Inferring Disjunctive Postconditions

Corneliu Popeea and Wei-Ngan Chin

Department of Computer Science, National University of Singapore
{corneliu,chinwn}@comp.nus.edu.sg

Abstract. Polyhedral analysis [9] is an abstract interpretation used for automatic discovery of invariant linear inequalities among numerical variables of a program. Convexity of this abstract domain allows efficient analysis but also loses precision via convex-hull and widening operators. To selectively recover the loss of precision, sets of polyhedra (disjunctive elements) may be used to capture more precise invariants. However a balance must be struck between precision and cost.
We introduce the notion of affinity to characterize how closely related is a pair of polyhedra. Finding related elements in the polyhedron (base) domain allows the formulation of precise hull and widening operators lifted to the disjunctive (powerset extension of the) polyhedron domain. We have implemented a modular static analyzer based on the disjunctive polyhedral analysis where the relational domain and the proposed operators can progressively enhance precision at a reasonable cost.

1 Introduction

Abstract interpretation [7, 8] is a technique for approximating a basic analysis, with a refined analysis that sacrifices precision for speed. Abstract interpretation relates the two analyses using a Galois connection between the two corresponding property lattices. The framework of abstract interpretation has been used to automatically discover program invariants. For example, numerical invariants can be discovered by using numerical abstract domains like the interval domain [6] or the polyhedron domain [9]. Such convex domains are efficient and their elements represent conjunctions of linear inequality constraints.

Abstract domains can be designed incrementally based on other abstract domains. The powerset extension of an abstract domain [8, 12] refines the abstract domain by adding elements that allow disjunctions to be represented precisely. Unfortunately, analyses using powerset domains can be exponentially more expensive compared to analyses on the base domain. One well-known approach to control the number of disjuncts during analysis is to use a powerset domain where the number of disjuncts is syntactically bounded. In this setting, the challenge is to find appropriate disjuncts that can be merged without (evident) losses in precision. Recently, a technique for disjunctive static analysis has been proposed and implemented [24]. The analysis is formulated for a generic numerical domain and an heuristic function based on the Hausdorff distance is used to merge related disjuncts. Besides combining related disjuncts, another difficulty in designing a disjunctive abstract domain is to define a precise and convergent widening operator.

In this paper, we develop a novel technique for *selective hulling* to obtain precise fixed points via disjunctive inference. Our framework uses a fixed point algorithm guided by an affinity measure to find and combine disjuncts that are related. We also develop a precise widening operator on the powerset domain by using a similar affinity measure. We have built a prototype system to show the utility of the inferred postconditions and the potential for tradeoff between precision and analysis cost.

This paper is organized as follows: an overview of our method with a running example is presented in Sect. 2. Section 3 introduces our source language and a set of reasoning rules that collect a (possibly recursive) constraint abstraction from each method/loop to be analyzed. Those recursive constraint abstractions are subjected to the disjunctive fixed point analysis based on selective hulling and widening as described in Sect. 4. Our implementation and experimental results are presented in Sect. 5. Section 6 presents related work, while Sect. 7 concludes.

2 Overview

To provide an overview of our method, we will consider the following example. This program computes the index l of a specific element in an array of size N. The array contents has been abstracted out and only the updates to the index variables l and x have been retained. The call to the method *randBool* abstracts whether the current element indexed by x is found to satisfy the search criterion. Whenever the criterion is satisfied, the index variable l is updated, as well as the boolean flag *upd*. An assertion at the end of the loop could check that, whenever an element has been found (*upd*=*true*), its index l is a valid index of the array ($0 \leq l < N$). The aim of our static analysis is to infer disjunctive invariants that can help prove such properties.

```
x:=0;upd:=False;
while (x < N) do {
  if (randBool()) then {
    l:=x;upd:=True
  } else { () };
  x:=x+1 }
```

A static analysis can be formulated as a *state-based* analysis: guided by the program state at the beginning of the loop, it computes the loop postcondition as a program state approximation [9, 13, 24]. As an alternative, our method is related to *trace-based* analysis [4] and computes the loop summary as a transition from the prestate (before the loop) to the poststate (after the loop body).

Our analysis is formulated in two stages. Firstly, it collects a constraint abstraction from the method/loop body to be analyzed. This abstraction can be viewed as an intermediate form and is related to the constraint abstraction introduced in [14]. As a second step, an iterating process will find the fixed point for the constraint abstraction function.

For the running example, the constraint abstraction named *wh* represents the input-output relation between the loop prestate (in terms of X, the unprimed variables x, N, l, upd) and the loop poststate (in terms of X', the primed variables x', N', l', upd').

$$\begin{aligned} wh(X,X') :-\ & ((nochange(X) \wedge x'{<}N') \circ_{\{l,upd\}} \\ & \quad (l'{=}x \wedge upd'{=}1 \vee nochange(l, upd)) \circ_{\{x\}} \\ & \quad (x'{=}x{+}1) \circ_X wh(X,X')) \\ & \vee (nochange(X) \wedge x' {\geq} N') \end{aligned}$$

The *nochange* operator is a special transition where original and primed variables are made equal: $nochange(\{\}) =_{df}$ `true`; $nochange(\{x\}{\cup}X) =_{df} (x'{=}x) {\wedge} nochange(X)$. The composition operator $(\phi_1 \circ_W \phi_2)$ is left-associative and composes the input-output relations ϕ_1 and ϕ_2 updating W variables as specified by ϕ_2 formula. Formally, given ϕ_1, ϕ_2, and the set of variables to be updated $X{=}\{x_1, \ldots, x_n\}$, the composition operator $\circ_X$ is defined as:

$$\begin{aligned} & \phi_1 \circ_X \phi_2 =_{df} \exists\, r_1..r_n \cdot \rho_1\, \phi_1 \wedge \rho_2\, \phi_2 \\ & \text{where } r_1, \ldots, r_n \text{ are fresh variables; } \rho_1 = [x_i' \mapsto r_i]_{i=1}^n \\ & \qquad\qquad\qquad\qquad ;\rho_2 = [x_i \mapsto r_i]_{i=1}^n \end{aligned}$$

Note that ρ_1 and ρ_2 are substitutions that link each latest value of x_i' in ϕ_1 with the corresponding initial value x_i in ϕ_2 via a fresh variable r_i.

