A Bounds Inference Method for Vector-Based Memoization

Wei-Ngan CHIN*
National University of Singapore
chinwn@iscs.nus.sg

Masami HAGIYA
University of Tokyo
hagiya@is.s.u-tokyo.ac.jp

Abstract

The dynamic-sized tabulation method can be used to eliminate redundant calls for certain classes of recursive programs. An innovative aspect of the method is the use of *lambda abstractions* that may subsequently be converted to *bounded vectors*, in order to share redundant calls via vector lookup.

To facilitate this conversion to vector form, we propose a new inference method to conservatively determine the bounds for arithmetic parameters of recursive functions. Suitable techniques for inferring the safe bounds of these parameters are introduced, together with supporting transformations. The resulting method can obtain efficient vector-based programs without the need for run-time bounds checking.

1 Introduction

Redundant calls frequently occur in recursive programs causing run-time inefficiency. Two examples are:

```
fib(n)     = case n of { 0 → 1 ; 1 → 1 ;
                         m+2 → fib(m+1)+fib(m) }
bin(n,k) = case n of { 0 → 1 ;
                         m+1 → if k≤0 or k≥n then 1
                                 else bin(m,k-1)+bin(m,k) }
```

Program transformation techniques can be used to eliminate some of these redundant calls. Where successfully applied, they can deliver asymptopic improvements to the time-complexity of these programs. A classic fold/unfold transformation tactic that could achieve this is called *tupling* [3, 14], with an automated version presented in [4]. This particular tactic is capable of eliminating the redundant calls of naive fibonacci definition via a *static-sized* tuple of two calls *(fib(m+1),fib(m))*, but it is unable to handle the binomial function which requires a *dynamic-sized* table of memoized results.

The difference between *fib* and *bin* can be seen in the first two call dependency graphs (DGs) in Fig. 1. While the width of the DG for *fib* is kept at two calls, the width of the DG for *bin* increases, the further it is from its root. As a result, dynamic-sized table is needed to compute the binomial function in a non-redundant fashion. In particular, each row of calls can only be computed, without redundancy, from a slightly larger row of calls down the recursion. For example, *bin(n,k)* requires the results of two calls, *bin(n-1,k-1),bin(n-1,k)*, which in turn require the results of three calls, *bin(n-2,k-2),bin(n-2,k-1),bin(n-2,k)*, and so on.

To widen the class of programs that could be tabulated, we previously proposed a transformation method [6] which uses two additional tactics, named: (i) *lambda abstraction* tactic, and (ii) *vector conversion* tactic.

The lambda abstraction tactic is used to lift out problematic accumulative parameters whose functions are rejected by the tupling tactic of [4]. A parameter is said to be *accumulative* if it syntactically becomes larger with successive recursive calls. An example is the second parameter of *bin*. Though this parameter's value decreases with successive recursive calls, its symbolic form actually becomes syntactically larger when the call of *bin* is repeatedly unfolded during program transformation. In contrast, the first parameter of *bin* shrinks or remains the same size (syntactically) with successive unfolding of the recursive calls during transformation. (We rely on pattern-matching *case* construct to ensure this non-accumulating property.)

Accumulative parameters could cause successive tuples to have increasing number of calls during transformation, as indicated by DGs with increasing width. This can cause tupling to be non-terminating - an effect similar to infinite specialisation in partial evaluation [10]. Our remedy is to lift out accumulating parameter(s) via the lambda abstraction tactic. This has the effect of *currying* the lifted parameter(s), as can be seen for *bin*, as outlined below.

```
; Apply Lambda Abstraction Tactic (to lift k)
bin(n,k) = let { bin'(n) = case n of { 0 → (\k → 1) ;
                   m+1 → (\k → if k≤0 or k≥n then 1
                                  else bin'(m)(k-1)+bin'(m)(k) ) } }
             in bin'(n)(k)
; Apply Tupling Tactic (to share bin'(m) call)
          = let { bin'(n) = case n of { 0 → (\k → 1) ;
                   m+1 → let z=bin'(m) in
                          (\k → if k≤0 or k≥n then 1
                                  else z(k-1)+z(k) ) } }
             in bin'(n)(k)
```

After lifting the accumulative *k* parameter, our new *bin'* function will be in a form that could be accepted by tupling, as evident from the fixed-width DG of *bin'* calls in Figure 1.

Note also that tupling simply shares the two function-type *bin'(m)* calls via a local *z* variable. However, unlike

*Part of this work was done while the first author was visiting Institute of Information Science, Academia Sinica, Taiwan, and Hitachi Advanced Research Laboratory, Japan.

In ACM SIGPLAN International Conference on Functional Programming, Amsterdam, Netherlands, June 1997, pg 176–187, ACM Press.

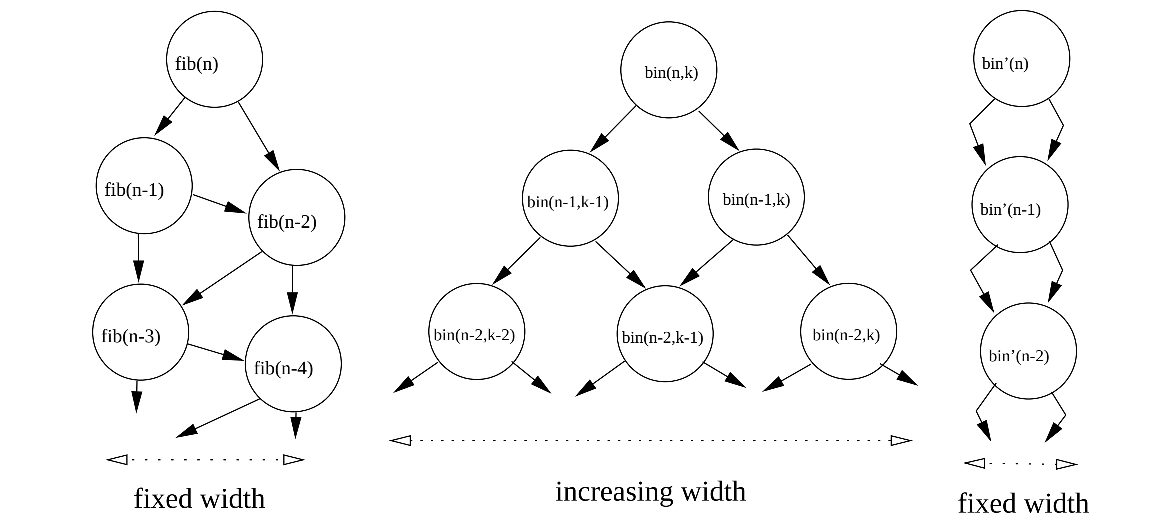

Figure 1: Dependency Graphs for *fib(n)*, *bin(n,k)* and *bin'(n)*.

ground-type calls, the sharing of function-type calls is only effective if their functions are being memoized. Given a function of type $A \rightarrow B$, we could either memoize its mapping via a list of type $[(A, B)]$, or via a vector of type $Array\ A\ B$. The list implementation is more general but suffers from a higher lookup overhead. Where possible, we advocate the use of the vector conversion tactic, to replace lambda abstraction by a bounded vector. A crucial requirement is that the lambda abstraction's parameter(s) must be of vector index type. A *vector index* is either an *enumerated* type or a *finitely bounded integer* type, or a tuple of such types. Such types have finite sets of ordered values which are suitable for use as indexes to bounded vectors.

The method for inferring *safe* parameter bounds was a missing puzzle in our previous work[6]. However, it is very crucial towards the automation of vector conversion. We propose a collection of systematic techniques to infer *safe* bounds for parameters of recursive functions. Formally:

Definition 1: *Safe Parameter Bounds*
The bounds for a parameter, a_i, of a recursive function, f, are said to be *safe* if the value for that parameter will never exceed its stated bounds, $lower \leq a_i \leq upper$, for each of its recursive calls, $f(a_1, .., a_i, .., a_n)$.

We are also interested in obtaining *tighter* parameter bounds, where possible. Formally:

Definition 2: *Tighter Parameter Bounds*
Given a safe bound $max\{lower_1, lower_2\} \leq a_i$, we say that $lower_1$ is tighter than $lower_2$ if $lower_2 \leq lower_1$ for all recursive calls encountered.

Similarly, given $a_i \leq min\{upper_1, upper_2\}$, we say that $upper_1$ is tighter than $upper_2$ if $upper_1 \leq upper_2$ for all recursive calls encountered.

With a little insight, one may propose the following set of *safe* parameter bounds for the *bin* function:

$$(0 \leq k^{\square} \leq n^{\square}) \textbf{ implies}$$
$$(0 \leq k \leq min\{n, k^{\square}\}) \wedge (k \leq n \leq n^{\square})$$

where $n^{\square}, k^{\square}$ refer to the argument values for the very first *bin* call, while $(0 \leq k^{\square} \leq n^{\square})$ is a required pre-condition for this call. Specifically, $(0 \leq k \leq n)$ came from the conditional test for recursive branches, while $(k \leq k^{\square}) \wedge (n \leq n^{\square})$ came from the strictly decreasing nature of the two parameters for successive recursive calls.

