitation of Omega's transitive-closure operation, we introduced in [4] the concept of generalized transitive closure with selective generalization. Basically, it introduces generalizations of size relation based on selective grouping and hulling of its various disjuncts. While hulling aids in ensuring termination of fixed-point computation at the expense of accuracy, selective grouping and hulling help maintain accuracy of such computation.

3.2 Context Synthesis for Recursive Functions

For recursive functions, our analysis must derive the most informative contextual constraint that is applicable to *all* recursive invocations of the function, including the degenerated case. For a more accurate analysis, our method differentiates two closely-related contexts: (a) The context of a labelled call encountered during the *first* time the function call is invoked; ie., before any nested recursive call is invoked. (b) The context of a labelled call encountered after some invocations of nested recursive calls. The reason for this separation is because the latter context is computed using the synthesized recursive-call invariant.

The contextual constraint of the first call is analyzed in the same way as that for non-recursive function. For each label L of a recursive function f, the context of the first call is:

$$ctxFst(L) \;=\; ctx(L) \;\wedge\; ctxSta(f)$$

where $ctx(L)$ is the derived contextual constraint at program point L, and $ctxSta(f)$ denotes the default context that can be assumed at procedural entry of f. For function $look$, $ctxSta(look) = a \geq 0$. (ie., the array must be of non-negative length.)

The contextual constraint for a labelled call encountered after subsequent recursive invocations of f-calls can be computed using:

$$ctxRec(L) \;=\; inverse(ctx(L)) \;\wedge\; inv(f) \;\wedge\; ctxSta(f)$$

Note that we make use of the synthesized invariant of f, namely $inv(f)$, while the *inverse* operation (as defined in the Omega Calculator) is used to obtain a mirror copy of $ctx(L)$ that applies to the recursive sizes (instead of the initial sizes).

Separate identification of contexts for both the first recursive call and subsequent recursive calls is instrumental to obtaining more accurate contextual constraints, which in turns induce more precise pre-condition for eliminating recursive checks.

For the function $look$, the labels used are $L4$ and $H4$. The context enclosing the labelled call is found to be $l \leq h$. Following the above procedure, we obtain the following contextual constraints:

$$\begin{array}{ll}
ctx(L4) & = l \leq h \\
ctxSta(look) & = a \geq 0 \\
inverse(ctx(L4)) & = l^\star \leq h^\star \\
inv(look) & = a = a^\star \wedge l \leq h,\ l^\star \wedge h^\star \leq h \wedge \\
 & \quad 2 + 2l + 2h^\star \leq h + 3l^\star \wedge l + 2h^\star < h + 2l^\star \\
ctxFst(L4) & = l \leq h \wedge a \geq 0 \\
ctxRec(L4) & = a = a^\star \wedge l \leq l^\star \leq h^\star \leq h \wedge 0 \leq a \wedge \\
 & \quad 2 + 2l + 2h^\star \leq h + 3l^\star \wedge l + 2\,h^\star < h + 2l^\star
\end{array}$$

4 Pre-Condition Derivation

The synthesis of contexts and invariants is essentially a forward analysis that gathers information about how values are computed and propagated and how the conditions of **if**-branches are inherited. In contrast, the derivation of pre-condition for check elimination is inherently a backward problem. Here, the flow of information goes from callee to caller, with the goal of finding weakest possible pre-condition which ensures that the operation can be performed safely without checking.

We propose a backward method for deriving safety pre-conditions. This method considers each function in turn, starting from the lowest one in the calling hierarchy. Our method attempts to derive the required pre-condition to make each check redundant. Working backwards, each pre-condition that we derive from a callee would be converted into a check for its caller. In this way, we are able to derive the pre-condition for each check, including those that are nested arbitrarily deep inside procedural calls. The main steps are summarized here.

- Determine each check to see if it is either unsafe, totally redundant or partially redundant.
- Derive a safety pre-condition for each partially redundant check. Checks from recursive functions must take into account the recursive invocations.
- Amalgamate related checks together.
- To support inter-procedural propagation, convert each required pre-condition of a function into a check at its call site based on the parameter instantiation.

