Exact vs. Approximate Array Region Analyses

Béatrice CREUSILLET and François IRIGOIN*
Centre de Recherche en Informatique, École des mines de Paris
35, rue Saint-Honoré, F-77305 FONTAINEBLEAU Cedex FRANCE

April 30, 1996

Abstract

Advanced program optimizations, such as array privatization, require precise array data flow analyses, usually relying on conservative over- (or *may*) and under- (or *must*) approximations of array element sets [26, 32, 21]. In a recent study [14], we proposed to compute *exact* sets whenever possible. But the advantages of this approach were still an open issue which is discussed in this paper.

It is first recalled that must array region analyses cannot be defined on lattices. It implies that there exists no *better* solution for such data flow problems, and that ad-hoc solutions must be defined.

For that purpose, it is suggested to perform under- and over-approximate analyses at the same time, and to enhance the results of must analyses with those of may analyses, when the latter can be proved exact according to an *exactness criterion*. This is equivalent to our previous approach, and is more effective than using only traditional techniques such as widening and narrowing operators which may fail to expose exact solutions even though their computability is decidable. This method is very general and could be applied to other types of analyses.

Keywords: Array region analysis, semantical analysis, exactness.

1 Introduction

Several studies [18, 4] have highlighted the need for advanced program optimizations to deal with memory management issues when compiling programs for massively parallel machines or hierarchical memory systems. For instance, BLUME and EIGENMANN [4] have shown that array privatization could greatly enhance the amount of potential parallelism in sequential programs. This technique basically aims at discovering array sections that are used as temporaries in loops, and can thus be replaced by local copies on each processor. An array section is said to be privatizable in a loop if each read of an array element is preceded by a write in the same iteration, and several different iterations may access each privatized array element [26, 33]. Solving such problems requires a precise intra- and inter-procedural analysis of array data flow.

*E-mail: {creusillet,irigoin}@cri.ensmp.fr

Several algorithms for array privatization or array expansion (a similar technique for shared memory machines) have already been proposed [16, 27, 26, 33, 22, 1, 14], based on different types of array data flow analyses. The first approach [16, 27] performs an *exact* analysis of array data flow, but for a restricted source language[1]. Most of the other methods use conservative approximations of array element sets, such as *MayBeDefined* and *MustBeDefined* sets. In fact, may sets are over-approximations of exact solutions, while must sets are under-approximations, according to a predefined approximation ordering. On the contrary, the last approach [1, 14] tries to compute exact solutions whenever possible, switching to conservative approximations only when exactness cannot be preserved anymore. However, except for specific applications [7, 8] requiring the knowledge of exactness, the advantages of our approach were still an open issue, which is discussed in this paper.

Traditionally, semantical analyses are defined on lattices. This ensures that they be precisely defined even in case of recursive semantical equations. For instance, for array region analyses, exact solutions belong to the lattice $(\wp(\mathbb{Z}^n), \varnothing, \mathbb{Z}^n, \cup, \cap)$. Over-approximate analyses of array regions are also defined on lattices: lattice of convex polyhedra in PIPS [25] (our interprocedural parallelizer of scientific programs), or regular section lattice in the case of RSDs [6] for instance. However, as will be shown in Section 2.3, under-approximate solutions do not belong to a predefined lattice. This is due to the fact that, given an exact array region, no *best* under-approximation can be defined. This is illustrated in Figure 1 where several possible under-approximations of an array region are provided. In this case, BOURDONCLE [5] suggests to approximate fix-points by decreasing under-approximate iterations[2], using a *narrowing operator*[3]. However, this approach may fail to compute under-approximations equal to the corresponding exact solutions while the computation of the latter is decidable, which is possible as shown in [1, 14].

We thus propose another approach, based on our previous experience of exact array dataflow analyses [14, 13, 15]: We suggest to perform under- and over-approximate analyses at the same time, and to enhance the results of must analyses with those of may analyses, when the latter can be proved exact according to an *exactness criterion*.

The paper is organized as follows. Section 2 presents the semantical analysis framework, and shows that some analysis solutions may not belong to a predefined lattice; traditional ad-hoc solutions are also presented. Our solution and the *exactness criterion* are presented in Section 3 in relation with our previous work.

2 Semantical Analysis Frameworks

The semantical analysis of a program aims at statically discovering its properties, or the properties of the objects it uses (variables, ...). However, semantical equations may not have solutions computable in finite time; this is the case of recursive functions in an infinite height lattice for instance. Moreover, exactly

[1] monoprocedural, *static* control language.

[2] He also proposes a technique called *dynamic partitioning*, but its complexity would be too high in the case of array region computations.

[3] More precisely the dual of his widening operator.