Our proposed bounds inference method can do even better. As we shall see, it can systematically deduce the following equally safe but somewhat *tighter* parameter bounds:

$$(0 \leq k^{\square}) \wedge (max\{1, k^{\square}\} \leq n^{\square}) \textbf{ implies}$$
$$(max\{0, n-(n^{\square}-k^{\square})\} \leq k \leq min\{n, k^{\square}\})$$
$$\wedge (max\{1, k\} \leq n \leq n^{\square}+k-k^{\square})$$

where $(0 \leq k^{\square}) \wedge (max\{1, k^{\square}\} \leq n^{\square})$ is its pre-condition. The bound $max\{0, n-(n^{\square}-k^{\square})\} \leq k \leq min\{n, k^{\square}\}$ gives a safe range for the k parameter in terms of the other parameters. It can be used, by vector conversion and related transformations, to obtain the following efficient $O(n \times k)$ definition. Note the adoption of Haskell's primitive functions, *array* and *(!)*, for array construction and indexing.

```
data Assoc a b = a := b ;
array :: Ix a => (a,a) → [Assoc a b] → Array a b ;
bin(n,k) = let { n0 = n ; k0 = k ;
                 bin'(n) = case n of { 0 → (\k → 1) ;
                 m+1 → let { z = bin'(m) ;
                             bn k = if k≤0 or k≥n then 1
                                    else z(k-1)+z(k) ;
                             l = max(0,n-(n0-k0)) ;
                             u = min(n,k0) ;
                             vec = array (l,u) [a := bn a
                                               | a ← [l..u]] }
                        in (!) vec } }
           in if (0 ≤ k0 ∧ max(1,k0) ≤ n0) then bin'(n0)(k0)
              else 1
```

Section 2 introduces a simple language and some preliminary notations for labelling self-recursive functions. Section 3 presents the inference techniques for determining the safe and tight bounds of arithmetic parameters of recursive functions. Section 4 and 5 introduce some transformation tactics which are needed to support vector conversion.

Section 6 contains some performance figures to justify the bounds inference method. Section 7 describes related work before a conclusion. Appendix A rehearses the bounds inference method on yet another example.

2 Language and Labelled Recursion

We consider a simple functional language:

Definition 3: *A Simple Language*
Components of our simple language include:

$$
\begin{array}{lll}
P & ::= [M_i]_{i=0}^{n} & \text{(Program)} \\
M & ::= \{f_i(v_1,\ldots,v_n) = tf_i\}_{i=0}^{m} & \text{(Mutual Rec. Set)} \\
A & ::= c_0 + \sum_{i\in N} c_i \times v_i^{\oplus} & \text{(Linear Arith. Term)} \\
B & ::= A_1 \le A_2 \mid A_1 < A_2 \mid A_1 \ge A_2 \mid A_1 > A_2 & \\
 & \quad \mid A_1 = A_2 \mid A_1 \ne A_2 \mid \neg B & \\
 & \quad \mid B_1 \wedge B_2 \mid B_1 \vee B_2 & \text{(Boolean Arith. Term)} \\
t & ::= v(t_1,\ldots,t_n) \mid C(t_1,\ldots,t_n) \mid f(t_1,\ldots,t_n) \mid & \\
 & \quad \mathit{let}\ \{p_i = t_i\}_{i=0}^{n}\ \mathit{in}\ t \mid \mathit{case}\ t\ \mathit{of}\ \{p_i \to t_i\}_{i=0}^{n} & \\
 & \quad \mid \backslash\ (v_1,\ldots,v_n) \to t \mid \mathit{if}\ B_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3\ ; & \\
 & & \text{(Expression)} \\
p & ::= v \mid C(p_1,\ldots,p_n)\ ; & \text{(Pattern)}
\end{array}
$$

The expressions allowed include variables (v), data constructors (C), integer constants (c), functions (f), *let* and *case* constructs and a special conditional with boolean arithmetic expression (B) which is formed from a class of arithmetic variables, marked as $v^{\oplus}$. The linear arithmetic terms are ideal for use in Presburger formulae. We shall abbreviate a sequence of terms $t_1,\ldots,t_n$ by $\vec{t}$, so that a function call $f(t_1,\ldots,t_n)$ can be abbreviated as $f(\vec{t}\,)$ or more precisely as $f(t_i)_{i\in 1..n}$.

We assume that the language's standard semantics is *non-strict* by default, with the exception of scalar-type parameters which are aggressively treated as *strict*. Such a compromise has been adopted in Imperial College's Hope language [13] which could give good performance without losing much on expressiveness. The main reason we adopt this semantics is that the vector conversion tactic will force parameters that are converted to vector indexes, to be *strict*. Alternatively, if a truly non-strict semantics is adopted, we must use a strictness analyser to ensure that parameters are strict before they are selected by the vector conversion tactic.

Our language is a near first-order language, with some residual higher-order features. Programs of this restricted form can be obtained via the higher-order removal method proposed in [5]. This method can remove *most* higher-order functions by either *function specialisation* or *eta-abstraction* until programs of the above form remains. The residual higher-order features are either *inherently higher-order* or have *complex control dependencies* that may be ignored by our analysis. Adopting this restricted language considerably simplifies our presentation, without much loss of generality. In particular, we can avoid the more complex control dependencies present in certain higher-order programs.

For simplicity, our bounds inference method shall be formulated for self-recursive functions, defined below.

Definition 4: *Self-Recursive Functions*
A function, f, is said to be *self-recursive* if its (mutual) recursive set of functions is just a singleton set $\{f\}$. In particular, each self-recursive function definition will have the following form where $m \ge 1$.

$$M \equiv \{f(\vec{v}) = \widehat{e}\langle f(\vec{t_1}),..,f(\vec{t_m})\rangle\}$$

Note the use of a special context notation, $\widehat{e}\langle t_1,\ldots,t_m\rangle$, with m holes for sub-terms, $t_1,\ldots,t_m$. Though not described in this paper, extension to mutual-recursive functions is possible.

We classify the variables (both parameters and local variables) of our functions into two groups, namely:

- **Arithmetic variables:**
  An integer-type variable is said to be arithmetic if it can be expressed as $v^{\oplus} = c_0 + \sum_{i\in N} c_i \times v_i^{\oplus}$ where $\{v_i^{\oplus}\}_{i\in N}$ are other arithmetic variables. An integer-type parameter is said to be arithmetic if its change across successive recursive calls can be expressed as above. For clarity, we shall use the $\oplus$ annotation to mark out arithmetic terms and/or variables.

- **Non-Arithmetic variables:**
  These are variables that are not *arithmetic* in form (e.g. non-integer variables or integer ones that are not expressed in linear arithmetic form.).

It is straightforward to formulate an analysis to distinguish arithmetic variables from non-arithmetic ones. Due to space consideration, we omit its presentation in the present paper.

To help formalise bounds inference for arithmetic parameters, we introduce a labelling scheme to uniquely identify successive recursive calls that may arise from a given function.

Definition 5: *Labelled Recursion Scheme*
Given a self-recursive function f, we shall label each of its recursive calls uniquely as $f^L(\vec{a}^L)$ where L is the path of the call from the root of its dependency graph (DG). The first recursive call at the root of the DG will be labelled as $f^{[]}(\vec{a}^{[]})$. This initial call is also referred to as the *root* call of f. Its immediate m child recursive calls will be labelled as $\{f^{[i]}(\vec{a}^{[i]})\}_{i\in 1..m}$. Subsequently, each function call $f^L(\vec{a}^L)$ may have upto m child recursive calls that are labelled as $\{f^{i:L}(\vec{a}^{i:L})\}_{i\in 1..m}$.

The superscripted suffix, L, given to each recursive call (and its parameters) identifies the position of the recursive call within its DG. Also, the recursion level of the call (distance from the root of DG) is $\#L$ where $\#$ finds the length of a sequence. In accordance with our labelling scheme, $a^{1:L}$ and $a^{2:L}$ denote two different argument values, while $f^{1:L}$ and $f^{2:L}$ denote two different call instances of the same function.

For example, the root call of *bin* is denoted by $bin^{[]}(n^{[]},k^{[]})$. Its two recursive calls (children) in its DG are denoted by $bin^{[1]}(n^{[1]},k^{[1]})$ and $bin^{[2]}(n^{[2]},k^{[2]})$. This pair of calls in turn can result in four more recursive calls, namely:

$$\{bin^{[1,1]}(n^{[1,1]},k^{[1,1]}),\ bin^{[2,1]}(n^{[2,1]},k^{[2,1]})\}\ \text{and}$$
$$\{bin^{[1,2]}(n^{[1,2]},k^{[1,2]}),\ bin^{[2,2]}(n^{[2,2]},k^{[2,2]})\}$$

In general, each $bin^L(n^L,k^L)$ can result in two recursive calls labelled as $bin^{1:L}(n^{1:L},k^{1:L})$ and $bin^{2:L}(n^{2:L},k^{2:L})$.