To help describe our method, consider the following simple example:

$$\begin{array}{lll} p(arr,i,j) & = & \textbf{if}\ 0 \le\ i\ \le\ j\ \textbf{then}\ L6@H6@sub(arr,i) + \\ & & \qquad\qquad\qquad\quad L7@H7@sub(arr,i-1) \\ & & \textbf{else}\ -1 \end{array}$$

This function is a minor modification of *newsub*. It takes an array and two integers i and j, and returns the sum of elements at i and $i-1$ if $0\ \le\ i\ \le\ j$, otherwise -1 is returned. From the definition of this procedure, we can provide the following sized type for p:

$$\begin{array}{l} p :: (\mathtt{Arr}^m\ \mathtt{Int}, \mathtt{Int}^i, \mathtt{Int}^j) \ \to\ \mathtt{Int}^r \\ \quad \textbf{Size}\ m\ \ge\ 0\ \wedge\ ((0\ \le\ i\ \le\ j)\ \vee\ ((i<0\ \vee\ (i>j\ \wedge\ i\ \ge\ 0))\ \wedge\ r=-1)) \end{array}$$

4.1 Check Classification

We classify each check as either *totally redundant*, *partially redundant* or *unsafe*. Given a check $chk(L)$ under a context $ctx(L)$, we can capture the weakest pre-condition, $pre(L)$ that enables $chk(L)$ to become redundant. The weakest pre-condition is computed using:

$$pre(L) \equiv \neg ctx(L) \vee chk(L)$$

This pre-condition should be simplified[2] using the invariant context at procedure entry, namely $ctxSta(p)$, whose validity would be verified by our sized-type system. If $pre(L) \equiv True$, we classify the check as totally redundant. If $pre(L) \equiv False$ (or unknown due to the limitation of Presburger solver), we classify the check as unsafe. Otherwise, the check is said to be partially redundant.
Example : The four checks in p are:

$$
\begin{array}{ll}
chk(L6) = i \geq 0 & chk(H6) = i < m \\
chk(L7) = i - 1 \geq 0 & chk(H7) = i - 1 < m
\end{array}
$$

Of these four checks, only the pre-condition of check at $L6$, namely $pre(L6) \equiv \neg ctx(L6) \vee chk(L6)$ evaluates to $True$. Hence, $chk(L6)$ is redundant, while the other three checks are partially redundant. In this example, we use the following contextual constraints:

$$
\begin{array}{l}
ctx(L6) = ctx(L7) = ctx(H6) = ctx(H7) \\
ctx(L6) = ctxSta(p) \wedge (0 \leq i \leq j) \quad \text{and} \quad ctxSta(p) = m \geq 0
\end{array}
$$

4.2 Derivation of Pre-condition

The derivation of $pre(L)$ is to a large extent dependent on $ctx(L)$. A more informative $ctx(L)$ could lead to a better $pre(L)$. For a given contextual constraint $ctx(L)$, $pre(L)$ can be computed by:

$$pre(L) \equiv \neg ctx(L) \vee chk(L)$$

The following lemma characterizes $pre(L)$ as the weakest pre-condition. We omit the proof in this paper.

Lemma 1 *The weakest pre-condition (pre) for the safe elimination of a check (chk) in a given context (ctx) is* $pre \equiv \neg ctx \vee chk$.

Example : Using the above formulae, we can derive the following pre-conditions for the three partially redundant checks:

$$
\begin{array}{l}
pre(H6) = (i \leq -1) \vee (j < i \wedge 0 \leq i) \vee (i < m) \\
pre(L7) = (i \leq -1) \vee (j < i \wedge 0 \leq i) \vee (i \geq 1) \\
pre(H7) = (i \leq -1) \vee (j < i \wedge 0 \leq i) \vee (i \leq m)
\end{array}
$$

Deriving pre-conditions for the elimination of checks from recursive procedure is more challenging. A key problem is that the check may be executed repeatedly, and any derived pre-condition must ensure that the check is completely eliminated. One well-known technique for the elimination of checks from loop-based program is the *loop limit substitution* method of [7]. Depending on the direction of monotonicity, the check of either the first or last iteration of the loop is used as a condition for the elimination of all checks. However, this

[2] In Omega, the simplification can be done by a special operator, called *gist*.

method is restricted to checks on *monotonic* parameters whose limits can be precisely calculated.

We propose a more general method to handle recursive checks. For better precision, our approach separates out the context of the initial recursive call from the context of the subsequent recursive calls. The latter context may use the invariant of recursive parameters from sized typing.