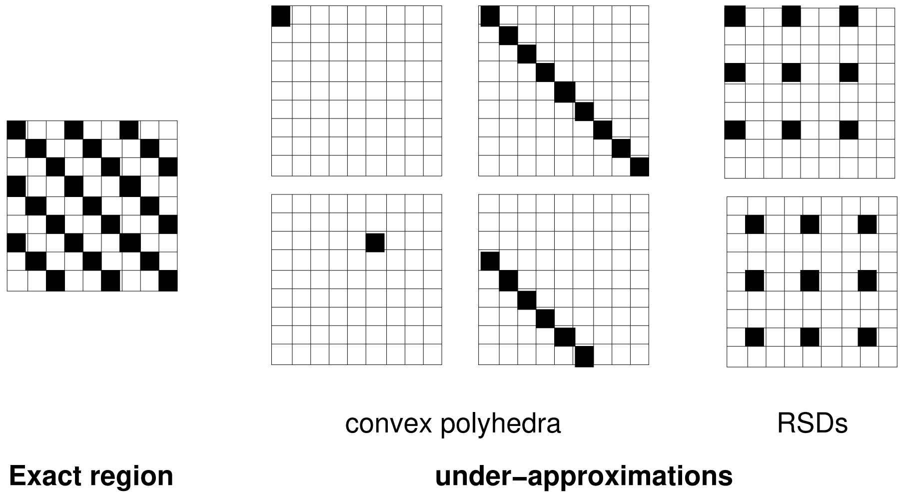

Figure 1: Possible under-approximations

describing the behavior of any program written in a given language may happen to be impossible, or at least of an overwhelming complexity; most languages have I/Os instructions whose effects are not completely foreseeable; representing all types of expressions, particularly real expressions, is also generally considered too complex for an average compiler. Facing these obstacles, the solution is either to restrict the input language (as in [16, 17, 29, 27]), or to perform conservative approximate analyses [9].

This section describes some usual characteristics of exact and approximate semantical analysis frameworks. It is then shown that some analyses do not respect these properties. Usual solutions are finally presented in the last subsection, and their limitations are discussed.

2.1 Sets and domains

Traditionally, mathematical spaces used for the denotation of programming constructs are called *semantic domains* and are *complete partial orders* or *lattices*.

Definition 2.1 A *complete partial order* (or cpo) is a set $\boldsymbol{E}$ partially ordered by a relation $\sqsubseteq_{\boldsymbol{E}}$; it has a least element $\perp_{\boldsymbol{E}}$ ($\boldsymbol{E}$ is *inductive*); and every countable increasing sequence $e_0 \sqsubseteq \cdots \sqsubseteq e_n \sqsubseteq \ldots$ has a limit in $\boldsymbol{E}$, denoted by $\bigsqcup_i e_i$.

Examples of basic domains are boolean values, natural integers $\mathbb{N}$, or relative integers $\mathbb{Z}$ (all of them augmented with an undefined element $\perp$). More complex domains are powerset domains, product domains, sum domains or continuous function domains[4].

Definition 2.2 A *lattice* is a cpo in which any two elements x and y have a greatest lower bound ($x \vee y$) and a least upper bound ($x \wedge y$).

[4] For a more complete description see [28, 20].

The advantage of using cpo's as semantic domains is that the semantics of any recursive continuous semantical equation $f : \boldsymbol{D} \longrightarrow \boldsymbol{E}$ can be defined as the *least fixed point* of a continuous functional $\Phi : (\boldsymbol{D} \longrightarrow \boldsymbol{E}) \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{E})$.

Definition 2.3 Given cpo's $\boldsymbol{D}$ and $\boldsymbol{E}$, and a function $g : \boldsymbol{D} \longrightarrow \boldsymbol{E}$,
g is *monotone* if $x \sqsubseteq_{\boldsymbol{D}} y \implies g(x) \sqsubseteq_{\boldsymbol{E}} g(y)$.
g is *continuous* if $\forall\, e_0 \sqsubseteq \cdots \sqsubseteq e_n \sqsubseteq \ldots \quad g(\bigsqcup_i e_i) = \bigsqcup_i g(e_i)$.

Definition 2.4 Let $\boldsymbol{D}$ be a cpo, and $g : \boldsymbol{D} \longrightarrow \boldsymbol{D}$ a continuous function.
$d \in \boldsymbol{D}$ is called a *fixed point* of g if $g(d) = d$. Moreover, d is called the *least fixed point* of g if $\forall\, d',\ g(d') = d' \implies d \sqsubseteq_{\boldsymbol{D}} d'$.

Theorem 2.1 *Let $\boldsymbol{D}$ be a cpo, and $g : \boldsymbol{D} \longrightarrow \boldsymbol{D}$ a continuous function. Then g has a least fixed point* $\mathrm{lfp}(g) = \bigsqcup_{i \geq 0} g^i(\perp)$.

In our framework, we require that exact semantical equations be continuous functions. This ensures that the exact semantic of the analysis be precisely defined. However, approximate semantical equations are not required to be continuous functions. This might be rather cumbersome in case of recursive definitions (which are usually needed to handle loops), since the existence of a fixed point is no more guaranteed. When the functions meet the weaker condition of monotonicity, Graham and Wegman [19] define an *acceptable* solution. However, as shown in Subsection 2.3, some approximate analysis solutions do not belong to predefined lattices, and recursive functions may not even be monotone.

2.2 Semantical functions

Since we are interested in the semantical analysis of program properties, semantical functions share the same pattern. Let $\mathcal{A}$ be an exact semantical analysis of a program written in a programming language $\boldsymbol{\mathcal{L}}$. $\mathcal{A}$ is generally a function from a subset $\tilde{\boldsymbol{\mathcal{L}}}$ of $\boldsymbol{\mathcal{L}}$ to another set, usually a function set:

$$\begin{aligned} \mathcal{A} : \tilde{\boldsymbol{\mathcal{L}}} &\longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}) \\ l &\longrightarrow \mathcal{A}[\![l]\!] \end{aligned}$$

For instance, the exact semantics of array region analysis is described by a function $\mathcal{R} : \tilde{\boldsymbol{\mathcal{L}}} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \wp(\mathbb{Z}^n))$, where $\boldsymbol{\Sigma}$ is the set of memory stores.