Based on this labelling scheme, we can annotate, in a general way, the parameters and recursive calls for the *bin* function definition, as follows.

$$
\begin{array}{l}
bin^L(n^L,k^L) = \mathit{case}\ n^L\ \mathit{of}\ \{\ 0 \to 1; \\
\qquad\qquad m+1 \to \mathit{if}\ k^L \le 0\ \text{or}\ k^L \ge n^L\ \mathit{then}\ 1 \\
\qquad\qquad\qquad \mathit{else}\ bin^{1:L}(m,k^L-1) + bin^{2:L}(m,k^L)\}
\end{array}
$$

Notice that the parameters of bin^L function call are named n^L and k^L. Also, there are two recursive calls which can be written as:

$$bin^{1:L}(n^{1:L}, k^{1:L}) \equiv bin^{1:L}(m, k^L - 1) \text{ and }$$
$$bin^{2:L}(n^{2:L}, k^{2:L}) \equiv bin^{1:L}(m, k^L)$$

since they are the children of the $bin^L(n^L, k^L)$ call. Hence, the above two recursive calls have argument values:

$$\{n^{1:L} = m = n^L - 1, k^{1:L} = k^L - 1\} \text{ and }$$
$$\{n^{2:L} = m = n^L - 1, k^{2:L} = k^L\}, \text{ respectively.}$$

3 Bounds Inference Method

We shall now present a bounds inference method which is made up of four main techniques:

Procedure 1: *Bounds Inference Method*
To infer the safe bounds of arithmetic parameters, we perform the following:

(1) Determine a *backward parameter substitution* for each recursive call.

(2) Determine an *individual constraint* for each recursive call.

(3) Determine a *unified constraint* for all recursive calls.

(4) Determine a *monotonic constraint* for each monotonically decreasing (or increasing) arithmetic parameter, as well as for groups of such parameters.

The four techniques together are aimed at accumulating safe bound constraints on the arithmetic parameters of self-recursive functions. The inference techniques are presented next, using *bin* as an example.

3.1 Backwards Parameter Substitutions

Consider a recursive function f with arithmetic parameters $\{a_j^L\}_{j\in N}$. For each recursive call of $\{f^{i:L}(\vec{a}^{i:L})\}_{i\in 1..m}$, we can obtain a set of *forward* substitutions (for arithmetic arguments $\vec{a}^{i:L}$) of the form:

$$F_f(i:L) \equiv \{a_j^{i:L} = c_{i0} + (\textstyle\sum_{k\in N} a_k^L \times c_{ik})\}_{j\in N}$$

By solving the linear equations of the above forward substitutions, we can obtain *backward* substitutions of the form $\{a_j^L = c'_{i0} + (\sum_{k\in N} a_k^{i:L} \times c'_{ik})\}_{j\in N}$. We shall denote such a backward parameter substitution by $B_f(i:L)$ for each recursive call, $f^{i:L}(\vec{a}^{i:L})$.

For the *bin* function, the forward substitutions are:

$$\begin{aligned} F_{bin}(1:L) &\equiv \{n^{1:L} = n^L - 1, k^{1:L} = k^L - 1\} \\ F_{bin}(2:L) &\equiv \{n^{2:L} = n^L - 1, k^{2:L} = k^L\} \end{aligned}$$

which are obtained from the two recursive calls of $bin^{1:L}$ and $bin^{2:L}$, respectively. After solving the two sets of equations, we obtain backward substitutions:

$$\begin{aligned} B_{bin}(1:L) &\equiv \{n^L = n^{1:L} + 1, k^L = k^{1:L} + 1\} \\ B_{bin}(2:L) &\equiv \{n^L = n^{2:L} + 1, k^L = k^{2:L}\} \end{aligned}$$

3.2 Individual Constraints

The second technique is to determine constraints[1] of the form $\bigwedge_{j\in N} lower_j \le a_j^{i:L} \le upper_j$ for each of the recursive calls, $\{f^{i:L}(\vec{a}^{i:L})\}_{i\in 1..m}$. Our technique consists of two steps.

Consider a function f with m self-recursive calls:

$$f^L(\vec{a}^L) = \widehat{e_f}\langle f^{1:L}(\vec{a}^{1:L}), .., f^{m:L}(\vec{a}^{m:L})\rangle \;;$$

In the first step, we determine a valid constraint, called the *context constraint*, for each of the self-recursive calls, $\{f^{i:L}(\vec{a}^{i:L})\}_{i\in 1..m}$. Context constraints are based on the particular contexts of individual self-recursive calls. We denote the context contraint of call $f^{i:L}(\vec{a}^{i:L})$ by $C_f(i:L)$. Each of these context constraints is made up of a conjunction of boolean arithmetic terms on the arithmetic parameters of $\vec{a}^L$. (Non-arithmetic parameters are ignored.) The inference rules to gather the context constraints are given in Figure 2. Apart from gathering the context constraint of form $C_f(L) \equiv B$ for each recursive call (marked by f^L), our rules also gather relevant contexts of form $C(v) \equiv B$ for each variable, v. This is to allow context constraints to be inferred also for arguments of function calls and *let* abstractions. No fix-point iteration is necessary since our inference is not performed beyond mutual-recursive function calls. Note how rule (4) simply glosses over its mutual-recursive call, while rule (5) attempts to infer beyond the non mutual-recursive call it is handling.

Consider the bin^L function definition. Based on the given inference rules, we can deduce that $n^L = 0$ is a valid context for the first *case* branch; while $(n^L = m + 1) \wedge (m \ge 0)$ is a valid context for the second branch. Under the context of the else branch (of the *if* conditional), we also have $(k^L > 0) \wedge (k^L < n^L)$. As the two recursive calls (of $bin^{1:L}$ and $bin^{2:L}$) lie inside the same context, we have the following context constraints, expressed in terms of the LHS arithmetic parameters, k^L, n^L.

$$\begin{aligned} C_{bin}(1:L) &\equiv C_{bin}(2:L) \\ &\equiv (n^L = m+1) \wedge (m \ge 0) \\ &\quad \wedge (k^L > 0) \wedge (k^L < n^L) \end{aligned}$$

In order to proceed further, we must simplify and normalise the boolean arithmetic terms. Initially, the local arithmetic variables are eliminated via their substitutions. After this, the normalisation rules (modulo associative and commutative properties of $+/\times$) in Figure 3 are exhaustively applied until the boolean arithmetic terms are in a canonical form. Specifically, the above context constraint normalises to:

$$\begin{aligned} C_{bin}(1:L) &\equiv C_{bin}(2:L) \\ &\equiv (-n^L + 1 \le 0) \wedge (-k^L + 1 \le 0) \\ &\quad \wedge (k^L - n^L + 1 \le 0) \end{aligned}$$

In the second step, we re-express the context constraint in terms of arithmetic parameters, $\vec{a}^{i:L}$, of the corresponding recursive call, $f^{i:L}(\vec{a}^{i:L})$. This results in *individual call constraint*, denoted by $I_f(i:L)$. Formally:

$$I_f(i:L) \equiv C_f(i:L)\, B_f(i:L)$$

where $C_f(i:L)$ is the context constraint for $f^{i:L}(\vec{a}^{i:L})$, and $B_f(i:L)$ is its backward parameter substitution. In the case

[1] Constraints are increasing used by language researchers in their attempt at making further advances in program analyses and optimization techniques. For example, Takano [16] advocated their use in the generalized partial computation method to help achieve a greater level of program specialization.

$$
\begin{array}{lll}
(0) & \phi_P & = [f \mapsto \mathcal{A}_C[\![tf]\!]\ True \mid f(v_1,\ldots,v_1) = tf \in P] \\
(1) & \mathcal{A}_C[\![v(t_1,\ldots,t_n)]\!]\ B & = \{C(v) \equiv B\} \cup \bigcup_{i=1}^{n} \mathcal{A}_C[\![t_i]\!]\ B \\
(2) & \mathcal{A}_C[\![C(t_1,\ldots,t_n)]\!]\ B & = \bigcup_{i=1}^{n} \mathcal{A}_C[\![t_i]\!]\ B \\
(3) & \mathcal{A}_C[\![(\backslash(v_1,\ldots,v_n) \to t)]\!]\ B & = \mathcal{A}_C[\![t]\!]\ B \\
(4) & \mathcal{A}_C[\![f^L(t_1,\ldots,t_n)]\!]\ B & = \{C_f(L) \equiv B\} \cup \bigcup_{i=1}^{n} \mathcal{A}_C[\![t_i]\!]\ B \\
(5) & \mathcal{A}_C[\![f(t_1,\ldots,t_n)]\!]\ B & = \bigcup_{i=1}^{n} \mathcal{A}_C[\![t_i]\!]\ (B \wedge \bigvee_{p \in Var_i} p) \\
 & & \text{where } Var_i = \{p \mid (C(v_i) \equiv p) \in (\phi_P[f])\}[t_i/v_i]_{i \in 1..n} \\
 & & \qquad f(v_1,\ldots,v_n) = tf \in P \\
(6) & \mathcal{A}_C[\![\mathit{if}\ b^{\oplus}\ \mathit{then}\ t_1\ \mathit{else}\ t_2]\!]\ B & = \mathcal{A}_C[\![t_1]\!]\ (B \wedge b) \cup \mathcal{A}_C[\![t_2]\!]\ (B \wedge \neg b) \\
(7) & \mathcal{A}_C[\![\mathit{case}\ b^{\oplus}\ \mathit{of}\ \{p_i \to t_i\}_{i \in N}]\!]\ B & = \bigcup_{i \in N} \mathcal{A}_C[\![t_i]\!]\ (B \wedge (b = p_i) \wedge (\forall v \in p_i.\ 0 \le v)) \\
(8) & \mathcal{A}_C[\![\mathit{case}\ t\ \mathit{of}\ \{p_i \to t_i\}_{i \in N}]\!]\ B & = \mathcal{A}_C[\![t]\!]\ B \cup \bigcup_{i \in N} \mathcal{A}_C[\![t_i]\!]\ B \\
(9) & \mathcal{A}_C[\![\mathit{let}\ \{v_i = t_i\}_{i \in N}\ \mathit{in}\ t]\!]\ B & = \mathcal{A}_C[\![t]\!]\ B' \cup \bigcup_{i \in N'} \mathcal{A}_C[\![t_i]\!]\ (B' \wedge \bigvee_{p \in Var_i} p) \\
 & & \text{where } N' = N - M \\
 & & \qquad M = \{i \mid i \in N \wedge IsArith(t_i)\} \\
 & & \qquad B' = B \wedge (\bigwedge_{i \in M} v_i = t_i) \\
 & & \qquad Var_i = \{p \mid (C(v_i) \equiv p) \in L\}\ \forall i \in N' \\
 & & \qquad L = \mathcal{A}_C[\![t]\!]\ True \cup \bigcup_{i \in M} \mathcal{A}_C[\![t_i]\!]\ True
\end{array}
$$