Using the recursive *look* function (whose parameters are non-monotonic) as an example, we shall provide two separate checks for the first and subsequent recursive calls, namely:

$$chkFst(L4) = 0 \leq l + h \qquad \text{and} \qquad chkRec(L4) = 0 \leq l^{\star} + h^{\star}$$

with their respective contexts:

$$\begin{array}{ll} ctxFst(L4) & = a \geq 0 \wedge l \leq h \\ ctxRec(L4) & = a \geq 0 \wedge l^{\star} \leq h^{\star} \wedge inv(look) \\ inv(look) & = a = a^{\star} \wedge l \leq l^{\star}, h \wedge h^{\star} \leq h \wedge \\ & \quad 2 + 2l + 2h^{\star} \leq h + 3l^{\star} \wedge l + 2h^{\star} < h + 2l^{\star} \end{array}$$

We next derive the pre-conditions for the two checks separately, as follows:

$$\begin{array}{ll} preFst(L4) & = \neg ctxFst(L4) \vee chkFst(L4) \\ & = (h < l) \vee (0 \leq l + h) \\ preRec(L4) & = \neg ctxRec(L4) \vee chkRec(L4) \\ & = \forall\, l^{\star}, h^{\star}.\ \neg(a \geq 0 \wedge l^{\star} \leq h^{\star} \wedge l \leq h, l^{\star} \wedge h^{\star} \leq h \wedge \\ & \quad 2 + 2l + 2h^{\star} \leq h + 3l^{\star} \wedge l + 2h^{\star} < h + 2l^{\star}) \vee (0 \leq l^{\star} + h^{\star}) \\ & = (h \leq l) \vee (0 \leq l < h) \vee (l = -1 \wedge h = 0) \end{array}$$

Note that *preRec* is naturally expressed in terms of the recursive variables. However, we must re-express each pre-condition in terms of the initial variables. Hence, universal quantification was used to remove the recursive variables.

We can now combine the two pre-conditions together in order to obtain a single safety pre-condition for the recursive check, as shown here:

$$pre(L4) = preFst(L4) \wedge preRec(L4) = (h < l) \vee (0 \leq l + h \wedge 0 \leq l)$$

Through a similar derivation, the other check of $H4$, based on the pre-condition $l + h < 2a$ from *getMid*, yields:

$$pre(H4) = preFst(H4) \wedge preRec(H4) = (h < l) \vee (h < a \wedge l + h < 2a)$$

The derived pre-conditions are very precise. Apart from ensuring that the given recursive checks are safe, it also captures a condition on how the checks may be avoided.

4.3 Amalgamating Related Checks

As some of the checks are closely related, it may be useful to amalgamate[3] these checks together. At the risk of missing out some opportunities for optimization, the amalgamation of related checks serves two purposes, namely:

- It can cut down the time taken for our analysis.
- It can reduce the number of specialization points, and hence the size of the specialized code.

We propose a simple technique to identify related checks. Given two checks C_1 and C_2, we consider them to be related if either $C_1 \Rightarrow C_2$ or $C_2 \Rightarrow C_1$. For example, checks $H6$ and $H7$ are related since $chk(H6) \Rightarrow chk(H7)$. Because of this similarity, we can combine the pre-conditions of these two checks, as follows:

$$pre(H6, H7) = pre(H6) \wedge pre(H7) = i \leq -1 \vee (j < i \wedge 0 \leq i) \vee i < m$$

The combined pre-condition can eliminate both checks simultaneously.

4.4 Inter-procedural Propagation of Checks

To support inter-procedural propagation of checks, each pre-condition for a partially redundant check must first be converted into a new check at the call site. After that, the process of classifying the check and deriving its safety pre-condition is repeated.

Consider two functions:

$$\begin{array}{lll} f(v_1, .., v_n) & = & \ldots\ L@sub(arr, i) \ldots \\ g(w_1, .., w_n) & = & \ldots\ C@f(v'_1, .., v'_n) \ldots \end{array}$$

Suppose that in f, we have managed to derive a non-trivial $pre(L)$ that would make $chk(L)$ redundant. Then, at each call site of f, such as $f(v'_1, .., v'_n)$ in the body of function g, we should convert the pre-condition of f into a new check at the call site, as follows:

$$\begin{array}{lll} chk(C) & = & \exists\, X.\, pre(L) \wedge subs(C) \\ subs(C) & = & \wedge_{i=1}^{n}\ (eq\ \tau_i\ \tau'_i) \qquad \textbf{where}\ v_i :: \tau_i;\ v'_i :: \tau'_i;\ X = \cup_{i=1}^{n} fv(\tau_i) \end{array}$$

The pre-condition of f is converted into a check via a size parameter substitution, $subs(C)$.