As stated in the introduction of this section, the solutions defined by $\mathcal{A}$ may be extremely difficult, indeed impossible, to compute. Hence the idea to replace $\mathcal{A}$ by approximate analyses, comparable to $\mathcal{A}$ by a partial order $\sqsubseteq$:

$$\begin{aligned} \overline{\mathcal{A}} \text{ is an over-approximation of } \mathcal{A} &\iff \mathcal{A} \sqsubseteq \overline{\mathcal{A}} \\ \underline{\mathcal{A}} \text{ is an under-approximation of } \mathcal{A} &\iff \underline{\mathcal{A}} \sqsubseteq \mathcal{A} \end{aligned}$$

This *approximation ordering* [11] is a logical ordering, and is not necessarily related to a partial ordering between semantic values. However, since $\mathcal{A}$, $\overline{\mathcal{A}}$ and

$\underline{\mathcal{A}}$ are defined on the same sub-domain of $\boldsymbol{\mathcal{L}}$, it is equivalent to an ordering of the solution functions:

$$\mathcal{A} \sqsubseteq \overline{\mathcal{A}} \iff \forall\, l \in \tilde{\boldsymbol{\mathcal{L}}} \quad \mathcal{A}[\![l]\!] \sqsubseteq \overline{\mathcal{A}}[\![l]\!]$$

Given this approximation ordering, a semantical analysis $\mathcal{A}$ can be surrounded by two new, and hopefully computable, approximate analyses: an over-approximation, denoted by $\overline{\mathcal{A}}$; and an under-approximation, denoted by $\underline{\mathcal{A}}$. The target sets of $\underline{\mathcal{A}}$ and $\overline{\mathcal{A}}$ are not necessarily the same as the target set of $\mathcal{A}$.

> For instance, array element index sets are parts of $\mathbb{Z}^n$. Due to the complexity of being able to represent any part of $\mathbb{Z}^n$, several compact representations have been defined: convex polyhedra [30], RSD [6], DAD [2], ...

The approximate analyses we are interested in are then such that:

$$\overline{\mathcal{A}} : \tilde{\boldsymbol{\mathcal{L}}} \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}') \quad \text{and} \quad \underline{\mathcal{A}} : \tilde{\boldsymbol{\mathcal{L}}} \longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}'')$$

2.3 Non-lattice frameworks: Why?

A natural question that arises is why we may need non-continuous, or even non-monotone functions, while very friendly data-flow frameworks have been so carefully designed for years. The answer is that, in some cases, it is not possible to define interesting approximations otherwise.

For instance, let us consider exact array region analysis:

$$\mathcal{R} : \tilde{\boldsymbol{\mathcal{L}}} \longrightarrow (\boldsymbol{\Sigma} \longrightarrow \wp(\mathbb{Z}^n))$$

Being able to represent any possible part of $\mathbb{Z}^n$ is generally considered too complex, and the representation is often limited to particular parts of $\mathbb{Z}^n$ (convex polyhedra, RSDs, DADs, lists of convex polyhedra with a maximum length, ...).

In the case of convex polyhedra ($\wp_{\text{convex}}(\mathbb{Z}^n)$), an over-approximate analysis framework can be obtained from the lattice $(\wp(\mathbb{Z}^n), \varnothing, \mathbb{Z}^n, \cup, \cap)$ by a simple upper closure[5] $\overline{\rho} = \lambda x.\, \text{convex\_hull}(x)$ (see Figure 2). This ensures that the approximate semantical equations derived from their exact counterparts using $\overline{\rho}$ are continuous functions.

However, there exists no interesting lower closure $\underline{\rho}$ (that is to say $\underline{\rho} \neq \lambda x.\varnothing$ and $\underline{\rho} \neq \lambda x.x$) from $(\wp(\mathbb{Z}^n), \varnothing, \mathbb{Z}^n, \cup, \cap)$ to $\wp_{\text{convex}}(\mathbb{Z}^n)$. If such a closure existed, $(\wp_{\text{convex}}(\mathbb{Z}^n), \varnothing, \mathbb{Z}^n, \cup, \lambda xy.\underline{\rho}(x \cap y))$ would be a complete lattice (see [9], Corollary 2.3.0.5); this is not possible, because $\wp_{\text{convex}}(\mathbb{Z}^n)$ is not closed under $\cup$.

The same conclusion holds for RSDs, DADs, and lists of convex polyhedra with a maximum length. It intuitively corresponds to the fact that the greatest convex polyhedron or RSD contained into any array region is not uniquely defined. An ad-hoc computable under-approximation must then be chosen. Some possible choices have already been described in [5]. This is the subject of the next subsection.

[5] An upper closure $\overline{\rho}$ is an internal operator, which is monotone ($x \subseteq y \implies \overline{\rho}(x) \subseteq \overline{\rho}(y)$), extensive ($\lambda x.x \sqsubseteq \overline{\rho}$), and idempotent ($\overline{\rho} \circ \overline{\rho} = \overline{\rho}$). A lower closure is reductive ($\underline{\rho} \sqsubseteq \lambda x.x$).