Figure 2: Context Constraints Inference for Program P

of *bin* function, we can obtain the following two individual call constraints:

$$
\begin{array}{lll}
I_{bin}(1:L) & \equiv & (-n^{1:L} \le 0) \wedge (-k^{1:L} \le 0) \\
 & & \wedge\, (k^{1:L} + 1 - n^{1:L} \le 0) \\
I_{bin}(2:L) & \equiv & (-n^{2:L} \le 0) \wedge (-k^{2:L} + 1 \le 0) \\
 & & \wedge\, (k^{2:L} - n^{2:L} \le 0)
\end{array}
$$

Using Bledsoe's SUP-INF procedure [2], we can project the above two constraints as bounds on the k and n parameters, giving:

$$
\begin{array}{lll}
I_{bin}(1:L) & \equiv & (max\{1, k^{1:L}+1\} \le n^{1:L}) \\
 & & \wedge\, (0 \le k^{1:L} \le n^{1:L} - 1) \\
I_{bin}(2:L) & \equiv & (max\{1, k^{2:L}\} \le n^{2:L}) \wedge (1 \le k^{2:L} \le n^{2:L})
\end{array}
$$

As the bounds constraints are specified over the respective parameters of each recursive call, we can omit the suffixes given to the parameters, without any ambiguity. Hence, we have:

$$
\begin{array}{lll}
I_{bin}(1:L) & \equiv & (max\{1, k+1\} \le n) \wedge (0 \le k \le n-1) \\
I_{bin}(2:L) & \equiv & (max\{1, k\} \le n) \wedge (1 \le k \le n)
\end{array}
$$

The SUP-INF procedure has to be enhanced to obtain bounds that are as tight as possible. A set of tight bounds checking rules, given in Figure 4, allows arithmetic bounds to be ordered, where possible.

3.3 Unified Constraints

With the help of *individual call constraints*, we can proceed to obtain a *unified constraint*, denoted by U_f, that would be valid for all self-recursive calls, namely:

$$\forall f^L.\ U_f \text{ holds for } \vec{a}^L$$

This unified constraint must be safe and should be as tight as possible. Specifically, we must show that it holds for *all* the recursive calls of f, with an exception for the root call. For the root call, $f^{[]}(\vec{a}^{[]})$, we can always use a *pre-condition* to guarantee that the unified constraint holds for this call. We denote such a pre-condition by $U_f^{[]} \equiv (U_f\ B_f([]))$ where $B_f([]) \equiv \{\vec{a} = \vec{a}^{[]}\}$ and $\vec{a}^{[]}$ denotes the arithmetic parameters of the root call.

For the non-root f calls, we propose two approaches to obtain *unified constraints*, that are as tight as possible. They are (i) **deductive approach**, and (ii) **inductive approach**. Both approaches use the same set of inference rules to unify bounds constraints, given in Figure 5. (Note that $\uparrow$ and $\downarrow$ are binary operators for finding maximum and minimum values, respectively.) In addition, the inductive approach is allowed to use an induction hypothesis in its assumption set. However, this hypothesis would have to be predicted in advance. We shall propose a simple but effective prediction strategy later in Section 3.3.2.

3.3.1 Unified Constraint by Deduction

One way of obtaining a unified constraint, which holds for all non-root calls, is to deduce it directly from the *individual call constraints*. Formally, we have:

$$\forall L. \forall i \in 1..m.\ I_f(i:L) \to U_f$$

The above proposition is equivalent to:

$$\forall L.\ \bigvee_{i \in 1..m} I_f(i:L) \to U_f$$

Hence, to infer an appropriate unified constraint, U_f, we can apply deduction steps to the disjunction $\bigvee_{i \in 1..m} I_f(i:L)$. Our deduction attempts to obtain unified constraints that are as tight as possible. In the case of the *bin* function, we can apply deduction to its individual call constraints, as shown below:

$$(A_1 > A_2) \Rightarrow (A_2 < A_1) \qquad (A_1 < A_2) \Rightarrow (A_1 + 1 \leq A_2) \qquad (A_1 \geq A_2) \Rightarrow (A_2 \leq A_1)$$

$$\neg(A_1 = A_2) \Rightarrow (A_1 \neq A_2) \qquad \neg(A_1 \leq A_2) \Rightarrow (A_1 > A_2) \qquad \neg(A_1 \neq A_2) \Rightarrow (A_1 = A_2)$$

$$\neg(B_1 \wedge B2) \Rightarrow (\neg B_1) \vee (\neg B_2) \qquad \neg(B_1 \vee B2) \Rightarrow (\neg B_1) \wedge (\neg B_2) \qquad \neg(\neg B) \Rightarrow B$$

$$A_1 \neq A_2 \Rightarrow (A_1 < A_2) \vee (A_1 > A_2) \qquad B \wedge (\bigvee_{i \in M} B_i) \Rightarrow \bigvee_{i \in M} (B \wedge B_i)$$

$$(A_1 \leq A_2) \Rightarrow (A_1 + (-1 \times A_2)) \leq 0 \qquad (A_1 = A_2) \Rightarrow (A_1 + (-1 \times A_2)) = 0$$

$$A + c_1 \times v + c_2 \times v \Rightarrow A + (c_1 + c_2) \times v \qquad A + 0 \times v \Rightarrow A$$

$$A + c \times (\textstyle\sum_{i \in M} c_i \times v_i) \Rightarrow A + (\textstyle\sum_{i \in M} (c \times c_i) \times v_i)$$

Figure 3: Normalisation Rules for (Boolean) Arithmetic Terms

$$-\infty \leq A \qquad A \leq -\infty \qquad \frac{(A_1 \leq A_2) \wedge (A_2 \leq A_3)}{A_1 \leq A_3}$$

$$\frac{(c_1 \times v) + A_1 \leq (c_2 \times v) + A_2}{A_1 \leq ((c_2 - c_1) \times v) + A_2} \qquad \frac{(0 \leq A_1) \wedge (0 \leq A_2)}{0 \leq A_1 + A_2}$$

Figure 4: Tight Bounds Checking Rules

$$\begin{array}{l}
\forall i \in 1..2.\, I_{bin}(i : L) \\
\quad \vdash \bigvee_{i \in 1..2} I_{bin}(i : L) \\
\quad \vdash (max\{1, k+1\} \leq n) \wedge (0 \leq k \leq (n-1)) \\
\qquad \vee (max\{1, k\} \leq n) \wedge (1 \leq k \leq n) \\
\quad \vdash (min\{max\{1, k+1\}, max\{1, k\}\} \leq n) \\
\qquad \wedge (min\{0, 1\} \leq k \leq max\{n-1, n\}) \\
\quad \vdash (max\{1, k\} \leq n) \wedge (0 \leq k \leq n)
\end{array}$$

This results in the following unified constraint for the two arithmetic parameters of *bin*.

$$U_{bin} \equiv (max\{1, k\} \leq n) \wedge (0 \leq k \leq n)$$

Effectively, this set of constraints has been shown to hold for all non-root calls of *bin*. Also, to cover the root call, we can simply add a pre-condition (suitably optimised) to the unified constraint, as follows:

$$\begin{array}{lll}
U_{bin} & \equiv & (max\{1, k^{\Box}\} \leq n^{\Box}) \wedge (0 \leq k^{\Box}) \\
& & \quad \textbf{implies } (max\{1, k\} \leq n) \wedge (0 \leq k \leq n)
\end{array}$$

3.3.2 Unified Constraint by Induction

Occasionally, the deductive approach may fail to obtain tight bounds for the unified constraint. In this case, we can resort to an inductive approach, based on the following formula:

$$\forall L. \forall i. (U_f \; B_f(i : L)) \; \wedge \; I_f(i : L) \rightarrow U_f$$

This formula assumes that U_f holds for a prior recursive call $f^L(\vec{a}^L)$, and is only valid when the same constraint holds for all subsequent recursive calls $\{f^{i:L}(\vec{a}^{i:L})\}_{i \in 1..m}$. Such use of an induction hypothesis can provide additional power for our inference method. However, one may justifiably think that it is non-trivial to predict plausible conjuncts that could be used as induction hypothesis of U_f.