With these two operators, the effects of the loop sub-expressions are composed to obtain the effect of the entire loop body. The 1st line of the constraint abstraction corresponds to the loop test that is satisfied. The 2nd line stands for the body of the conditional expression from the loop. Note that the boolean constants *False* and *True* are modeled as integers 0 and 1. The 3rd line represents the assignment that increments x by 1 composed with the effect of subsequent loop iterations (the occurrence of the wh constraint abstraction). The 4th and last line stands for the possibility that the loop test is not satisfied.

After some simplifications, the constraint abstraction reduces to:

$$\begin{aligned} wh(X,X') :- \exists X_1 \cdot (\ & (x_1{=}x{+}1 \wedge N_1{=}N \wedge l_1{=}x \wedge upd_1{=}1 \wedge wh(X_1, X')) \\ \vee\ & (x_1{=}x{+}1 \wedge N_1{=}N \wedge l_1{=}l \wedge upd_1{=}upd \wedge wh(X_1, X')) \\ \vee\ & (x'{=}x \wedge N'{=}N \wedge l'{=}l \wedge upd'{=}upd \wedge x'{\geq}N')) \end{aligned}$$

where X_1 denotes the local variables (x_1, N_1, l_1, upd_1).

The analysis goal is then to compute a fixed point approximation for the constraint abstraction function. This function takes as argument a transition depending on X, X' and its result is also expressed as a transition dependent on the same variables. Both transitions can either be approximated by polyhedra or, more precisely, by sets of polyhedra. The first case is akin to the polyhedral analysis from [9] and is reviewed next. For the second case, we will use our running example to show how to compute a disjunctive loop postcondition.

2.1 Computing Fixed Points in the Polyhedron Abstract Domain

We briefly review the method based on Kleene's fixed point iteration applied to the polyhedron abstract domain. Let $(\mathcal{L}, \leq)$ be a complete lattice, and denote by $(\mathcal{P}, \Rightarrow)$ the lattice of polyhedra. We write $\perp$ for its least element (in $\mathcal{P}$, the empty polyhedron or its representation, the formula false), and $\top$ for its greatest element (in $\mathcal{P}$, the entire n-dimensional space or its representation, the formula true). The meet and join operations in the lattice of polyhedra are, respectively, the set intersection and the convex polyhedral hull, the latter being denoted by $\oplus$. A function f that is a self-map of a complete lattice is monotone if $x \le y$ *implies* $f(x) \le f(y)$. In particular, the constraint abstraction functions derived by our analysis are monotone self-maps of the (powerset) polyhedra lattice.

The least fixed point of a monotone function f can be obtained by computing the ascending chain $f_0 = \bot$, $f_{n+1} = f(f_n)$, with $n{\ge}0$. If the chain becomes stationary, i.e., if $f_m = f_{m+1}$ for some m, then f_m is the least fixed point of f. In the case of a lattice infinite in height (as the lattice of polyhedra), an ascending chain may be infinite, and a widening operator must be used to ensure convergence. A widening operator ∇ is a binary operator to ensure that the iteration sequence $f_0 = \bot$, $f_{k+1} = f(f_k)$ followed by $f_{n+1} = f_n \nabla f(f_n)$, with $n > k$, converges. In this case, the limit of the sequence is known as a *post fixed point* of f. A post fixed point is a sound approximation of the least fixed point, and the criterion to verify that x is a post fixed point for f is that $x \ge f(x)$. For the polyhedron domain, the standard widening operator was introduced in [9]. Intuitively, the result of the widening $\phi_1 \nabla \phi_2$ is obtained by removing from ϕ_1 those conjuncts that are not satisfied by the next iteration ϕ_2.

For our running example, the fixed point iteration starts with the least element of the abstract domain represented by the *false* formula. The first approximation wh_1 is a transition formula that considers that the loop test fails and the loop body is never executed:

$$wh_1 :\!-\; (x'{=}x \wedge N'{=}N \wedge l'{=}l \wedge upd'{=}upd \wedge x'{\ge}N')$$

The next iteration is a three-disjunct formula that cannot be represented in the polyhedron domain. An approximation for the disjunctive formula is computed using the convex hull operator. A disjunctive formula can be viewed as a set of disjuncts: $\phi = \vee_{i=1}^{n} d_i = \{d_i\}_{i=1}^{n}$. Operators on these disjuncts could be used either infixed or prefixed. For example, given $\phi{=}d_1{\vee}d_2$ then $\oplus\phi = \oplus\{d_1, d_2\} = d_1{\oplus}d_2$.

$$\begin{array}{ll} wh_2 :\!- & (x'{=}x{+}1 \wedge N'{=}N \wedge l'{=}x \wedge upd'{=}1 \wedge x'{\ge}N') \\ & \vee\, (x'{=}x{+}1 \wedge N'{=}N \wedge l'{=}l \wedge upd'{=}upd \wedge x'{\ge}N') \\ & \vee\, (x'{=}x \wedge N'{=}N \wedge l'{=}l \wedge upd'{=}upd \wedge x'{\ge}N') \\ wh'_2 :\!- & \oplus wh_2 = (x{\le}x'{\le}x{+}1 \wedge N'{=}N \wedge x'{\ge}N) \\ wh'_3 :\!- & \oplus wh_3 = (x{\le}x'{\le}x{+}2 \wedge N'{=}N \wedge x'{\ge}N) \end{array}$$

The iterating sequence will not converge since the inequality $x'{\le}x$ will be translated at the following iterations into $x'{\le}x{+}1$, $x'{\le}x{+}2$ and so on. Convergence is ensured by the widening operator which simplifies as follows:

$$wh''_3 :\!-\; wh'_2 \nabla wh'_3 = (x{\le}x' \wedge N'{=}N \wedge x'{\ge}N)$$

This result proves to be a post fixed point for the wh function. However, the result is rather imprecise as it does not capture any information about the value of l or the flag upd at the end of the loop. Intuitively, such information was present in wh_2 and wh_3, but approximated by the convex hull operator to obtain wh_2' and wh_3'. Next, we outline a method to compute disjunctive fixed points able to capture this kind of information.