Example : Consider a function q:

$$\begin{array}{l} q :: (\texttt{Arr}^n\ \texttt{Int}, \texttt{Int}^k) \rightarrow \texttt{Int}^s \\ q(arr, k)\ =\ \textbf{let}\ \ r = random(); l = k + 1\ \ \textbf{in}\ C8@C9@p(arr, r, l) \end{array}$$

[3] In general, any two checks can be amalgamated together. However, closely related checks will have a higher probability of being satisfied at the same time. This can help ensure amalgamation without loss of optimization.

At the labelled call site, we have:

$$subs(C8) = subs(C9) \quad \text{and} \quad subs(C9) = (m = n) \wedge (j = l) \wedge (i = r)$$

We assume that the size variables assigned to the arguments of the p call are n, r and l, respectively. Using our formula for converting the pre-condition of p into a check at its call site, we obtain:

$$\begin{aligned} chk(C8) &= \exists\, i, j.\, pre(L7) \wedge subs(C8) \\ &= (r \leq -1) \vee (l < r \wedge 0 \leq r) \vee (r \geq 1) \\ chk(C9) &= \exists\, i, j.\, pre(H6, H7) \wedge subs(C9) \\ &= (r \leq -1) \vee (l < r \wedge 0 \leq r) \vee (r < n) \end{aligned}$$

With this, we can propagate the check backwards across the procedure of q by deriving the following two pre-conditions.

$$\begin{aligned} pre(C8) &= \forall\, r, l. \neg\,(l = k + 1) \vee ((r \leq -1) \vee (l < r \wedge 0 \leq r) \vee (r \geq 1)) \\ &= k \leq -2 \\ pre(C9) &= \forall\, r, l. \neg\,(l = k + 1) \vee ((r \leq -1) \vee (l < r \wedge 0 \leq r) \vee (r < n)) \\ &= (k \leq -2) \vee (-1 \leq k \leq n - 2) \end{aligned}$$

Note that since r and l are local variables; we must eliminate them from our pre-condition by using universal quantification. Universal quantification ensures that we get a new pre-condition that is safe for all values of r and l.

Inter-procedural propagation of checks applies to recursive functions without exception.

Example : The pre-condition for *look* can be converted to checks at its call site in *bsearch*, as follows:

$$\begin{aligned} chk(L5) &= \exists\, l, h.\, pre(L4) \wedge subs(L5) \\ &= \exists\, l, h.((h < l) \vee (0 \leq l + h \wedge 0 \leq l)) \wedge (l = 0 \wedge h = v - 1) \\ &= (v \leq 0) \vee (1 \leq v) \\ chk(H5) &= \exists\, l, h.\, pre(H4) \wedge subs(L5) \\ &= \exists\, l, h.((h < l) \vee (h < a \wedge l + h < 2a)) \wedge (l = 0 \wedge h = v - 1) \\ &= (v \leq 0) \vee (v \leq a, 2a) \\ subs(L5) &= (l = 0) \wedge (h = v - 1) \end{aligned}$$

From here, we can derive the safety pre-conditions for *bsearch* as shown below.

$$\begin{aligned} pre(L5) &= \neg\, ctx(L5) \vee chk(L5) \\ &= \forall\, v. \neg(v = a \wedge a \geq 0) \vee (v \leq 0 \vee 1 \leq v) \\ &= True \\ pre(H5) &= \neg\, ctx(H5) \vee chk(H5) \\ &= \forall\, v. \neg(v = a \wedge a \geq 0) \vee (v \leq 0 \vee v \leq a, 2a) \\ &= True \end{aligned}$$

Through this inter-procedural propagation, we have successfully determined that the recursive checks of *look* inside *bsearch* are totally redundant. Hence, all bound checks for *bsearch* can be completely eliminated. This is done by providing specialized versions of *look* and *getmid* (without bound checks) that would be called from *bsearch*.