To rectify this difficulty, we propose a unique strategy that *combines* both the deductive and inductive approaches. We begin with the deductive approach to find suitable bounds for the unified constraint. Some of these bounds will be immediately accepted since they hold for all the recursive calls via deduction. Other bounds may be contradictory and can be immediately rejected. Between these two extremes, we may find bounds that are not accepted by the deductive approach but could be candidates for the inductive approach.

In the case of *bin* function, all the bounds obtained via deductions could be immediately accepted for the unified constraint. To better highlight the inductive approach, consider another classic problem : Given *n* items of positive weights and a knapsack which can be filled with selected items up to a total weight of *w*, maximise the value of the knapsack. The maximum value for this knapsack can be specified in the following definition where *weight(i)* and *value(i)* are constant expressions[2] for the weight and value of the *i*-th item.

$$\begin{array}{l}
\textit{knap}(n^L, w^L) = \\
\quad \textit{case } n^L \textit{ of } \{ \; 0 \rightarrow 0 \; ; \\
\quad m+1 \rightarrow \textit{if } w^L < \textit{weight}(n^L) \textit{ then } \textit{knap}^{1:L}(m, w^L) \\
\qquad\qquad \textit{else } \textit{max}(\textit{knap}^{2:L}(m, w^L), \\
\qquad\qquad\quad \textit{knap}^{3:L}(m, w^L\text{-}\textit{weight}(n^L)) + \textit{value}(n^L)) \; \}
\end{array}$$

The backward parameter substitutions for the three recursive calls are:

[2] Such expressions with unknown constants actually violate the Presburger arithmetic form. In order to handle them correctly, we shall treat them as unique (arithmetic) variables whose bounds need not be inferred.

$$\bigwedge_{i \in N} B_i \quad \vdash \quad (-\infty \le v \le \infty) \wedge \bigwedge_{i \in N} B_i$$

$$\bigvee_{i \in N}((l_i \le v \le u_i) \wedge \bigwedge_{j \in M} B_{ij}) \quad \vdash \quad (min\{l_i\}_{i \in N} \le v \le max\{u_i\}_{i \in N}) \wedge (\bigvee_{i \in N} \bigwedge_{j \in M} B_{ij}))$$

$$(\forall i \in N.\, l \le l_i) \wedge (min\{l_i \uparrow L_i\}_{i \in N} \le v) \quad \vdash \quad (l \uparrow min\{L_i\}_{i \in N}) \le v$$

$$(\forall i \in N.\, u \ge u_i) \wedge (v \le (max\{u_i \downarrow U_i\}_{i \in N}) \quad \vdash \quad v \le (u \downarrow min\{U_i\}_{i \in N})$$

Figure 5: Deductive Rules to obtain Unified Constraints

$$\begin{aligned} B_{knap}(1:L) &\equiv \{n^L = n^{1:L} + 1, w^L = w^{1:L}\} \\ B_{knap}(2:L) &\equiv \{n^L = n^{2:L} + 1, w^L = w^{2:L}\} \\ B_{knap}(3:L) &\equiv \{n^L = n^{3:L} + 1 \\ &\qquad , w^L = w^{3:L} + weight(n^{3:L} + 1)\} \end{aligned}$$

Using the second inference technique, we can obtain the following individual call constraints:

$$\begin{aligned} I_{knap}(1:L) &\equiv (0 \le n) \wedge (w \le weight(n+1) - 1) \\ I_{knap}(2:L) &\equiv (0 \le n) \wedge (0 \le w) \\ I_{knap}(3:L) &\equiv (0 \le n) \wedge (weight(n+1) \le w) \end{aligned}$$

An attempt to obtain a unified constraint via deduction results in the following:

$$\begin{aligned} &\bigvee_{i \in 1..3} I_{knap}(i:L) \\ &\quad \vdash (min\{0,0,0\} \le n) \wedge min\{-\infty, 0, weight(n+1)\} \\ &\qquad\qquad \le w \le max\{weight(n+1) - 1, \infty\} \\ &\quad \vdash (0 \le n) \wedge -\infty \le w \le \infty \end{aligned}$$

In the above deduction, the conjunct $0 \le n$ was immediately accepted. However, the bound $-\infty \le w \le \infty$ is useless since its condition is true for all parameters without analysis. Nevertheless, during the above deduction, there were some (disjunctive) bounds on the parameter w (from individual call constraints) which might be accepted if a more powerful induction mechanism is available.

The three disjunctive bounds are:

1. $(0 \le w)$
2. $(weight(n+1) \le w)$
3. $(w \le weight(n+1) - 1)$

The second and third bounds contradict each other and should be immediately rejected. This leaves $(0 \le w)$ as a possible bound for the inductive approach. The procedure for accepting plausible inductive bounds is as follows:

Assume that P_C is a plausible inductive bound. Apply the following deduction to obtain U_f:

$$\forall L. \forall i. (P_C\ B_f(i:L)) \ \wedge\ I_f(i:L) \rightarrow U_f$$

If it can be shown that $U_f \rightarrow P_C$ holds, then P_C is an acceptable inductive bound for the unified constraint. In the case of *knap* function, we add $(0 \le w)$ to the induction hypothesis, and re-apply the SUP-INF deduction procedure, as follows:

$$\begin{aligned} &\bigvee_{i \in 1..3} ((0 \le w^L)\ B_{knap}(i:L)) \wedge\ I_{knap}(i:L) \\ &\quad \vdash (0 \le n) \wedge (0 \le w \le (weight(n+1) - 1)) \vee \\ &\qquad\quad (0 \le n) \wedge (0 \le w) \vee \\ &\qquad\quad (0 \le n) \wedge (max\{0, weight(n+1)\} \le w) \\ &\quad \vdash (min\{0,0,0\} \le n) \wedge min\{0, 0, max\{0, weight(n+1)\}\} \\ &\qquad\qquad \le w \le max\{weight(n+1) - 1, \infty\} \\ &\quad \vdash (0 \le n) \wedge (0 \le w \le \infty) \end{aligned}$$

The presence of $(0 \le w)$ in the final result confirms that it is an acceptable inductive bound for U_f. By adding a pre-condition on the root call, we get the following unified constraint for *all* recursive calls of *knap*.

$$U_{knap} \equiv (0 \le n^{[]}) \wedge (0 \le w^{[]})\ \textbf{implies}\ (0 \le n) \wedge (0 \le w)$$

3.4 Monotonic Constraints

Another source of bounds constraints, called *monotonic constraints*, can be obtained from arithmetic parameters that are monotonically increasing (or decreasing). Formally:

Definition 6: *Monotonic Parameters*
A parameter, a_j, is said to be a *monotonic* parameter if in each self-recursive call $f^{i:L}(a_1^{i:L}, .., a_j^{i:L}, .., a_n^{i:L})$, we have $a_j^{i:L} = a_j^L + e_{ij}$ where e_{ij} is an expression whose sign (positive, negative or zero) is determinable. This parameter is said to be monotonically *increasing* if $\forall i.\ e_{ij} \ge 0$; or monotonically *decreasing* if $\forall i.\ e_{ij} \le 0$.

The monotonic constraint for f, denoted by M_f, must hold for all calls of $f^L(\vec{a}^L)$. This is made up of a conjunction of inequalities of the form:

- $(a \le a^{[]})$ for monotonically decreasing parameter, a.
- $(a \ge a^{[]})$ for monotonically increasing parameter, a.
- $(k_1 \times a_1 + k_2 \times a_2 \le k_1 \times a_1^{[]} + k_2 \times a_2^{[]})$ for an arithmetic expression that is monotonically decreasing, with monotonic parameters (or expressions) a_1, a_2 and constants k_1, k_2.
- $(k_1 \times a_1 + k_2 \times a_2 \ge k_1 \times a_1^{[]} + k_2 \times a_2^{[]})$ for an arithmetic expression that is monotonically increasing, with monotonic parameters (or expressions) a_1, a_2 and constants k_1, k_2.

To illustrate the above classifications, consider the following self-recursive function.

$$g(x,y) = \widehat{e_g}\langle g(x-2, y-1), g(x-1, y+1)\rangle\ ;$$

Function g has two *monotonic* parameters x and y. Parameter x is monotonically decreasing, but parameter y is neither monotonically decreasing nor increasing. Nevertheless, a linear arithmetic expression $x + y$ is monotonically decreasing, while $-x + 2y$ is monotonically increasing. Such relationships can help us obtain new monotonic constraints for our parameters.

Monotonic constraints should, of course, be shown to hold for all recursive calls. A general inductive formula for this purpose is shown below.