2.2 Computing Fixed Points in a Disjunctive Abstract Domain

The two ingredients that we use to compute disjunctive fixed points are counterparts to the convex hull and widening operators from the conjunctive case. Both operators ensure a bound on the number of disjuncts allowed in the formulae.

We first propose a *selective hull* operator $\oplus_m$ parameterized by a constant m that takes as argument a disjunctive formula and collapses these disjuncts into a result with at most m disjuncts. The crux of this operator is an affinity measure to choose the two most related (affine) disjuncts from a disjunctive formula. Formally, given $\phi = \vee_{i=1}^{n} d_i$, and let d_i,d_j be the most related disjuncts as determined by their affinity, we define the selective hull operator as follows:

$$\begin{aligned} \oplus_m \phi =_{df}\ & if\ n{\leq}m\ then\ \phi \\ & else\ \ \oplus_m\ (\phi \setminus \{d_i, d_j\} \cup \{d_i \oplus d_j\}) \end{aligned}$$

Note that the convex hull operator from the polyhedron domain $\oplus$ is equivalent to $\oplus_1$ since it reduces its disjunctive argument to a conjunctive formula with one disjunct. The affinity function aims to quantify how close is the approximation $d_1 {\oplus} d_2$ from the disjunctive formula $d_1 {\vee} d_2$. Intuitively, it works by counting the number of inequalities (planes in the n-dimensional space) from the disjunctive formula that are preserved in the approximation $d_1 {\oplus} d_2$. Since it counts the number of inequalities (relations between variables), the affinity function is able to handle the relational information captured by the formulae in the polyhedron domain.

As an example, consider wh_2 and wh_3 obtained previously. The results of selective hull with m=3 the bound on the number of disjuncts are as follows:

$$\begin{aligned} wh_2''' :-\ \oplus_3\ wh_2 &= wh_2 \\ wh_3''' :-\ \oplus_3\ wh_3 &= (x{\leq}x'{\leq}x{+}2 \wedge N'{=}N \wedge x{\leq}l'{\leq}x{+}2 \wedge upd'{=}1 \wedge x{+}2{\geq}N) \\ &\vee (x{\leq}x'{\leq}x{+}2 \wedge N'{=}N \wedge l'{=}l \wedge upd'{=}upd \wedge x{+}2{\geq}N) \\ &\vee (x'{=}x \wedge N'{=}N \wedge l'{=}l \wedge upd'{=}upd \wedge x{\geq}N) \end{aligned}$$

The second operator needed in the disjunctive abstract domain is a widening operator. We propose a similar affinity measure to find related disjuncts for pairwise widening. For the two disjunctive formulae $wh_2''' = (d_1{\vee}d_2{\vee}d_3)$ and $wh_3''' = (e_1{\vee}e_2{\vee}e_3)$, the most affine pairs will distribute the widening operator:

$$\begin{aligned} wh_2''' \nabla_3 wh_3''' :-\ & (d_1{\vee}d_2{\vee}d_3)\nabla_3(e_1{\vee}e_2{\vee}e_3) = (d_1 \nabla e_1) \vee (d_2 \nabla e_3) \vee (d_3 \nabla e_3) \\ &= (x'{=}N \wedge N'{=}N \wedge x{\leq}l'{\leq}N \wedge upd'{=}1) \\ &\vee (x'{=}N \wedge N'{=}N \wedge l'{=}l \wedge upd'{=}upd \wedge x{\leq}N) \\ &\vee (x'{=}x \wedge N'{=}N \wedge l'{=}l \wedge upd'{=}upd \wedge x{>}N) \end{aligned}$$

This result proves to be a post fixed point for the wh function in the powerset domain. The first disjunct captures the updates to the variable l, thus l' can safely be used as an index for the array of size N. The last two disjuncts capture the cases where, either the loop was executed but the *then* branch of the conditional has never been taken ($x \leq N \wedge upd'=upd$), or the loop has not been executed ($x>N$).

Note that our disjunctive fixed point computation works not only for loops, but also for general recursion. Our analysis also supports mutual recursion where fixed points are computed simultaneously for multiple constraint abstraction functions.

Since the computed fixed point represents a transition, the analysis does not rely on a fixed initial state and can be implemented in a modular fashion. While modular analysis may expose more disjuncts (because no information is assumed about the initial state) and benefits more from our approach, disjunctive analysis has been shown to be also useful for global static analyses [13, 24].

3 Forward Reasoning Rules

We propose a set of forward reasoning rules for collecting a constraint abstraction for each method/loop. Some primitive methods may lack a method body and be given instead a formula ϕ: the given formula may include a safety precondition (for example, bound checks for array operations), or simply represent the input-output relation (for primitive numerical operations like *add* or *multiply*). The reasoning process is modular, starting with the methods at the bottom of the call graph.

For simplicity, we shall use an imperative language with minimal features, as given in Fig. 1. We use *meth* for method declaration, t for type, and e for expression. This language is expression-oriented and uses a normalised form for which only variables are allowed as arguments to a method call or a conditional test. A preprocessor can transform arbitrarily nested arguments to this core language form. Pass-by-reference parameters are declared for each method via the `ref` keyword, while the other parameters from $\{v^*\}$ are passed-by-value. For simplicity, we disallow aliasing amongst pass-by-reference parameters. This restriction is easily enforced in our simple language by ensuring that such arguments at each call site are distinct variables.

$$
\begin{array}{lll}
P & ::= & meth^* \\
meth & ::= & t\ mn\ ((\texttt{[ref]}\ t\ v)^*)\ \texttt{where}\ \phi\ \{e\} \\
t & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{void} \\
k & ::= & \texttt{true} \mid \texttt{false} \mid k^{\texttt{int}} \mid () \\
e & ::= & v \mid k \mid v{:=}e \mid e_1; e_2 \mid mn(v^*) \mid t\ v;\ e \\
 & & \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{while}\ v\ \texttt{do}\ e \\
\phi & ::= & s_1{=}s_2 \mid s_1{\leq}s_2 \mid \phi_1{\wedge}\phi_2 \mid \phi_1{\vee}\phi_2 \mid \exists v{\cdot}\phi \\
s & ::= & k^{\texttt{int}} \mid v \mid v' \mid k^{\texttt{int}} * s \mid s_1 + s_2
\end{array}
$$

Fig. 1. Simple imperative language

The constraint language defined by ϕ is based on Presburger arithmetic. Our framework can accommodate either a richer sublanguage for better expressivity or a more restricted sublanguage (e.g. weakly relational difference constraints [20]) for better performance. We shall assume that a type-checker exists to ensure that expressions and constraints used in a program are well-typed.