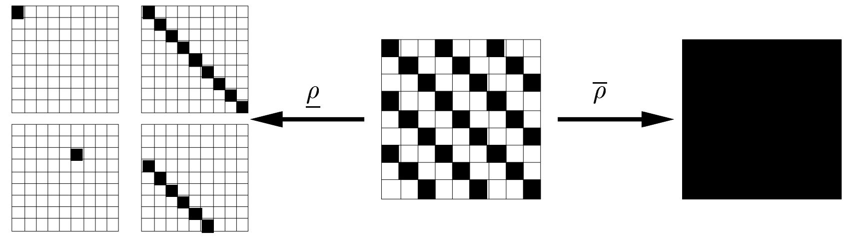

Figure 2: Upper and under closures

2.4 Usual ad-hoc solutions

When the approximate solution space $\boldsymbol{A}''$ is a lattice or cpo, its relations with the exact solution space $\boldsymbol{A}$ can be described by a GALOIS connection [3, 10], which defines two monotone functions: An *abstraction* function $\alpha : \boldsymbol{A} \longrightarrow \boldsymbol{A}''$, and a *concretization* or *meaning* function $\gamma : \boldsymbol{A}'' \longrightarrow \boldsymbol{A}$ (see Figure 3). Least fixed points can then be safely approximated by successive iterations relying on sequences of *narrowing* operators in the case of under-approximations [10, 9].

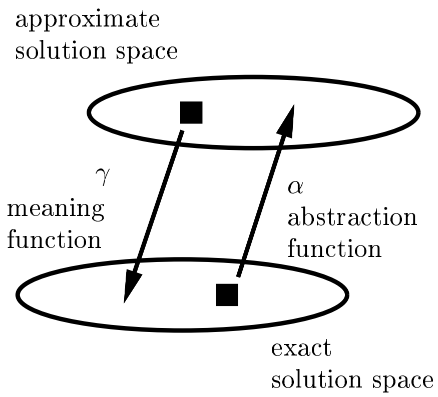

Figure 3: GALOIS Connection

To handle cases where the approximate solution space is not a lattice or a cpo, but a mere partial order, BOURDONCLE [5] generalizes the traditional approach of GALOIS connections by defining *representations*, in which α is not required to be a monotone function.

Definition 2.5 (Under-representation) Let $(\boldsymbol{A}, \perp, \sqsubseteq)$ be a cpo, $\boldsymbol{A}''$ a set, and $\gamma : \boldsymbol{A}'' \longrightarrow \boldsymbol{A}$ a *meaning* function. $(\boldsymbol{A}'', \preceq, \gamma, \underline{\Delta} = (\underline{\Delta}_i)_{i \in \mathbb{N}})$ is an *under-representation* of $\boldsymbol{A}$ if:

1. $(\boldsymbol{A}'', \perp, \preceq)$ is a partial order.

2. γ, the *meaning* or *concretization* function is monotone.

3. Each element $a \in \boldsymbol{A}$ can be safely under-approximated by $\alpha(a) \in \boldsymbol{A}''$, that is to say $\gamma(\alpha(a)) \sqsubseteq a$.

4. $\underline{\Delta}$ is a sequence of operators $\underline{\Delta}_i : \boldsymbol{A}'' \times \boldsymbol{A}'' \longrightarrow \boldsymbol{A}''$, such that:

   - $\forall\, \underline{\Delta}_i \in \underline{\Delta}$, $\forall\, a, a' \in \boldsymbol{A}''$, $\left\{ \begin{array}{ccc} a \,\underline{\Delta}_i\, a' & \preceq & a \\ \gamma(a \,\underline{\Delta}_i\, a') & \sqsubseteq & \gamma(a') \end{array} \right.$
   - $\forall\, (a_i)_{i \in \mathbb{N}} \in \boldsymbol{A}''$, the sequence $(a'_i)_{i \in \mathbb{N}}$ defined by:

   $$\left\{ \begin{array}{ccl} a'_0 & = & a_0 \\ a'_{i+1} & = & a'_i \,\underline{\Delta}_i\, a_{i+1} \end{array} \right.$$

   has a lower bound.

BOURDONCLE then shows that if $\Phi : \boldsymbol{A} \longrightarrow \boldsymbol{A}$ is a continuous recursive function, and if $\underline{\Phi} : \boldsymbol{A}'' \longrightarrow \boldsymbol{A}''$ is a safe under-approximation of Φ $(\gamma \circ \underline{\Phi} \sqsubseteq \Phi \circ \gamma)$, then the sequence

$$\left\{ \begin{array}{ccl} a_0 & = & \perp \\ a_{i+1} & = & a_i \,\underline{\Delta}_i\, \underline{\Phi}(a_i) \end{array} \right.$$

is a decreasing sequence whose limit is a safe under-approximation of the least fixed point of Φ.

Again, this method does not give the *best solution* but *one* possible solution. The following example shows its limitations. Our purpose is to show that the solution built using $\underline{\Phi}$ as the under-approximation of the least fixed point of Φ can lead to disappointing approximations, whereas the exact solution is is known to be computable.

Example 2.1 We consider array region analysis, and approximate array element sets by convex polyhedra. With the notations of Definition 2.5, we have: $(\boldsymbol{A}, \perp, \sqsubseteq) = (\boldsymbol{\Sigma} \longrightarrow \wp(\mathbb{Z}^n), \lambda x\sigma.\varnothing, \subseteq)$ and $(\boldsymbol{A}'', \perp, \preceq) = (\boldsymbol{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^n), \lambda x\sigma.\varnothing, \subseteq)$. γ is trivially defined by $\gamma = \lambda x.x$. And $\underline{\Delta}$ by:

$$\forall\, a, a' \in \boldsymbol{A}'', \quad \left\{ \begin{array}{ccl} a \,\underline{\Delta}_0\, a' & = & a' \\ a \,\underline{\Delta}_i\, a' & = & a',\ \forall\, i > 0 \end{array} \right.$$

Thus the under-approximation of the least fixed point of any continuous recursive function is reached in exactly one iteration and is equal to $\underline{\Phi}(\varnothing)$.