Base : $(M_f\ B_f([]))$
Induction : $\forall L. \forall i \in 1..m. (M_f\ B_f(i:L)) \wedge A_f(i:L) \to M_f$

where $A_f(i:L)$ is an assumption set (of constraints) that is valid for the $f^{i:L}$ call. To simplify matters, we can just use $A_f(i:L) \equiv I_f(i:L)$. While this induction formula is similar to that used for unified constraints, it differs in its reliance on *monotonic parameters* rather than *context constraints* to predict suitable induction hypothesis.

Given suitable monotonic parameter(s), the base case is trivially true for all their monotonic constraints, while the inductive case follows trivially from the monotonically increasing (or decreasing) properties of the parameters or arithmetic terms.

Our main task is therefore to find useful monotonic constraints. We propose a simple technique, based purely on monotonic properties of parameters, to predict suitable monotonic constraints. For simplicity, we assume that the monotonic parameters are of a simpler form: $a_j^{i:L} = a_j^L + c_{ij}$ where c_{ij} is a constant. With this simpler form, we can ignore $\{A_f(i:L)\}_{i \in 1..m}$ in the above induction formula since its monotonic behaviour can be determined directly, without the assumption set.

For single increasing (or decreasing) monotonic parameters, the following technique can find suitable conjuncts for the monotonic constraint.

Procedure 2: *Handling Single Monotonic Parameters*
Consider a monotonic parameter, a_j, whose forward substitutions for the m recursive calls are $\{a_j^{i:L} = a_j^L + c_{ij}\}_{i \in 1..m}$ where $\{c_{ij}\}_{i \in 1..m}$ are constants with determinable signs.

We allow new conjuncts to be added to M_f under the following scenarios:

- Add $(a_j \geq a_j^{[]})$ if $\forall i.\ c_{ij} \geq 0$.
- Add $(a_j \leq a_j^{[]})$ if $\forall i.\ c_{ij} \leq 0$.

For example, the two parameters of *bin* satisfy the monotonically decreasing property. Hence, $n \leq n^{[]}$ and $k \leq k^{[]}$. It is also possible to apply monotonic constraints to a group of two or more arithmetic parameters. For example, given two monotonically decreasing parameters, it automatically follows that their sum is also decreasing. In the case of *bin*, we have $(n + k \leq n^{[]} + k^{[]})$ for its two parameters. This fact follows automatically but does not result in tighter bounds. What is perhaps more interesting is that the *difference* of the two parameters might be monotonically decreasing (or increasing). Exploiting this fact, allows new tighter bounds to be inferred. For example, the difference $(n - k)$ is monotonically decreasing since $(n^{1:L} - k^{1:L} = n^L - k^L)$ and $(n^{2:L} - k^{2:L} = n^L - k^L - 1)$ hold for the two recursive calls. Thus, the conjunct, $(n - k \leq n^{[]} - k^{[]})$, can be added to obtain tighter bounds. We propose the following technique to predict plausible bounds for *two* monotonic parameters.

Procedure 3: *Handling Double Monotonic Parameters*
Consider two monotonic parameters, a_1, a_2, defined by $\bigcup_{j \in 1..2} \{a_j^{i:L} = a_j^L + c_{ij}\}_{i \in 1..m}$ with $\{c_{i1}, c_{i2}\}_{i \in 1..m}$ as constants.

To find suitable monotonic constraint M_f for the two monotonic parameters, we attempt to find two non-zero values $\{k_1, k_2\}$ which satisfy the following inequalities:

$$\bigwedge_{i \in 1..m} (k_1 \times c_{i1} + k_2 \times c_{i2} \leq 0) \wedge \bigwedge_{j \in 1..2} (k_j \neq 0).$$

Such a relationship is a sufficient condition for the existence of a monotonic constraint (for two monotonic parameters) of the form: $(k_1 \times a_1 + k_2 \times a_2 \leq k_1 \times a_1^{[]} + k_2 \times a_2^{[]})$. However, some of these constraints may actually be redundant (or looser than existing bounds). To select only *tighter* monotonic constraint, where possible, we impose two extra conditions, namely:

1. $\exists i \in 1..m.\ k_1 \times c_{i1} + k_2 \times c_{i2} = 0 \wedge (k_1 \times c_{i1} \neq 0)$.
2. Minimise $max\{\ |\ k_1 \times c_{i1} + k_2 \times c_{i2}\ |\ \}_{i \in 1..m}$.

The first condition permits the boundary relationship between the two parameters to be detected, and also avoids the summation of two parameters that are both increasing (or both decreasing). In other words, it forces us to consider the difference (rather than the sum) of the two monotonic parameters. The second condition is to obtain a normalised set of values for $\{k_1, k_2\}$. Though the above proposal is presently formulated for two parameters, it is straightforward to extend to multiple parameters. This can be achieved by regarding previous monotonic expressions as parameters.

For the *bin* example, a tighter constraint for its pair of monotonic parameters is $(n - k \leq n^{[]} - k^{[]})$. After projecting the bounds on the two parameters, we obtain:

$$\begin{aligned} M_{bin} &\equiv (n - n^{[]} + k^{[]}) \leq k \leq k^{[]} \wedge n \leq min\{n^{[]} - k^{[]} + k, n^{[]}\} \\ &\equiv (n - n^{[]} + k^{[]}) \leq k \leq k^{[]} \wedge n \leq n^{[]} - k^{[]} + k \end{aligned}$$

Note that the new upper bound of $(n \leq n^{[]} - k^{[]} + k)$ (from two parameters) is actually tighter than the earlier monotonic bound, $(n \leq n^{[]})$ (from one parameter). This redundant bound can be detected by the tight bounds checking rules of Figure 4, and then eliminated as shown above.

3.5 Final Bounds Constraint

The final bounds constraint is obtained by combining the *unified* and *monotonic* constraints, as follows:

$$BOUNDS_f \equiv U_f \wedge M_f$$

In the case of *bin* function, the *combined* bounds constraint for k, n is:

$$\begin{aligned} & BOUNDS_{bin} \\ &\equiv (0 \leq k^{[]}) \wedge (max\{1, k^{[]}\} \leq n^{[]}) \ \textbf{implies} \\ &\qquad (0 \leq k \leq n) \wedge (max\{1, k\} \leq n) \\ &\qquad \wedge (n \leq n^{[]} - k^{[]} + k) \wedge (n - (n^{[]} - k^{[]}) \leq k \leq k^{[]}) \\ &\equiv (0 \leq k^{[]}) \wedge (max\{1, k^{[]}\} \leq n^{[]}) \ \textbf{implies} \\ &\qquad max\{0, n - (n^{[]} - k^{[]})\} \leq k \leq min\{n, k^{[]}\} \\ &\qquad \wedge\ max\{1, k\} \leq n \leq n^{[]} - k^{[]} + k \end{aligned}$$

The bound, $max\{0, n - (n^{[]} - k^{[]})\} \leq k \leq min\{n, k^{[]}\}$, is needed when our tactic introduces a lambda abstraction with the k parameter. The bound, $max\{1, k\} \leq n \leq n^{[]} - k^{[]} + k$, is useful under a different scenario should a lambda abstraction with the n parameter be introduced instead (see Section 5).

4 Supporting Tactics

With the bounds constraint for the various parameters, it is possible to proceed with the vector conversion tactic [6]. However, two basic supporting transformations are needed, namely:

1. The pre-condition $U_f^{[]}(\equiv U_f B_f([]))$ must be explicitly added for the root call.

2. Initial arithmetic parameters of $\vec{v}^{[]}$ may have to be made available to all recursive calls.

Consider a self-recursive function f:

$$f(\vec{v}) = \widehat{e_f}\langle f(\vec{t_1}),..,f(\vec{t_m})\rangle$$

To incorporate the pre-condition, we introduce an identical local function and add $U_f^{[]}$, as follows:

$$\begin{array}{l} f(\vec{v}) = let\ \{\ f'(\vec{v}) = \widehat{e_f}\langle f'(\vec{t_1}),..,f'(\vec{t_m})\rangle\ \} \\ \qquad\quad in\ if\ U_f^{[]}\ then\ f'(\vec{v})\ else\ C_{f[]} \end{array}$$

where $C_{f[]}$ is the value (possibly $\perp$) of f when $U_f^{[]}$ is false. The value of $C_{f[]}$ could either be obtained by symbolic evaluation or directly by calling the old definition of f.

Also, the bounds constraint may refer to the arithmetic parameters, $\vec{v}^{[]}$, of the root recursive call. These parameters can be made available via local variables, $\vec{v0}$, which captures the argument values of the root call, as shown below.

$$\begin{array}{l} f(\vec{v}) = let\ \{\ \vec{v0} = \vec{v}\ ;\ f'(\vec{v}) = \widehat{e_f}\langle f'(\vec{t_1}),..,f'(\vec{t_m})\rangle\ \} \\ \qquad\quad in\ if\ U_f([])\ then\ f'(\vec{v})\ else\ C_{f[]} \end{array}$$

Applying these two supporting transformations to *bin* results in:

$$\begin{array}{l} bin(n,k) = let\ \{\ n0 = n\ ;\ k0 = k\ ; \\ \qquad\qquad\qquad bin'(n,k) = case\ n\ of\ \{\ 0 \rightarrow 1\ ; \\ \qquad\qquad\qquad\qquad m{+}1 \rightarrow if\ k \leq 0\ or\ k \geq n\ then\ 1 \\ \qquad\qquad\qquad\qquad\qquad else\ bin'(m,k{-}1){+}bin'(m,k)\}\} \\ \qquad\qquad in\ if\ (0 \leq k \wedge max\{1,k\} \leq n)\ then\ bin'(n,k) \\ \qquad\qquad\quad else\ 1 \end{array}$$

where $C_{bin[]} \equiv 1$ since the *bin* function returns the value 1 if its pre-condition is false. This result is now in a form suitable for vector conversion to the target program shown in Section 1.