5 Bound Check Specialization

With the derived pre-condition for each partially redundant check, we can now proceed to eliminate *more* bound checks by specializing each call site with respect to its context. The apparatus for bound check specialization is essentially the same as contextual specialization [6, 16] where each function call can be specialized with respect to its context of use. A novelty of our method is the use of derived pre-condition to guide specialization. This approach is fully automatic and can give better reuse of specialized code.

Suppose that we have a function f with N checks, that is used in one of its parent function g as follows:

$$\begin{array}{lcl} f(v_1, .., v_n) & = & t_f \ \mathbf{Req} \ \{P_i\}_{i=1}^N \\ g(v_1, .., v_n) & = & \ldots \ \{C_i\}_{i=1}^N @ f(v_1', .., v_n') \ \ldots \end{array}$$

Notation-wise, we write $\{C_i\}_{i=1}^N @ f(v_1', .., v_n')$ as the short form for $C_1 @ \ \ldots \ @ C_N @ \ f(v_1', .., v_n')$.

Suppose further that we have a context $ctx(C)$, for the labelled call, which may encompass a context that could be inherited from the specialization of g. Let the set of pre-conditions whose checks could be made redundant be:

$$G = \{P_i | i \in 1 \ldots N \wedge ctx(C) \Rightarrow chk(C_i)\}$$

For maximal bound check optimization, we should specialize each of the call for f to a version that would maximize bound check elimination. In the above example, our specialization would introduce f_G, as follows:

$$\begin{array}{lcl} g(v_1, .., v_n) & = & \ldots \ f_G(v_1', .., v_n') \ \ldots \\ f_G(v_1, .., v_n) & = & \mathcal{S}[t_f] \ G \qquad \mathbf{where} \ \ ctxSta(f_G) = \ G \ \wedge \ ctxSta(f) \end{array}$$

Note how the context G, which contains the maximum pre-conditions that are satisfiable in $ctx(C)$, is propagated inside the body of f by specializer $\mathcal{S}$. This specialization is commonly known as *polyvariant* specialization. It will generate a specialized version of the code for each unique set of checks that can be eliminated. It can provide as many variants of the specialized codes as there are distinguishable contexts. To minimize the number of variants used, the specialization process will proceed top-down from the main function, and generate a specialized version only if it is required directly (or indirectly) by the main function. Polyvariant specialization can help maximize the elimination of bound checks. However, there is a potential explosion of code size, as the maximum number of specialized variants for each function is 2^N where N is the number of partially redundant checks that exist. In practice, such code explosion seldom occur, unless the function is heavily reused under different contexts.

If code size is a major issue (say for embedded systems), we could use either *monovariant* specialization or *duovariant* specialization.

In monovariant specialization, we will need an analysis technique to help identify the best common context, call it $ctxMin(f)$, that is satisfied by all the call sites. Let the set of call sites to f in a given program be:

$$\{\{C_{ij}\}_{i=1}^{N} @ f(v j_1, .., v j_n)\}_{j=1}^{M},$$

and their corresponding contexts be $\{ctx(C_j)\}_{j=1}^{M}$. We define the best common context of these call sites to be:

$$ctxMin(f) = \{P_i | i \in 1..N, \forall j \in 1..M. ctx(C_j) \Rightarrow chk(C_{ij})\}$$

With this most informative common context, we could now provide a least specialized variant for f that could be used by all call sites in our program, as follows:

$$\begin{aligned} f_{min}(v_1, .., v_n) &= \mathcal{S}[t_f]\ ctxMin(f) \qquad \textbf{where} \\ ctxSta(f_{min}) &= ctxMin(f) \ \wedge \ ctxSta(f) \end{aligned}$$

For duovariant specialization, we shall generate a version of each function f that is maximally specialized, namely:

$$\begin{aligned} f_{max}(v_1, .., v_n) &= \mathcal{S}[t_f]\ ctxMax(f) \qquad \textbf{where} \\ ctxSta(f_{max}) &= ctxMax(f) \ \wedge \ ctxSta(f) \\ ctxMax(f) &= \{P_i \mid i \in 1..N, \exists j \in 1..M.\ ctx(C_j) \Rightarrow chk(C_{ij})\} \end{aligned}$$

This most specialized variant should be used whenever possible. With the three variants of bound check specialization, we now have a spread of the classic space-time tradeoff. We hope to investigate the cost-effectiveness of these alternatives in the near future.