The rule [METH] associates each method mn with a constraint abstraction of the same name. Namely, $mn(v^*, w^*) :- \phi$, where v^* covers the input parameters,

$$[\textsc{const}]\quad \frac{\phi_1 = (\phi \wedge \mathbf{res}{=}k)}{\vdash \{\phi\}\, k\, \{\phi_1\}}$$

$$[\textsc{var}]\quad \frac{\phi_1 = (\phi \wedge \mathbf{res}{=}v')}{\vdash \{\phi\}\, v\, \{\phi_1\}}$$

$$[\textsc{assign}]\quad \frac{\vdash \{\phi\}\, e\, \{\phi_1\} \quad \phi_2 = \exists \mathbf{res}{\cdot}(\phi_1 {\circ_{\{v\}}} v'{=}\mathbf{res})}{\vdash \{\phi\}\, v{:=}e\, \{\phi_2\}}$$

$$[\textsc{blk}]\quad \frac{\vdash \{\phi\}\, e\, \{\phi_1\}}{\vdash \{\phi\}\, t\ v;\ e\, \{\exists v'{\cdot}\phi_1\}}$$

$$[\textsc{if}]\quad \frac{\begin{array}{c}\vdash \{\phi \wedge v'{=}1\}\, e_1\, \{\phi_1\} \\ \vdash \{\phi \wedge v'{=}0\}\, e_2\, \{\phi_2\}\end{array}}{\vdash \{\phi\}\, \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\, \{\phi_1 \vee \phi_2\}}$$

$$[\textsc{seq}]\quad \frac{\begin{array}{c}\vdash \{\phi\}\, e_1\, \{\phi_1\} \\ \vdash \{\exists \mathbf{res}{\cdot}\phi_1\}\, e_2\, \{\phi_2\}\end{array}}{\vdash \{\phi\}\, e_1; e_2\, \{\phi_2\}}$$

$$[\textsc{call}]\quad \frac{\begin{array}{c}W{=}\{v_i\}_{i=1}^{m-1} \quad \mathit{distinct}(W) \\ t_0\ mn((\mathtt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n}) \\ \mathtt{where}\ \phi_{po}\ \{...\}\end{array}}{\vdash \{\phi\}\, mn(v_1..v_n)\, \{\phi \circ_W \phi_{po}\}}$$

$$[\textsc{while}]\quad \frac{\begin{array}{c}X{=}\mathit{freevars}(v,e) \quad \vdash \{\mathit{nochange}(X) \wedge v'{=}1\}\, e\, \{\phi_1\} \\ \phi_2{=}(\phi_1 {\circ_X} wh(X,X')) \vee (\mathit{nochange}(X) \wedge v'{=}0) \\ Q{=}\{wh(X,X') :\!-\ \phi_2\} \quad \phi_{po} = fix(Q)\end{array}}{\vdash \{\phi\}\, \mathtt{while}\ v\ \mathtt{do}\ e\, \{\phi {\circ_X} \phi_{po}\} \Rightarrow \phi_{po}}$$

Fig. 2. Forward reasoning rules

while w^* covers the method's output **res** and the primed variables from pass-by-reference parameters. The fixed point analysis outlined in the previous section is invoked by $\mathit{fix}(Q)$ and returns ϕ_{po}, the input-output relation of the method. To derive suitable postconditions, we shall subject each method declaration to the following rule:

$$[\textsc{meth}]\quad \frac{\begin{array}{c}W{=}\{v_i\}_{i=1}^{n} \quad V{=}\{v_i'\}_{i=m}^{n} \quad \vdash \{\mathit{nochange}(W)\}\, e\, \{\phi\} \\ X{=}\{v_1,..,v_n,\mathbf{res},v_1',..,v_{m-1}'\} \quad Q{=}\{mn(X) :\!-\ \exists V{\cdot}\phi\} \quad \phi_{po} = \mathit{fix}(Q)\end{array}}{\vdash t_0\ mn((\mathtt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \mathtt{where}\ mn(X)\{e\} \Rightarrow \phi_{po}}$$

The inference uses a set of Hoare-style forward reasoning rules of the form $\vdash \{\phi_1\}\, e\, \{\phi_2\}$. Given a transition ϕ_1 from the beginning of the current method/loop to the prestate before e's evaluation, the judgement will derive ϕ_2, a transition from the beginning of the current method/loop to the poststate after e's evaluation. A special variable **res** is used to denote the result of method declaration as well as that of the current expression under program analysis.

In Fig. 2, the [ASSIGN] rule captures imperative updates with the help of the prime notation. The [SEQ] rule captures flow-sensitivity, while the [IF] rule captures path-sensitivity. The [CALL] rule accumulates the effect of the callee postcondition using $\phi \circ_W \phi_{po}$. This rule postpones the checking of the callee precondition to a later stage. The two rules [METH] and [WHILE] compute a postcondition (indicated to the right of the $\Rightarrow$ operator) which will be inserted in the code and used subsequently in the verification rules. The result of these rules is a definition for each constraint abstraction. As an example, consider:

```
void mnA(ref int x, int n) where (mnA(x,n,x'))
{ if x>n then x:=x-1; mnA(x,n) else () }
```

After applying the forward reasoning rules, we obtain the following constraint abstraction:

$\mathtt{mnA(x,n,x')} :- (\mathtt{x{>}n} \wedge (\exists \mathtt{x1}\cdot \mathtt{x1{=}x{-}1} \wedge \mathtt{mnA(x1,n,x')})) \vee (\mathtt{x{\leq}n} \wedge \mathtt{x'{=}x})$