5 Improvement Tactics

Two more advanced transformation tactics are frequently needed to enhance the lambda abstraction tactic. This may be required when different parameters are targeted for elimination via lambda abstraction. They are (i) *invariant test floating* tactic, and (ii) *circular variable* tactic. Due to space limitation, we shall only provide an outline of the two tactics using *bin* as an example.

For the earlier *bin* function, we could actually decide to keep *k* as the main recursion parameter, and have *n* lifted as part of the lambda abstraction. This results in:

$$\begin{array}{l} bin(n,k) = let\ \{\ n0 = n\ ;\ k0 = k\ ; \\ \qquad\qquad\quad bin2(k) = (\backslash n \rightarrow case\ n\ of\ \{\ 0 \rightarrow 1\ ; \\ \qquad\qquad\qquad\quad m{+}1 \rightarrow if\ k \leq 0\ or\ k \geq n\ then\ 1 \\ \qquad\qquad\qquad\qquad\quad else\ bin2(k{-}1)(m){+}bin2(k)(m)\})\} \\ \qquad\qquad in\ if\ (0 \leq k \wedge max(1,k) \leq n)\ then\ bin2(k)(n) \\ \qquad\qquad\quad else\ 1 \end{array}$$

Often, after such a lambda abstraction tactic, we may have one or more conditional tests (or case selections) that are *invariant* within the lambda abstraction (i.e. not depending on its bound variables). As the lambda abstraction may be invoked multiple times, it is beneficial to float out such conditional tests (or case selections) to a location where it would be invoked only once. This can help improve the overall performance of the resulting code.

We propose to float invariant tests (selections) of *recursive* conditionals (case), out of the lambda abstraction. Recursive conditionals (or case) are constructs with self-recursive calls in one or more of their branches. There is no need to deal with non-recursive conditionals (or case) since they are invoked only once on reaching their respective base cases. The two rules for floating invariant tests are given below. Note that $n \notin freeVars(b)$ holds, while *IsRec* is a predicate to check for recursive calls.

$$\begin{array}{l} (\backslash n \rightarrow \widehat{e_a}\langle if\ \widehat{e_b}\langle b\rangle\ then\ t1\ else\ t2\rangle) \Rightarrow \\ \quad if\ b\ then\ (\backslash n \rightarrow \widehat{e_a}\langle if\ \widehat{e_b}\langle True\rangle\ then\ t1\ else\ t2\rangle) \\ \quad else\ (\backslash n \rightarrow \widehat{e_a}\langle if\ \widehat{e_b}\langle False\rangle\ then\ t1\ else\ t2\rangle) \\ \qquad\qquad\qquad\qquad\qquad IF\ IsRec(t1) \vee IsRec(t2) \\ (\backslash n \rightarrow \widehat{e}\langle case\ b\ of\ \{p_i \rightarrow t_i\}_{i \in N}\rangle) \Rightarrow \\ \quad (case\ b\ of\ \{p_i \rightarrow (\backslash n \rightarrow \widehat{e}\langle t_i\rangle[p_i/b])\}_{i \in N}) \\ \qquad\qquad\qquad\qquad\qquad IF\ \exists j \in N.\ IsRec(t_j) \end{array}$$

The above floatations enable the invariant tests to be evaluated and invoked only once per lambda abstraction. In addition, we can propagate the test results' contexts (e.g. $b = True$, $b = False$ or $b = p_i$) to enable further optimisation. However, there is a minor change to the program's semantics due to the possibility of floating conditional/case that lies in a non-strict context (e.g. inside the branch of another conditional/case). To avoid introducing non-termination, we require such (non-strict) invariant tests be terminating[3], assuming that the arithmetic parameters are strict.

In the case of *bin*, there are two base tests, $(k \leq 0)$ and $(k \geq n)$. As the first base test is *invariant* (of the bound parameter *n*) and *terminating*, we can float it out, followed by further optimisation, as follows:

$$\begin{array}{l} bin(n,k) \\ \quad = let\ \{\ n0 = n\ ;\ k0 = k\ ; \\ \qquad\quad bin2(k) \\ \qquad\quad = if\ (k \leq 0)\ then\ (\backslash n \rightarrow case\ n\ of\ \{\ 0 \rightarrow 1; \\ \qquad\qquad\qquad\qquad\qquad\qquad m{+}1 \rightarrow 1\ \}) \\ \qquad\qquad else\ (\backslash n \rightarrow case\ n\ of\ \{\ 0 \rightarrow 1; \\ \qquad\qquad\quad m{+}1 \rightarrow if\ k \geq n\ then\ 1 \\ \qquad\qquad\qquad else\ bin2(k{-}1)(m){+}bin2(k)(m))\ \}\ \} \\ \quad\ in\ if\ (0 \leq k \wedge max(1,k) \leq n)\ then\ bin2(k)(n)\ else\ 1\ ; \\ \quad = let\ \{\ n0 = n\ ;\ k0 = k\ ; \\ \qquad\quad bin2(k) = if\ (k \leq 0)\ then\ (\backslash n \rightarrow 1) \\ \qquad\qquad\qquad else\ (\backslash n \rightarrow if\ k \geq n\ then\ 1 \\ \qquad\qquad\qquad\quad else\ bin2(k{-}1)(n{-}1){+}bin2(k)(n{-}1))\ \}\ \} \\ \quad\ in\ if\ (0 \leq k \wedge max(1,k) \leq n)\ then\ bin2(k)(n)\ else\ 1 \end{array}$$

Such floating of invariant tests can often lead to further optimisation. In particular, note how the two *case* constructs which test on variable *n* can be eliminated. The first *case* has two identical branches which can be combined, while the special context of $n > 0$ in the second *case* allowed us to reduce it to a single branch, before its elimination.

A remaining problem of the above *bin2* definition is that there is a recursive call *bin2(k)* which is an exact copy of its LHS. Using a rule of the circular variable tactic, we can eliminate such calls, as follows:

$$(f(\vec{v}){=}\widehat{e}\langle f(\vec{v})\rangle) \Rightarrow (f(\vec{v}){=}let\ r{=}\widehat{e}\langle r\rangle\ in\ r)$$

Together with vector conversion, our final target program becomes:

[3]This is typically so for boolean arithmetic terms using primitive operators.

```
bin(n,k) = let { n0 = n ; k0 = k ;
                 bin2(k) = if (k ≤ 0) then (\n → 1)
                           else let { r = (!) vec ; z = bin2(k-1) ;
                                      bn n = if k ≥ n then 1
                                             else z(n-1)+r(n-1) ;
                                      l = max(1,k) ;
                                      u = min(n0+k-k0,n0) ;
                                      vec = array (l,u) [a := bn a
                                                         | a ← [l..u]] }
                           in r }
           in if (0 ≤ k0 ∧ max(1,k0) ≤ n0) then bin2(k0)(n0)
              else 1
```

Note that r is a circular variable to represent the vector of *bin2(k)*. This vector is computed from the elements of the previous vector z down the recursion, as well as from other elements of r itself.

6 Run-Time Performance

To help justify our transformation (and analysis), we collected the CPU times and heap space statistics for five different versions of *bin* in Table 1. They are for (i) original definition of *bin* (ii) fully-memoized version (iii) vectorized version with a loose bound of $0 \leq k \leq n$ (iv) vectorized version with a tighter bound from our bounds inference method (v) circular version. As expected, the original version performed badly (those with '-' did not complete after an hour). The fully-memoized version is also impractical for large arguments. The vectorized version with tighter bounds is still *7-26%* faster than a corresponding vectorized version with looser bound typically given by programmer. This improvement represents a good payoff from using our more sophisticated bounds inference method for obtaining tighter parameter bounds. Lastly, the circular *bin* function performed even better due to a further reduction of its conditional test, achieved using the *invariant test floating* tactic from Section 5.

7 Related Work and Conclusion

On the analyses front, a type-based analysis was recently proposed in [9] to obtain information about the sizes of recursively defined data structures. This analysis can be used to prove some basic properties (such as deadlocks, non-termination) of reactive systems. We considered the option of using this sized analysis for bounds inference but found it to be unsuitable. Apart from its restriction to the natural number domain and the adoption of type-checking (rather than inference), the present sized analysis can only relate the *sizes* of parameters, not the *values* of the parameters themselves. In other words, it could state that two parameters have the same bound (size) but not that one parameter is always less than another - a more precise result achieved by our bounds inference.