6 Performance Analysis

In this section, we address the practicality of using constraint solving for implementing both our forward analysis (essentially sized typing) and backward analysis (for deriving pre-conditions).

Our experiments were performed with Omega Library 1.10, running on a Sun System 450. We took our examples mostly from [26], with the exception of *sumarray* from [27]. The reported time measurements are the average values out of 50 runs. The first column reports the time taken by forward analysis (largely for computing invariants), while the second column reports the time taken for backward derivation of safety pre-condition.

The results shows that the time taken by the analyses required by array bound checks optimization are largely acceptable. A slightly higher analysis time was reported for *hanoi*, due largely to the more complex recursive invariant being synthesized.

Our analysis determines that all checks in these examples are totally redundant. Consequently, they are eliminated in the specialized codes. Gains in

	Forward	**Backward**
bcopy	0.03	0.21
binary search	0.54	0.07
bubble sort	0.05	0.31
dot product	0.03	0.21
hanoi	1.59	2.74
matrix mult	0.12	0.98
queens	0.19	0.53
sumarray	0.03	0.42

Fig. 7. Computation Time (in Secs) for Forward and Backward Analyses

run-time efficiency range between 8% (for "sumarray" program) and 56% ("matrix mult" program), which is comparable to those found in the literature (such as [26]).

7 Related Work

Traditionally, data flow analysis techniques have been employed to gather available information for the purpose of identifying redundant checks, and anticipatable information for the purpose of hoisting partially redundant checks to a more profitable location. The techniques employed have gradually increased in sophistication, from the use of family of checks in [15], to the use of difference constraints in [3]. While the efficiency of the techniques are not in question, data flow analysis techniques are inadequate for handling checks from recursive procedures, as deeper invariants are often required.

To handle checks from programs with more complex control flow, verification-based methods have also been advocated by Suzuki and Ishihata [24], Necula and Lee [19, 20] and Xu *et al* [27]; whilst Cousot and Halbwachs [5] have advocated the use of abstract interpretation techniques. Whilst powerful, these methods have so far been restricted to eliminating totally redundant checks.

It is interesting to note that the basic idea behind the backward derivation of weakest pre-condition was already present in the inductive iteration method, pioneered by Suzuki and Ishihata[24], and more recently improved by Xu *et al* [27]. However, the primary focus has been on finding totally redundant checks. Due to this focus, the backward analysis technique proposed in [24] actually gathers both pre-condition and contextual constraints together. Apart from missing out on partially redundant checks, their approach is less accurate than forward methods (such as [5]) since information on local variables are often lost in backward analysis.

Xi and Pfenning have advocated the use of dependent types for array bound check elimination[26]. While it is possible to specify pre-conditions through dependent types, they do not specially handle partially redundant checks. More-over, the onus for supplying suitable dependent types rest squarely on the programmers.

Recently, Rugina and Rinard [23] proposed an analysis method to synthesize symbolic bounds for recursive functions. In their method, every variable is expressed in terms of a lower and an upper symbolic bound. By assuming a polynomial form for the symbolic bounds, their method is able to compute these bounds *without* using fix-point iteration. In some sense, this inductive approach is similar to the proposal made in [25, 21], where size information is inductively captured to analyze program termination property. Whilst the efficiency of the inductive approach is not in question, we have yet to investigate the loss in precision that come with fixing the expected target form.

8 Conclusion and Future Work

Through a novel combination of both forward technique to compute contextual constraint, and backward method to derive weakest pre-conditions, we now have a comprehensive method for handling both totally redundant and partially redundant checks. Both analysis methods are built on top of a Presburger constraint solver that has been shown to be both accurate and practically efficient[22]. Our new approach is noteworthy in its superior handling of partially redundant checks.

There are several promising directions for future research. They deal largely with how the precision of optimization and efficiency of analysis method could be further improved.

Firstly, our contextual constraint analysis presently inherits its constraints largely from conditional branches. We can further improve its accuracy by propagating prior bounds checks in accordance with the flow of control. For this to work properly, we must be able to identify the weakest pre-conditions for each function that could be asserted as post-condition, after each call has been successfully executed. As bound errors could be caught by exception handling, the extent of check propagation would be limited to the scope where the bound errors are uncaught.