Note that the forward rules can be used to capture the postcondition of any recursive method, not just for tail-recursive loops. For example, consider the following recursive method:

$\mathtt{int\ mnB(int\ x)\ where\ (mnB(x,\mathbf{res}))}\ \{\ \mathtt{if\ x{\leq}0\ then\ 1\ else\ x{:=}x{-}1; 2{+}mnB(x)}\ \}$

Applying forward reasoning rules will yield the following constraint abstraction:

$\mathtt{mnB(x,\mathbf{res})} :- (\mathtt{x{\leq}0} \wedge \mathbf{res}\mathtt{{=}1}) \vee (\mathtt{x{>}0} \wedge (\exists \mathtt{x1,r1}\cdot \mathtt{x1{=}x{-}1} \wedge \mathtt{mnB(x1,r1)} \wedge \mathbf{res}\mathtt{{=}2{+}r1}))$

The next step is to apply fixed point analysis on each *recursive* constraint abstraction. By applying disjunctive fixed point analysis, we can obtain:

$\mathtt{mnB(x,\mathbf{res})} :- (\mathtt{x{\leq}0} \wedge \mathbf{res}\mathtt{{=}1}) \vee (\mathtt{x{\geq}0} \wedge \mathbf{res}\mathtt{{=}2{*}x{+}1})$

Once a closed-form formula has been derived, we shall return to checking the validity of preconditions that were previously skipped. The rules for verifying preconditions are similar to the forward rules for postcondition inference, with the exception of three rules, namely:

$$\frac{\begin{array}{c}[\textbf{VERIFY-CALL}]\\ t_0\ mn((\mathtt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \mathtt{where}\ \phi_{po}\\ W{=}\{v_i\}_{i=1}^{m-1}\quad Z{=}\{\mathbf{res}, v'_1, .., v'_{m-1}\}\\ \phi_{pr}{=}\exists Z\cdot\phi_{po}\quad \phi \implies [v_i{\mapsto}v'_i]_{i=1}^{n}\phi_{pr}\end{array}}{\vdash \{\phi\}\ mn(v_1..v_n)\ \{\phi \circ_W \phi_{po}\}}$$

$$\frac{\begin{array}{c}[\textbf{VERIFY-WHILE}]\\ X = \mathit{freevars}(v,e)\quad \rho = X{\mapsto}X'\\ \phi_{pr} = \exists X'\cdot\phi_2\qquad \phi \implies \rho\phi_{pr}\\ \vdash \{\phi \wedge \rho\phi_{pr}\}\ e\ \{\phi'\}\end{array}}{\vdash \{\phi\}\ \mathtt{while}\ v\ \mathtt{do}\ e\ \mathtt{where}\ \phi_2\ \{\phi \circ_X \phi_2\}}$$

$$\frac{\begin{array}{c}[\textbf{VERIFY-METH}]\\ W{=}\{v_i\}_{i=1}^{n}\quad Z{=}\{\mathbf{res}, v'_1, .., v'_{m-1}\}\\ \phi_{pr}{=}\exists Z\cdot\phi_{po}\quad \vdash \{\phi_{pr} \wedge \mathit{nochange}(W)\}\ e\ \{\phi\}\end{array}}{\vdash t_0\ mn((\mathtt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \mathtt{where}\ \phi_{po}\ \{e\}}$$

The [**VERIFY−CALL**] rule checks that the precondition of each method call can be verified as *statically safe* by the current program state. If it cannot be proven statically safe, a run-time test will be inserted prior to the call site to guarantee the safety of the precondition during program execution. The precondition derived for recursive methods is meant to be also satisfied recursively. The [**VERIFY−METH**] rule ensures that each of its callees is either statically safe or has a runtime test inserted. The [**VERIFY−WHILE**] rule uses X to denote the free variables appearing in the loop body; the substitution ρ maps the unprimed to primed variables. This rule uses the loop formula ϕ_2 to compute a precondition ϕ_{pr} necessary for the correct execution of the loop body. The precondition is checked for satisfiability using ϕ, the state at the beginning of the loop. We refer to this new set of rules as *forward verification* rules. We define a special class of *totally-safe* programs, as follows:

Definition 1 (Totally-Safe Program). *A method is said to be* totally-safe *if the precondition derived from all calls in its method's body can be verified as statically safe. A program is* totally-safe *if all its methods are* totally-safe.

For each totally-safe program, we can guarantee that it never encounters any runtime error due to unsatisfied preconditions.

4 Computing Disjunctive Fixed Points

Classical fixed point analysis technique in the polyhedron domain [9] attempts to obtain a conjunctive formula result with the help of convex-hull and widening operators. A challenge for disjunctive fixed point inference is to apply *selective hulling* on closely related disjuncts whenever needed.

In this paper, we propose a qualitative measure called *affinity* to determine the suitability of two formulae for hulling. In order to obtain the affinity between two terms ϕ_1 and ϕ_2, we have to compute two main expressions (i) $\phi_{hull} = \phi_1 \oplus \phi_2$ and (ii) $\phi_{diff} = \phi_{hull} \wedge \neg(\phi_1 \vee \phi_2)$. Furthermore, we also require a heuristic function *heur* that indicates how closely related is the approximation $\phi_1 \oplus \phi_2$ from the original formula $\phi_1 \vee \phi_2$. With this, we can formally define the affinity measure using:

Definition 2 (Affinity Measure). *Given a function heur that returns a value in the range 1..99, the affinity measure can be defined as:*

$$\begin{array}{ll} \mathit{affin}(\phi_1, \phi_2) =_{\mathtt{df}} & \mathtt{if}\ \phi_{diff}{=}\mathtt{false}\ \mathtt{then}\ 100 \\ & \mathtt{else\ if}\ \phi_{hull}{=}\mathtt{true}\ \mathtt{then}\ 0 \\ & \mathtt{else}\ \mathit{heur}(\phi_1, \phi_2) \end{array}$$

The precise extreme (100) indicates that the convex-hull operation is exact without any loss of precision. The imprecise extreme (0) indicates that the convex-hull operation is inexact and yields the weakest possible formula `true`. In between these two extremes, we will use an affinity measure to indicate the closeness of the two terms by returning a value in the range 1..99.