The omega calculator of [15] uses constraint solving to support exact dependence analysis for vectorising and parallelizing compilers. Pugh's work is complimentary to our bounds inference analysis, since we could make use of practical advances in his constraint technology for efficiently solving our Presburger bounds too. The bounds inference work here is also related to a compiler optimization technique, called array subscript range checks elimination [8, 11]. While we infer the bounds of the parameters of lambda abstractions in order to convert to arrays, range checks optimization attempts to remove redundant checks present in user-supplied array-based loops. Our work can be seen as an utopian alternative to range checking. It is formulated for the more general recursive programs, and automatically provides 100% elimination of range checks, whenever lambda abstraction are successfully converted to vectors with safe bounds. An interesting future work is to adapt our bounds inference method to handle array-based programs directly. Apart from the possibility of static bounds checking, we can expect the sizes of arrays (that are dynamically allocated) to be adjusted downwards whenever tighter bounds could be inferred.

On the transformation front, many techniques have been proposed in the past for avoiding redundant function calls. One of the earliest techniques uses memo-functions [12]. Memo-functions are special functions which remember/store either some or all of their previously computed function calls in a memo-table, so that re-occurring function calls can have their results retrieved from the memo-table rather than re-computed. Though general (with no analysis required), memo-functions suffer from large table management overheads.

Other transformation techniques (e.g. tupling and tabulation) may result in more efficient programs but they usually require program analyses and may be restricted to sub-classes of programs. Most of these techniques analyse the DGs algebraically. This approach typically uses appropriate conditions of descent functions to infer redundancy patterns in the DGs. Descent functions are functions applied to the arguments of subsidiary recursive calls. Some common relationships of descent functions which were discovered by Cohen[7] are the *common generator*, *periodic commutative* and *commutative* redundancy relationships. Though fairly extensive, there are still other classes of programs which are not easily addressed by the algebraic approach, without getting ad-hoc via an increasing set of exceptions.

Our approach advocates the use of a collection of transformation tactics. By a novel combination of tupling and lambda abstraction, we can efficiently memoize programs which require dynamic-sized tabulations. To obtain efficient memoization, we have also proposed the vector conversion tactic together with a bunch of supporting tactics, including more advanced ones given in Section 5. The main contribution of this paper is a systematic collection of bounds inference analyses needed to support vector conversion. Our analyses can collect relevant context constraint for each recursive call before attempting to obtain a unified constraint which holds for all recursive calls. In addition, each set of parameters which are monotonically increasing (or decreasing) can also be used to obtain new monotonic contraints. The final parameter bounds are formed by combining unified and monotonic constraints together. The decidability of the analyses is facilitated by our use of arithmetic expressions in Presburger form. With this analyses, many of the more exotic transformations that are hand-analysed in [1] can now be systematically handled.

Acknowlegements We would like to thank Aik-Hui and Siau-Cheng for their helpful discussions, and to ICFP referees for providing insightful comments.

References

[1] Richard S. Bird. Tabulation techniques for recursive

Table 1: Time & Space Statistics for different *bin* under the Glasgow Haskell Compiler

Calls	*Original*		*Fully Memoized*		*Vectorized (loose)*		*Vectorized (tight)*		*Circular*	
	Sec	*Cells*	*Sec*	*Cells*	*Sec*	*Cells*	*Sec*	*Cells*	*Sec*	*Cells*
bin(10,5)	0.034	46924	0.044	15632	0.034	10904	0.032	9628	0.021	7068
bin(25,12)	218.04	936054720	0.480	93616	0.058	47168	0.054	37956	0.044	28612
bin(30,15)	-	-	0.91	134648	0.74	64920	0.070	51484	0.046	39604
bin(100,50)	-	-	99.24	1501772	0.400	606044	0.296	449248	0.250	366948
bin(200,100)	-	-	-	-	1.422	2331032	1.070	1697424	0.870	1412524

programs. *ACM Computing Surveys*, 12(4):403–417, December 1980.

[2] W.W. Bledsoe. A new method for proving certain Presburger formulae. In *Proc of ICJAI*, pages 15–21, 1975.

[3] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, January 1977.

[4] Wei-Ngan Chin. Towards an automated tupling strategy. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Copenhagen, Denmark, June 1993. ACM Press.

[5] Wei-Ngan Chin and John Darlington. A higher-order removal method. *LISP and Symbolic Computation*, 9(4):287–322, 1996.

[6] Wei-Ngan Chin and Masami Hagiya. A transformation method for dynamic-sized tabulation. *Acta Informatica*, 32:93–115, March 1995.

[7] Norman H. Cohen. Eliminating redundant recursive calls. *ACM Trans. on Programming Languages and Systems*, 5(3):265–299, July 1983.

[8] 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.

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

[10] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[11] 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.

[12] Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, April 1968.

[13] N. Perry. Hope+. Tech Note: IC/FPR/LANG/2.5.1/7, DoC, Imperial College, 1989.

[14] Alberto Pettorossi. A powerful strategy for deriving programs by transformation. In *3rd ACM LISP and Functional Programming Conference*, pages 273–281. ACM Press, 1984.

[15] William Pugh. The omega test: A fast practical integer programming algorithm for dependence analysis. *Communications of ACM*, 8:102–114, 1992.

[16] Akihiko Takano. Generalized partial computation for a lazy functional language. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–11, New Haven, Connecticut, August 1991. ACM Press.

A One More Example

We show how the bounds inference method can be used to optimize the longest common sub-sequence problem. Consider the following function to find the longest common subsequence of two strings, *XS* and *YS*, where *N=length(XS)*, *M=length(YS)*.

$$
\begin{aligned}
csub(j^L,k^L) \;=\; & \mathit{if}\ j^L > N\ \mathit{or}\ k^L > M\ \mathit{then}\ 0 \\
& \mathit{else\ if}\ XS\ !!\ j^L = YS\ !!\ k^L \\
& \mathit{then}\ 1+csub(j^L+1,k^L+1) \\
& \mathit{else}\ max(csub(j^L,k^L+1),csub(j^L+1,k^L));
\end{aligned}
$$

The backwards parameter substitutions for the three recursive calls are:

$$
\begin{aligned}
B_{csub}(1:L) &\equiv \{j^L = j^{1:L}-1, k^L = k^{1:L}-1\} \\
B_{csub}(2:L) &\equiv \{j^L = j^{2:L}, k^L = k^{2:L}-1\} \\
B_{csub}(3:L) &\equiv \{j^L = j^{3:L}-1, k^L = k^{3:L}\}
\end{aligned}
$$

Also, the individual call constraints for the three recursive calls are:

$$
\begin{aligned}
I_{csub}(1:L) &\equiv (j \le N+1) \wedge (k \le M+1) \\
I_{csub}(2:L) &\equiv (j \le N) \wedge (k \le M+1) \\
I_{csub}(3:L) &\equiv (j \le N+1) \wedge (k \le M)
\end{aligned}
$$

An attempt is now made to obtain its unified bounds constraint, as follows:

$$
\begin{aligned}
\forall i.\, I_{csub}(i:L) \;& \vdash \textstyle\bigvee_{i\in 1..2} I_{csub}(i:L) \\
& \vdash ((j \le N+1) \wedge (k \le M+1)) \\
& \qquad \vee\ ((j \le N) \wedge (k \le M+1)) \\
& \qquad \vee\ ((j \le N+1) \wedge (k \le M)) \\
& \vdash (j \le min\{N+1, N, N+1\}) \\
& \qquad \wedge (k \le min\{M+1, M+1, M\}) \\
& \vdash (j \le N) \wedge (k \le M)
\end{aligned}
$$

Adding in the pre-condition, we have:

$$U_{csub} \equiv (j^{\square} \le N) \wedge (k^{\square} \le M)\ \textbf{implies}\ (j \le N) \wedge (k \le M)$$

Since the two parameters of *csub* are monotonically increasing parameters, we propose:

$$M_{csub} \equiv (j \ge j^{\square}) \wedge (k \ge k^{\square})$$

Though both parameters are monotonically increasing, it is *not* possible to find an appropriate difference that is monotonically increasing. This fact is predicted by our technique on the pair of monotonic parameters (see Defn 3). Both the unified constraint and the monotonic constraint can be used to obtain the following the projected bounds constraint.

$$BOUNDS_{csub} \equiv (j^{\square} \leq N) \wedge (k^{\square} \leq M) \textbf{ implies } (j^{\square} \leq j \leq N) \wedge (k^{\square} \leq k \leq M)$$

With the bounds for k, we can apply the lambda abstraction tactic (plus supporting tactics), before vector conversion to obtain the following dynamically-tabulated definition:

```
csub(j,k)
  = let { j0 = j ; k0 = k ;
    csub'(j) = if j > N then (\k → 0)
               else let { r = (!) vec ;
                          z = csub'(j+1) ;
                          cn k = if k > M then 0
                                 else if (XS !! j == YS !! k )
                                 then 1+z(k+1)
                                 else max(r(k+1),z(k)) ;
                          l = k0; u = M;
                          vec = array (l,u) [a := cn a
                                           | a ← [l..u]] }
                    in r }
    in if (j0 ≤ N ∧ k0 ≤ M) then csub'(j0)(k0)
       else 0
```

Note that $(j > N)$, an invariant test, is lifted out of its lambda abstraction by the tactic in Section 5, while $(k > M)$, which has a bound variable, is not lifted.