Secondly, not all partially redundant checks could be eliminated by its caller's context. Under this scenario, it may be profitable to insert speculative tests that could capitalize on the possibility that safety pre-condition are present at runtime. Whilst the idea of inserting speculative runtime test is simple to implement, two important issues that need to be investigated are (i) what test to insert, and (ii) where and when will it be profitable to insert the selected test. Specifically, we may strip out the avoidance condition from the speculative test, and restricts such runtime tests[4] to only recursive checks.

Lastly, the efficiency of our method should be carefully investigated. The cost-benefit tradeoff of check amalgamation and bound check specialization would need to be carefully studied in order to come up with a practically useful strategy.

[4] The insertion of speculative tests may look similar to check hoisting. The key different is that no exception is raised if speculative test fails.

Also, the sophistication (and cost) of our approach is affected by the type of constraints that is supported. Whilst Presburger formulae have been found to be both precise and efficient, it may still be useful to explore other types of constraint domains.

9 Acknowledgment

We would like to thank the anonymous referees for their valuable comments. This work has been supported by the research grants RP3982693 and RP3991623.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. F. Benoy and A. King. Inferring argument size relationships with CLP(R). In *Logic Programming Synthesis and Transformation*, Springer-Verlag, August 1997.
3. R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 321–333, 2000.
4. W.N. Chin and S.C. Khoo. Calculating sized types. In *2000 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–72, Boston, Massachusetts, United States, January 2000.
5. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of Programming Languages*, pages 84–96. ACM Press, 1978.
6. F. Fioravanti, A. Pettorossi, and M. Proietti. Rules and stratigies for contextual specialization of constraint logic programs. *Electronic Notes in Theoretical Computer Science*, 30(2), 1990.
7. R. Gupta. A fresh look at optimizing array bound checking. In *ACM SIGPLAN Conf. on Program Lang. Design and Impl.*, pages 272–282, New York, June 1990.
8. R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters Program Lang. Syst.*, 2(1-4):135–150, Mar-Dec 1994.
9. N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Pub., 1993.
10. N. Halbwachs. About synchronous programming and abstract interpretation. *Science of Computer Programming, Special Issue on SAS'94*, 31(1), May 1998.
11. N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.
12. W. H. Harrison. Compiler analysis for the value ranges for variables. *IEEE TOSE*, SE-3(3):243–250, May 1977.
13. P. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Version 1.1.0 Interface Guide. Technical report, University of Maryland, College Park, November 1996. `http://www.cs.umd.edu/projects/omega`.
14. A. King, K. Shen, and F. Benoy. Lower-bound time-complexity analysis of logic programs. In Jan Maluszynski, editor, *International Symposium on Logic Programming*, pages 261 – 276. MIT Press, November 1997.

15. 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.
16. L. Lafave and J. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In *Program Synthesis and Transformation, LOPSTR'97*, pages 70–82, LNCS 1463, 1997.
17. V. Markstein, J. Cooke, and P. Markstein. Optimization of range checking. In *ACM SIGPLAN Symp. on Compiler Construction*, pages 114–119, June 1982.
18. S.P. Midkiff, J.E. Moreira, and M. Snir. Optimizing bounds checking in Java programs. *IBM Systems Journal*, 37(3):409–453, 1998.
19. G. Necula. Proof-carrying code. In *ACM Principles of Programming Languages*, pages 106–119, 1997.
20. G Necula and P. Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 333–344, 1998.
21. L. Plumer. Termination proofs for logic programs. In *Lecture Notes in Artificial Intelligence*, volume 446. Springer Verlag, 1990.
22. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of ACM*, 8:102–114, 1992.
23. R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 182–195. ACM Press, June 2000.
24. N. Suzuki and K. Ishihata. Implementation of array bound checker. In *ACM Principles of Programming Languages*, pages 132–143, 1977.
25. J.D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *Journal of ACM*, 35(2):345–373, 1988.
26. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM Conference on Programming Language Design and Implementation*, pages 249–257. ACM Press, June 1998.
27. Z. Xu, B.P. Miller, and T. Reps. Safety checking of machine code. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 70–82. ACM Press, June 2000.