Selective Hull based on Planar Affinity. The planar affinity measure computes the fraction of planes from the geometrical representation of the original formula that are preserved in the hulled approximation:

Definition 3 (Planar Affinity Measure). *Given two disjuncts ϕ_1, ϕ_2 and the convex-hull approximation $\phi_{hull} = \phi_1 \oplus \phi_2$, we first define the set of conjuncts $mset{=}\{c \in (\phi_1 \cup \phi_2) \mid \phi_{hull} \Longrightarrow c\}$. The planar affinity measure is shown below:*

$$p\text{-}heur(\phi_1, \phi_2) {=}_{df} (|mset|/|\phi_1 \cup \phi_2| * 98) + 1$$

The denominator $|\phi_1 \cup \phi_2|$ represents the number of planes corresponding to the original formulae (from both polyhedra ϕ_1 and ϕ_2). Some of these planes are approximated by the hulling process, while others are preserved in the approximation ϕ_{hull}. The number of preserved planes is represented by the cardinality of *mset* and indicates the suitability of the two disjuncts for hulling.

As an example, consider the following disjunctive formula:

$$\phi = (x{\leq}0 \wedge x'{=}x) \vee (x{=}1 \wedge x'{=}0) \vee (x{=}2 \wedge x'{=}0)$$

Firstly, the three disjuncts (denoted respectively by d_1, d_2 and d_3) are converted to a minimal form. As with other operators on polyhedra (e.g. the standard widening operator from [15]), the minimal form requires that no redundant conjuncts are present and, furthermore, each equality constraint is broken into two corresponding inequalities as follows:

$$\begin{aligned} d_1 &= (x{\leq}0 \wedge x'{\geq}x \wedge x'{\leq}x) \\ d_2 &= (x{\geq}1 \wedge x{\leq}1 \wedge x'{\geq}0 \wedge x'{\leq}0) \\ d_3 &= (x{\geq}2 \wedge x{\leq}2 \wedge x'{\geq}0 \wedge x'{\leq}0) \end{aligned}$$

We compute three affinity values, one for each pair of disjuncts from ϕ. Note that the cardinality of the set of conjuncts $(\phi_1{\cup}\phi_2)$ is considered after removing duplicate conjuncts that appear both in ϕ_1 and ϕ_2.

$$\begin{array}{ll} d_1 \oplus d_2 = (x'{\leq}x \wedge x'{\leq}0 \wedge x'{\leq}x{-}1) & mset(d_1, d_2) = \{x'{\leq}x, x{\leq}1, x'{\leq}0\} \\ & p\text{-}heur(d_1, d_2) = 3/7 * 98 + 1 = 43 \\ d_1 \oplus d_3 = (x'{\leq}x \wedge x'{\leq}0 \wedge x'{\leq}x{-}2) & mset(d_1, d_3) = \{x'{\leq}x, x{\leq}2, x'{\leq}0\} \\ & p\text{-}heur(d_1, d_3) = 3/7 * 98 + 1 = 43 \\ d_2 \oplus d_3 = (x{\geq}1 \wedge x{\leq}2 \wedge x'{\leq}0 \wedge x'{\leq}0) & mset(d_2, d_3) = \{x{\geq}1, x{\leq}2, x'{\leq}0, x'{\leq}0\} \\ & p\text{-}heur(d_2, d_3) = 4/6 * 98 + 1 = 66 \end{array}$$

Based on these affinities, the most related pair of disjuncts is $\{d_2, d_3\}$. The result for selective hull of ϕ will therefore capture a precise relation between x and x':

$$\oplus_2\phi = \oplus_2\{d_1, d_2{\oplus}d_3\} = (x{\leq}0 \wedge x'{=}x) \vee (x{\geq}1 \wedge x{\leq}2 \wedge x'{=}0)$$

Selective Hull based on Hausdorff distance. Related to our affinity measure, Sankaranarayanan et al [24] have recently introduced a heuristic function that uses the Hausdorff distance to measure the distance between the geometrical representations of two disjuncts.

The Hausdorff distance is a commonly used measure of distance between two sets. Given two polyhedra, P and Q, their Hausdorff distance can be defined as: $h\text{-}heur(P, Q){=}_{df} max_{x\in P}\{min_{y\in Q}\{d(x, y)\}\}$ where $d(x, y)$ is the Euclidian distance between two points x and y. This heuristic was deemed as hard to compute in [24] and, as an alternative, a range-based Hausdorff heuristic was used.

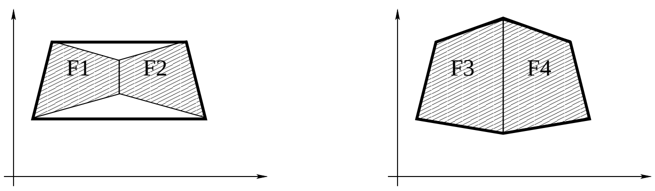

Fig. 3. Pairs of disjuncts with similar Hausdorff distance

Because it reduces the information about variables to non-relational ranges, we can argue that a range-based heuristic is less suitable for a relational abstract domain like the polyhedron domain. Furthermore, we present an intuitive argument why such a distance based heuristic is less appropriate. The pairs of disjuncts {F1,F2} and {F3,F4} from Fig. 3 may have similar *h-heur* values; on the other hand, the affinity based on *p-heur* precisely indicates that the second pair {F3,F4} is more suited for hulling. In the Sect. 5, we compare these two heuristic functions when inferring postconditions for a suite of benchmark programs.

4.1 Powerset Widening Operator

The standard widening operator for the convex polyhedron domain was introduced in [9]. For disjunctive fixed point inference, a (powerset) widening operator for sets of polyhedra is required. Given two disjunctive formulae ϕ_1 and ϕ_2, the challenge is to find pairs of related disjuncts $\{d_i, e_i\}$ ($d_i{\in}\phi_1$, $e_i{\in}\phi_2$) such that the result of widening d_i wrt e_i is as precise as possible.