For `while` loops where no array element is referenced in the condition, the semantics of read regions is defined by:

$$\mathcal{R}[\![\texttt{while}(C)\ \ S\ \ \texttt{endwhile}]\!] = \text{lfp}\big(\lambda f\sigma.(f \circ \mathcal{T}[\![S]\!] \cup \mathcal{R}[\![S]\!]) \circ \mathcal{E}_c[\![C]\!]\sigma\big) = \text{lfp}(\Phi)$$

where lfp stands for *least fixed point*, $\mathcal{E}_c[\![C]\!]\sigma$ is the result of the evaluation of condition C in the store σ, and $\mathcal{T}[\![S]\!]$ represents the transformation of the memory store by the execution of S.

Φ is then under-approximated by:

$$\underline{\Phi} = \lambda f\sigma.(f \circ \mathcal{T}[\![S]\!] \,\underline{\cup}\, \underline{\mathcal{R}}[\![S]\!]) \circ \mathcal{E}_c[\![C]\!]\sigma$$

where $\underline{\cup}$ is an under-approximation of $\cup$ defined by:

$$x \,\underline{\cup}\, y = \textbf{if } x \cup y \in \boldsymbol{\Sigma} \longrightarrow \wp_{\text{convex}}(\mathbb{Z}^n) \textbf{ then } x \cup y \textbf{ else } \lambda x\sigma.\varnothing$$

For the sake of simplicity, we assume that $\mathcal{E}_c$, $\mathcal{T}$, and $\circ$ do not need to be approximated. Given the previous definition of $\underline{\Delta}$, an under-approximation of $\mathrm{lfp}(\Phi)$ is:

$$\underline{\Phi}(\lambda x.\varnothing) = \lambda\sigma.\underline{\mathcal{R}}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]\sigma$$

This is the under-approximation of the region corresponding to the first iteration of the loop.

Now let us consider the following loop nest:

```
DO I = 1,5
   DO J = 1,5
      A(I) = A(I) + ...
   ENDDO
ENDDO
```

In the innermost loop body, the sole element `A(I)` is referenced. It can be exactly represented, and thus under-approximated, by a convex polyhedron: $\{\phi_1 = \mathtt{I}\}$ where ϕ_1 represents the first dimension index of `A`. The under-approximation obtained using $\underline{\Phi}$ for the innermost loop is the set of elements read during the first iteration, and is thus $\{\phi_1 = \mathtt{I}\}$. It exactly describes the set of array elements read by the five iterations of the `J` loop. Repeating the process for the outermost loop, we then obtain $\{\phi_1 = 1\}$, which is far from the set of array elements actually referenced by the loop nest: $\{1 \leq \phi_1 \leq 5\}$. However, this last set is computable and can even be proved to exactly represent the actual set [14].

The previous example has been chosen for its simplicity, and it could be objected that the chosen representation or operators could be enhanced. But similar examples can be built for more complex representations or sequences of $(\underline{\Delta}_i)_{i \in \mathbb{N}}$) operators. For instance, more iterations could be allowed; in our previous example, five iterations would give the exact result. And if there was a step (as in `DO I = 1,5,2`), representations such as RSDs with bounds and strides [23], or intersection of polyhedra and lattices could be used. But what if the upper bound values are so great that the number of iterations would be overwhelming, or are even unknown in which case the number of iterations is undecidable?

Moreover, since there exists no *best* solution, chosing another representation would only lessen the problem, but would not eliminate it: there would always be cases in which the exact solution is known to be computable, but the under-approximation is not exact. The next section brings a solution to this problem.

3 Approximations and Exactness

The previous section has just shown that under-approximate analyses may fail to expose interesting results. This section describes a method to alleviate this problem, based on our previous experience of array region analyses [15]. It relies on the use of an *exactness criterion* which is introduced in a first subsection. Its optimality and applications are then discussed in Section 3.2. The last subsection gives the relations with our previous work.

3.1 Exactness of approximate analyses

From the definition of $\overline{\mathcal{A}}$ and $\underline{\mathcal{A}}$, we already know that:

$$\forall\, l \in \tilde{\boldsymbol{\mathcal{L}}}, \quad \underline{\mathcal{A}}[\![l]\!] \sqsubseteq \mathcal{A}[\![l]\!] \sqsubseteq \overline{\mathcal{A}}[\![l]\!]$$

An immediate consequence is:

$$(\underline{\mathcal{A}}[\![l]\!] \equiv \overline{\mathcal{A}}[\![l]\!]) \implies (\underline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!] \equiv \overline{\mathcal{A}}[\![l]\!])$$

Hence, if the over-approximation is equal to the under-approximation, then both solutions are equal to the exact solution. However, the converse is not true:

$$(\overline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!]) \not\Rightarrow (\underline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!])$$

A very simple counter-example is given by:

$$\overline{\mathcal{A}} : l \longrightarrow (\boldsymbol{D} \longrightarrow \top)$$
$$\underline{\mathcal{A}} : l \longrightarrow (\boldsymbol{D} \longrightarrow \bot)$$

It's not possible to have $\overline{\mathcal{A}}[\![l]\!] \equiv \underline{\mathcal{A}}[\![l]\!]$.