For this purpose, Bagnara et al [1] introduced a framework to lift a widening operator over a base domain to a widening operator over its powerset domain. The strategy used by the powerset widening based on a connector starts by joining (connecting) elements in ϕ_2 to ensure that each such connected element approximates some element from ϕ_1. Secondly, it chooses related pairs $\{d_i, e_i\}$ based on the logical implication relation, where $d_i \Rightarrow e_i$. Mostly concerned with convergence guarantees for widening operators, the framework from [1] does not give a recipe for defining connector operators able to find related disjuncts. Later, the generic widening operator definition was instantiated for disjunctive polyhedral analysis by Gulavani et al in [13]. However, their proposal uses a connector operator that relies on the ability to find one minimal element from a set of polyhedra. In general, the most precise result cannot be guaranteed by a deterministic algorithm, since the polyhedron domain is partially ordered. To overcome this problem, we propose an affinity measure to find related disjuncts for pairwise widening.

The strategy that we adopt for widening is to choose related pairs $\{d_i, e_i\}$ based on their affinity. After pairwise widening, we subject the result to a selective hull operation provided it contains more disjuncts than ϕ_1. In general, there may be more disjuncts in ϕ_2 than in ϕ_1. A reason for non-convergence of the powerset widening operator is that some element from ϕ_2 is not involved in any widening computation and included unchanged in the result. Our operator (similar to the connector-based widening) distributes each disjunct from the arguments ϕ_1 and ϕ_2 in a widening computation and thus ensures convergence.

Formally, given two disjunctive formulae $\phi_1{=}\bigvee_{i=1}^{m} d_i$ and $\phi_2{=}\bigvee_{i=1}^{n} e_i$, we define a powerset widening operator ∇_m as follows: $\phi_1 \nabla_m \phi_2 = \oplus_m\{d_i \nabla e_i | d_i{\in}\phi_1, e_i{\in}\phi_2\}$, where d_i is the best match for widening e_i as found by the *widen_affin* measure. Similar to the affinity from Def. 2, the widen-affinity aims to find related disjuncts, but proceeds by indicating how closely related is the approximation $\phi_1 \nabla \phi_2$ from the original formula ϕ_1.

$$\begin{aligned} \mathit{widen\_affin}(\phi_1, \phi_2) = \;& \texttt{if } (\phi_1 \nabla \phi_2) \wedge \neg\phi_1 {=} \texttt{false then 100} \\ & \texttt{else if } (\phi_1 \nabla \phi_2) {=} \texttt{true then 0} \\ & \texttt{else } \mathit{heur}(\phi_1, \phi_2) \end{aligned}$$

The planar affinity measure from Def. 3 can be used for widening, provided we redefine *mset* to relate ϕ_1, ϕ_2 with the approximation $\phi_{widen} = \phi_1 \nabla \phi_2$ as follows:

$$\mathit{mset} = \{c \in (\phi_1 {\cup} \phi_2) | \phi_{widen} \Rightarrow c\}$$

5 Experiments

We have implemented the proposed inference mechanisms with the goal of analyzing imperative programs. Our implementation includes a pre-processing phase

Benchmark Programs	Source (lines)	Recursive constraints	m=1 (secs)	m=2 (secs)	m=2 post	m=3 (secs)	m=3 post	m=4 (secs)	m=4 post	m=5 (secs)	m=5 post
binary search	31	1	0.44	1.02	1	-	-	-	-	-	-
bubble sort	39	2	0.78	0.89	1	-	-	-	-	-	-
init array	5	1	0.17	0.24	1	-	-	-	-	-	-
merge sort	58	3	1.42	3.39	3	3.76	1	3.91	1	4.48	1
queens	39	2	1.89	2.41	2	2.48	1	-	-	-	-
quick sort	43	2	0.63	1.51	2	1.70	1	-	-	-	-
FFT	336	9	8.24	10.17	5	11.62	3	11.90	1	12.15	1
LU Decomp.	191	10	10.27	13.41	8	14.44	3	-	-	-	-
SOR	84	5	1.46	2.41	3	3.49	1	3.64	1	-	-
Linpack	903	25	28.14	33.23	20	35.04	2	-	-	-	-

Fig. 4. Statistics for postcondition inference. Timings include precondition verification. ("-" signifies a time or post similar to those from the immediate lower value of m)

to convert each C-like input program to our core language. The entire prototype system was built using Glasgow Haskell compiler [16] extended with the Omega constraint solving library [23]. Our test platform was a Pentium 3.0 GHz system with 2GBytes main memory, running Fedora 4.

We tested our system on a set of small programs with challenging recursion, and also the Scimark and Linpack benchmark suites [21, 10]. Figure 4 summarizes the statistics obtained for each program. To quantify the analysis complexity of the benchmark programs, we counted the program size (column 2) and also the number of recursive methods and loops present in each program (column 3).

The main objective for building this prototype was to certify that the disjunctive analysis can be fully automated and that it gives more precise results compared to a conjunctive analysis. To this end, we experimented with different bounds on the number m of disjuncts allowed during fixed point analysis. For each value of m, we measured the analysis time and the number of methods for which the postcondition was *more precise* than using $(m-1)$ disjuncts. For each analyzed program, we detected a *bound* on the value of m: increasing m over this bound does not yield more precision for the formulae. The analysis time remains constant for cases where m is bigger than this bound, therefore the values beyond these bounds are marked with "-". Capturing a precise postcondition for algorithms like binary search, bubble sort, or init array was done with a value of m equal to 2. We found that queens and quick sort require 3 disjuncts, while merge sort can be inferred by making use of 5 disjuncts.

We also evaluated the usefulness of the disjunctive fixed point inference for static array bound check elimination. The results are summarized in the Fig. 5. Column 2 presents the total number of checks (counted statically) that are present in the original programs. Columns 3 and 5 present the number of checks that cannot be proved safe by using conjunctive analysis (m=1) and, respectively, disjunctive analysis with m=5 and planar affinity. For comparison, column 4 shows results of analysis using the Hausdorff distance heuristic, where the number of checks not proven is greater than using planar affinity.