Thus a trivial way to check if the result of an approximation is exact is to verify that it is equal to the opposite approximation. A question that arises is whether this is the best possible criterion.

3.2 Optimality of the Exactness Criterion

This section discusses the previous question. In particular, the next theorem shows that if proving that $\overline{\mathcal{A}}[\![l]\!] \equiv \underline{\mathcal{A}}[\![l]\!]$ is not the best known criterion for checking their exactness, then two other computable analyses $\overline{\mathcal{A}}'$ and $\underline{\mathcal{A}}'$ can be defined, which are better approximations of $\mathcal{A}$ than $\overline{\mathcal{A}}$ and $\underline{\mathcal{A}}$. A constructive proof is then given, which shows how to build $\overline{\mathcal{A}}'$ and $\underline{\mathcal{A}}'$ from $\overline{\mathcal{A}}$ and $\underline{\mathcal{A}}$. The consequences of this theorem are discussed afterwards.

Theorem 3.1 *If there exists a computable criterion $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$ such that:*

$$\forall\, l \in \tilde{\boldsymbol{\mathcal{L}}},\; C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}(l) \implies \overline{\mathcal{A}}[\![l]\!] \equiv \underline{\mathcal{A}}[\![l]\!]$$

then any computable criterion $C_{\overline{\mathcal{A}}\equiv\mathcal{A}}$ or $C_{\underline{\mathcal{A}}\equiv\mathcal{A}}$ such that:

$$\forall\, l \in \tilde{\boldsymbol{\mathcal{L}}},\; C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l) \implies \overline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!]$$
$$\forall\, l \in \tilde{\boldsymbol{\mathcal{L}}},\; C_{\underline{\mathcal{A}}\equiv\mathcal{A}}(l) \implies \underline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!]$$

is equivalent to $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$, or there exists two computable approximations $\overline{\mathcal{A}}'$ and $\underline{\mathcal{A}}'$ such that $\underline{\mathcal{A}} \sqsubseteq \underline{\mathcal{A}}' \sqsubseteq \mathcal{A} \sqsubseteq \overline{\mathcal{A}}' \sqsubseteq \overline{\mathcal{A}}$ and $C_{\underline{\mathcal{A}}'\equiv\mathcal{A}} \iff C_{\underline{\mathcal{A}}'\equiv\overline{\mathcal{A}}'} \iff C_{\overline{\mathcal{A}}'\equiv\mathcal{A}}$.

Proof

- If $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}(l)$ is true then $\overline{\mathcal{A}}[\![l]\!] \equiv \underline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!]$. It implies that $C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l)$ and $C_{\underline{\mathcal{A}}\equiv\mathcal{A}}(l)$ are also true.

- On the contrary, let us assume that there exists $l \in \tilde{\boldsymbol{\mathcal{L}}}$ such that $C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l)$ is true and $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}(l)$ is false.

  $C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l)$ implies that $\mathcal{A}[\![l]\!]$ is computable and is equal to $\overline{\mathcal{A}}[\![l]\!]$. Then, we can define a new approximation of $\mathcal{A}$, $\underline{\mathcal{A}}'$ by:

  $$\begin{aligned} \underline{\mathcal{A}}' : \tilde{\boldsymbol{\mathcal{L}}} &\longrightarrow (\boldsymbol{D} \longrightarrow \boldsymbol{A}' \cup \boldsymbol{A}'') \\ l &\longrightarrow \textbf{if } C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l) \textbf{ then } \overline{\mathcal{A}}[\![l]\!] \textbf{ else } \underline{\mathcal{A}}[\![l]\!] \end{aligned}$$

  And the exactness criterion $C_{\underline{\mathcal{A}}'\equiv\overline{\mathcal{A}}}$ is defined by:

  $$\forall\, l \in \tilde{\boldsymbol{\mathcal{L}}},\ C_{\underline{\mathcal{A}}'\equiv\overline{\mathcal{A}}}(l) = \text{if } C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}(l) \text{ then } \mathit{true} \text{ else } C_{\underline{\mathcal{A}}\equiv\mathcal{A}}(l)$$

  It is easy to verify that $\underline{\mathcal{A}}'$ is an under-approximation of $\mathcal{A}$, and that the definition of $\underline{\mathcal{A}}'$ implies:

  $$\forall\, l,\ C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l) \implies C_{\underline{\mathcal{A}}'\equiv\overline{\mathcal{A}}}(l)$$

  Thus, $C_{\overline{\mathcal{A}}\equiv\mathcal{A}} \iff C_{\underline{\mathcal{A}}'\equiv\overline{\mathcal{A}}}$.
  The previous assumption that $\exists l : C_{\overline{\mathcal{A}}\equiv\mathcal{A}}(l) \wedge \neg C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}(l)$ implies that for at least one element l of $\mathcal{L}$ we know that $\overline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!] \equiv \underline{\mathcal{A}}'[\![l]\!]$, but we do not know whether $\underline{\mathcal{A}}[\![l]\!] \equiv \mathcal{A}[\![l]\!]$ or not. A consequence is that $\underline{\mathcal{A}}'$ is either identical to $\underline{\mathcal{A}}$ or more accurate:

  $$\underline{\mathcal{A}} \sqsubseteq \underline{\mathcal{A}}' \sqsubseteq \mathcal{A}$$

  Thus, we have shown that either $C_{\overline{\mathcal{A}}\equiv\mathcal{A}} \iff C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$ or there exists a computable under-approximation $\underline{\mathcal{A}}'$ of $\mathcal{A}$, more accurate than $\underline{\mathcal{A}}$, and such that $C_{\overline{\mathcal{A}}\equiv\mathcal{A}} \iff C_{\underline{\mathcal{A}}'\equiv\overline{\mathcal{A}}}$.

The proof is identical for $C_{\underline{\mathcal{A}}\equiv\mathcal{A}}$ and $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$. □

Figure 4 summarizes the relations between the different criteria of exactness, and the actual exactness of the approximate analyses. To sum up, if $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$ is not the best possible criterion for checking exactness, it is possible to define two other approximations, which are more accurate than the original ones. The new criterion $C_{\underline{\mathcal{A}}'\equiv\overline{\mathcal{A}}'}$ is then the best possible computable criterion (relatively to the available information about the program behavior). Moreover, from the previous proof, computing and testing the exactness of the new approximations is not more expensive than with the original solution.

Theorem 3.1 has two more consequences:

1. It may not always be useful to compute both $\overline{\mathcal{A}}$ and $\underline{\mathcal{A}}$; but we may well want to check the exactness of the chosen analysis, say $\overline{\mathcal{A}}$. However, it may sometimes be difficult to find a computable exactness criterion $C_{\overline{\mathcal{A}}\equiv\mathcal{A}}$: it may depend on $\mathcal{A}$, which is not computable! The above theorem gives a solution. If an opposite approximation $\underline{\mathcal{A}}$ is sufficiently well defined, $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$ is the best possible exactness criterion for $\overline{\mathcal{A}}$. Thus, it is sufficient to define $\underline{\mathcal{A}}$, deduce the exactness criterion $C_{\underline{\mathcal{A}}\equiv\overline{\mathcal{A}}}$ and use it as $C_{\overline{\mathcal{A}}\equiv\mathcal{A}}$. It is not even necessary to explicitly compute $\underline{\mathcal{A}}$.

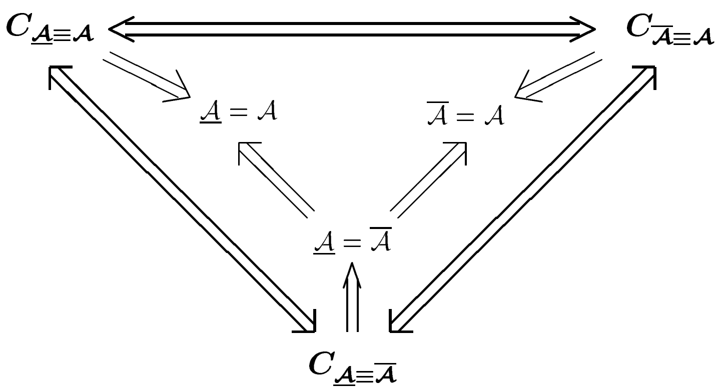

Figure 4: Relations between approximations

2. As shown in the previous section, defining an interesting approximation analysis is not always an easy task and general solutions may be disappointing. In this case, if a computable exactness criterion for the opposite analysis ($C_{\overline{\mathcal{A}}\equiv\mathcal{A}}$) is available, then $\underline{\mathcal{A}}$ can be refined by using $\overline{\mathcal{A}}$ whenever $C_{\overline{\mathcal{A}}\equiv\mathcal{A}}$ is true.

3.3 Relation with previous work [14, 15]

In previous papers, and in accordance with our implementation in PIPS, only *may* array region analyses ($\mathcal{R}$) were presented, as well as exactness criteria to flag them as exact ($C_{\overline{\mathcal{R}}\equiv\mathcal{R}}$) whenever possible. By chance, these exactness criteria are computable, though mostly[6] independent from the corresponding under-approximation $\underline{\mathcal{R}}$. $\underline{\mathcal{R}}$ can thus be defined as follows, using the notations presented throughout this paper:

$$\forall\, l \in \tilde{\boldsymbol{\mathcal{L}}},\ \underline{\mathcal{R}}[\![l]\!] = \textbf{if } C_{\overline{\mathcal{R}}\equiv\mathcal{R}}(l) \textbf{ then } \overline{\mathcal{R}}[\![l]\!] \textbf{ else } \lambda\sigma.\varnothing$$

As stated in Section 2, unless using advanced techniques too complex both in terms of memory space and computation time for array region analyses, obtaining interesting under-approximate array regions is rather hazardous. We had thus chosen not to bother about computing arbitrary solutions, but to solely focus on exact results, which is justified by the present study.

4 Conclusion

Due to a considerable amount of efforts over the last ten years, over-approximate array regions analyses are now well-known techniques, even if debates about the

[6]It was found *a posteriori* that the exactness criterion for `IF` constructs could be deduced from trivial and intuitive definitions of $\underline{\mathcal{R}}$ and $\overline{\mathcal{R}}$ derived from the exact semantics $\mathcal{R}$.

choice of the representation are still alive: Time and space complexity *versus* accuracy is the main issue, but usefulness is widely acknowledged [31, 4, 24, 21, 15].

The need for under-approximations appeared only recently [32, 26, 21], mainly for locality analysis, to allow advanced program transformations such as array privatization. But no study of the underlying semantical analysis framework had been made, thus missing its inherent problems due to the fact that under-approximate solutions of array region analyses do not belong to predefined lattices, as over-approximations do. We show that traditional ad-hoc solutions based on iteration techniques do not bring interesting results, even though problems can be lessened by more accurate representations and more complex approximations of fixed points.