Using the planar affinity, the two programs bubble sort and init array were proven totally safe with 2-disjunctive analysis. Merge sort and SOR exploited the precision of 4-disjunctive analysis for total check elimination. Even if not all the checks could be proven safe for queens, quick sort, FFT, LU and Linpack benchmarks, the number of potentially unsafe checks decreased gradually, for analyses with higher values of m. As a matter of fact, our focus in this paper was to infer precise postconditions and we relied on a simple mechanism to derive preconditions. To eliminate more checks, we could employ the technique of [3] which is powerful enough to derive sufficient preconditions and eliminate all checks in this set of benchmarks [26]. However, we stress that, either kind of prederivation we use, disjunctive analysis is needed for better check elimination.

Benchmark Programs	Static Chks.	Conj. m=1	Haus. m=5	Plan. m=5
binary search	2	2	2	2
bubble sort	12	3	0	0
init array	2	2	0	0
merge sort	24	9	4	0
queens	8	4	2	2
quick sort	20	5	5	1
FFT	62	17	12	5
LU Decomp.	82	42	9	4
SOR	32	15	2	0
Linpack	166	92	65	52

Fig. 5. Statistics for check elimination

In general, analysis with higher values for m has the potential of inferring more precise formulae. The downside is that computing the affinities of m disjuncts is an operation with quadratic complexity in terms of m and may become too expensive for higher values of m. In practice, we found that the case (m=3) computes formulae sufficiently precise, with a reasonable inference time.

6 Related Work

Our analysis is potentially useful for software verification and for static analyses based on numerical abstract domains.

Program verification may be performed by generating verification conditions, where their validity implies that the program satisfies its safety assertions. Verification condition generators assume that loop invariants are present in the code, either annotated by the user or inferred automatically. Methods for loop invariant inference include the induction-iteration approach [25] and approaches based on predicate abstraction [11, 17]. Leino and Logozzo [18] designed a loop invariant computation that can be invoked on demand when an assertion from the analyzed program fails. The invariant that is inferred satisfies only a subset of the program's executions on which the assertion is encountered. Comparatively, our method infers a disjunctive formula that is valid for all the program's executions, with each disjunct covering some related execution paths. We achieve this modularly, regardless of any subsequent assertions. Thus, our results can be directly used in the inter-procedural setting.

Partitioning of the abstract domain was first introduced in [5]. Recently, Mauborgne and Rival [19] have given strategies for partition creation and demonstrated their feasibility through their use in ASTRÉE static analyzer [2]. Like them, we make the choice of which disjunctions to keep at analysis time. However, the partitioning criterion is different. In their case, the control flow is used to choose which disjunctions to keep. Specifically, a token representing some conditions on the execution flow is attached to a disjunct, and formulae with similar tokens are hulled together. In our case, the partitioning criterion is based on a property of the disjuncts themselves, with the affinity measure aiming to hull together the most closely related disjuncts.

Various abstract numerical domains have been developed for static analysis based on abstract interpretation. The form of invariants to be discovered is determined by the chosen numerical domain: from the interval domain that is able to discover relations of the form $(\pm x \leq c)$, to the lattice of polyhedra that represents invariants of the form $(a_1x_1+..+a_nx_n \leq c)$, all these abstract domains represent conjunctions of linear inequalities. Our pre/post analysis is formalised in a manner that is independent of the abstract domain used. It can therefore readily benefit from advances in constraint solving techniques for these numerical domains.

7 Conclusion

We have proposed a new method for inferring disjunctive postconditions. Our approach is based on the notion of selective hulling as a means to implement adjustable precision in our analysis. We introduced a simple but novel concept called affinity and showed that planar affinity is superior to a recently introduced method based on Hausdorff distance. We have built a prototype system for disjunctive inference and have proven its correctness in the technical report [22]. Our experiments demonstrate the utility of the disjunctive postconditions for proving a class of runtime checks safe at compile-time, and the potential for tradeoff between precision and analysis cost.

Acknowledgements: This work benefited much from research discussions with Siau-Cheng Khoo and Dana Xu from an earlier project on array bounds inference. The authors would like to acknowledge two donation grants from Microsoft Singapore and Microsoft Research Asia, and the support of A*STAR research grant R-252-000-233-305.

References

[1] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Widening operators for powerset domains. In *VMCAI*, pages 135–148, 2004.

[2] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.

[3] Wei-Ngan Chin, Siau-Cheng Khoo, and Dana N. Xu. Deriving pre-conditions for array bound check elimination. In *PADO*, pages 2–24, 2001.

[4] Christopher Colby and Peter Lee. Trace-based program analysis. In *POPL*, pages 195–207, 1996.

[5] Patrick Cousot. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, 1981.

[6] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130, 1976.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[8] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[9] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[10] Jack Dongarra, Piotr Luszczek, and Antoine Petitet. The Linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[11] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.

[12] Roberto Giacobazzi and Francesco Ranzato. Optimal domains for disjunctive abstract intepretation. *Sci. Comput. Program.*, 32(1-3):177–210, 1998.

[13] Bhargav S. Gulavani and Sriram K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS*, 2006.

[14] Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In *PADO*, pages 63–83, 2001.

[15] Nicholas Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme.* Thèse de $3^{\text{ème}}$ cycle d'informatique, Université scientifique et médicale de Grenoble, Grenoble, France, March 1979.

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

[17] Shuvendu K. Lahiri and Randal E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV*, pages 135–147, 2004.

[18] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *APLAS*, pages 119–134, 2005.

[19] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, pages 5–20, 2005.

[20] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO*, pages 155–172, 2001.

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

[22] Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. Technical report. http://www.comp.nus.edu.sg/~corneliu/research/disjunctive.tr.pdf.

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

[24] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *SAS*, 2006.

[25] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *POPL*, pages 132–143, 1977.

[26] Dana N. Xu, Corneliu Popeea, Siau-Cheng Khoo, and Wei-Ngan Chin. A modular type inference and specializer for array bound checks elimination (under preparation). Technical report. http://www.comp.nus.edu.sg/~corneliu/research/array.pdf.