We thus propose a method based on our previous experience of array region analyses [14, 15]. The idea is to perform corresponding may and must analyses at the same time, and to enhance the results of must analyses with those of may analyses according to an *exactness criterion*. In our implementation, whose results are already used in PIPS for privatizing array regions [12], must regions are not even computed; instead, may regions are flagged as exact whenever the exactness criterion is true; under-approximations are thus always equal to the empty set, unless they are exact and equal to the corresponding over-approximations. Less drastic solutions, combining our approach and iterative methods would certainly give better under-approximations; but their interest is still to be demonstrated.

The method presented in this paper, though primarily introduced for array region analyses, is not solely restricted to them. It can be useful for every type of analysis where one of the approximation is not defined on a predefined lattice. But also when exact results are required for specific applications [7, 8]. Indeed, we show how to obtain an optimal *exactness criterion* valid for both may and must analyses. Finally, and from a theoretical point of view, being able to detect the exactness of an analysis gives a partial answer to a still unresolved question from COUSOT's thesis [9], about the distance between an approximation and its corresponding exact solution.

Acknowledgments

We are very thankful to François BOURDONCLE and Pierre JOUVELOT for insightful discussions about semantical analyses, and to Denis BARTHOU for reading this paper.

References

[1] Béatrice Apvrille-Creusillet. Régions exactes et privatisation de tableaux (Exact array region analyses and array privatization). Master's thesis, Université Paris VI, France, September 1994. Available via `http://www.cri.ensmp.fr/~creusil`.

[2] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *International Conference on Programming Language Design and Implementation*, pages 41–53, June 1989.

[3] Garrett Birkhoff. *Lattice Theory*, volume XXV of *AMS Colloqium Publications*. American Mathematical Society, Providence, Rhode Island, third edition, 1967.

[4] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[5] François Bourdoncle. *Sémantique des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite*. PhD thesis, École Polytechnique, November 1992.

[6] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[7] Fabien Coelho. Compilation of I/O communications for HPF. In *Frontiers'95*, pages 102–109, February 1995. Available via `http://www.cri.ensmp.fr/~coelho`.

[8] Fabien Coelho and Corinne Ancourt. Optimal compilation of HPF remappings. Technical Report A-277-CRI, CRI, École des Mines de Paris, October 1995. To appear in JPDC in 1996.

[9] Patrick Cousot. *Méthodes Itératives de Construction et d'Approximation de Points Fixes d'Opérateurs Monotones sur un Treillis, Analyse Sémantique des Programmes*. PhD thesis, Institut National Polytechnique de Grenoble, March 1978.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[11] Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *International Conference on Computer Languages, IEEE Computer Socitey Press*, pages 95–112, May 1994.

[12] Béatrice Creusillet. Array regions for interprocedural parallelization and array privatization. Report A-279, CRI, École des Mines de Paris, November 1995. Available at `http://www.cri.ensmp.fr/~creusil`.

[13] Béatrice Creusillet. IN and OUT array region analyses. In *Fifth International Workshop on Compilers for Parallel Computers*, pages 233–246, June 1995.

[14] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 46–60. Springer-Verlag, August 1995.

[15] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. Accepted to IJPP - special issue on LCPC A/282/CRI, CRI, École des Mines de Paris, January 1996.

[16] Paul Feautrier. Array expansion. In *International Conference on Supercomputing*, pages 429–441, July 1988.

[17] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, September 1991.

[18] Kyle Gallivan, William Jalby, and Dennis Gannon. On the problem of optimizing data transfers for complex memory systems. In *International Conference on Supercomputing*, pages 238–253, July 1988.

[19] S. Graham and M. Wegman. Fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

[20] C. Gunter and D. Scott. Denotational semantics. In Jan van Leeuwen, editor, *Theoretical Computer Science*, volume B, chapter 12. Elsevier Science Publisher, 1990.

[21] Mary Hall, Saman Amarasinghe, Brian Murphy, Shih-Wei Liao, and Monica Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Supercomputing*, December 1995.

[22] Mary Hall, Brian Murphy, Saman Amarasinghe, Shih-Wei Liao, and Monica Lam. Interprocedural analysis for parallelization. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 61–80. Springer-Verlag, August 1995.

[23] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[24] Michael Hind, Michael Burke, Paul Carini, and Sam Midkiff. An empirical study of precise interprocedural array analysis. *Scientific Programming*, 3(3):255–271, May 1994.

[25] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *International Conference on Supercomputing*, pages 144–151, June 1991.

[26] Zhiyuan Li. Array privatization for parallel execution of loops. In *International Conference on Supercomputing*, pages 313–322, July 1992.

[27] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array dataflow analysis and its use in array privatization. In *Symposium on Principles of Programming Languages*, January 1993.

[28] Peter Mosses. Denotational semantics. In Jan van Leeuwen, editor, *Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publisher, 1990.

[29] Peiyi Tang. Exact side effects for interprocedural dependence analysis. In *International Conference on Supercomputing*, pages 137–146, July 1993.

[30] Rémi Triolet. *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédures.* PhD thesis, Paris VI University, 1984.

[31] Rémi Triolet, Paul Feautrier, and François Irigoin. Direct parallelization of call statements. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 176–185, 1986.

[32] Peng Tu and David Padua. Array privatization for shared and distributed memory machines (extended abstract). In *Workshop on Languages and Compilers for Distributed Memory Machines*, pages 64–67, 1992.

[33] Peng Tu and David Padua. Automatic array privatization. In *Languages and Compilers for Parallel Computing*, August 1993.