Abstract debugging of higher-order imperative languages

François Bourdoncle

DIGITAL Paris Research Laboratory
85, avenue Victor Hugo
92500 Rueil-Malmaison — France
Tel: +33 (1) 47 14 28 22

Centre de Mathématiques Appliquées
Ecole des Mines de Paris
Sophia-Antipolis
06560 Valbonne — France

bourdoncle@prl.dec.com

Abstract

Abstract interpretation is a formal method that enables the static determination (i.e. at compile-time) of the dynamic properties (i.e. at run-time) of programs. We present an abstract interpretation-based method, called *abstract debugging*, which enables the static and formal debugging of programs, prior to their execution, by finding the origin of potential bugs as well as necessary conditions for these bugs not to occur at run-time. We show how *invariant assertions* and *intermittent assertions*, such as termination, can be used to formally debug programs. Finally, we show how abstract debugging can be effectively and efficiently applied to higher-order imperative programs with exceptions and jumps to non-local labels, and present the *Syntox* system that enables the abstract debugging of the *Pascal* language by the determination of the range of the scalar variables of programs.

1 Introduction

Even though software quality is becoming more and more important, relatively few methods have been proposed to help programmers debug their programs, and debugging is typically done "post-mortem", that is, after a bug has occurred. The main drawback of this approach is that it is often very difficult, if not impossible, to find the *origin* of a bug just by looking at the memory state after the abortion of a program. Methods have been proposed to allow the reverse execution of higher-order functional languages such as *ML* [24], but these methods do not seem to be quite applicable to imperative languages such as *Pascal*, *Modula-2*, *Modula-3*, *C* or *C++*, since they have to keep track of every variable assignment. Even more important, post-mortem debugging can fail to detect bugs when programs are not extensively tested, since some parts of the code may have never been executed.

Abstract interpretation, as defined by Patrick and Radhia Cousot [8, 11, 13], is a formal method that enables the static determination of the run-time properties of programs. So far, abstract interpretation has only been applied to fairly simple languages (first-order imperative, functional, logic or parallel languages) and has been used to build sophisticated optimizing compilers. In this paper, we propose a novel, semantic-based approach to the debugging of programs, called *abstract debugging*, which combines several traditional, although rarely used, abstract interpretation techniques to allow the *static* and *formal* determination of the *origin* of certain bugs in higher-order imperative programs.

This method enables the programmer to insert assertions into the source-code of the program being debugged, and violations of these assertions are treated as run-time errors by the debugger. There are two kinds of assertions. *Invariant assertions* are properties which must *always hold* at a given control point, and are similar to the classical `assert` statement in *C* programs. *Intermittent assertions* are properties which must *eventually hold* at a given control point. Differently stated, intermittent assertions are *inevitable properties* of programs, that is, properties such that every execution of the program inevitably leads to the control point with a memory state satisfying the intermittent property.

For instance, the invariant assertion "*false*" can be used to specify that a particular control point should *not* be reached, whereas the intermittent assertion "*true*" at the end of a program specifies the termination of the program. Invariant and intermittent assertions can be freely mixed and give the programmer a great flexibility to express correctness conditions of programs.

Abstract debugging differs from traditional abstract interpretation techniques in that a very precise interprocedural

analysis of programs is needed in order to locate bugs. Therefore, many traditional approximations, which are acceptable for optimizing compilers, cannot be used here. In particular, aliasing cannot be approximated if one wants to compute a precise information about the value of the scalar variables of programs, and methods [2, 7, 14, 15, 16, 20, 22] used to determine the set of possible "alias pairs" of programs, that is, pairs of distinct variables for which there exists at least a procedure activation in which both variables have the same address, are inappropriate here, since they do not describe the simultaneous aliasing of three or more variables, and would lead to a very unprecise "abstract assignment" primitive.

This paper is organized as follows. In the first section, we give several examples of the kind of bugs that can be found by an abstract debugger. Then, in sections 3 and 4, we describe the basic techniques of abstract debugging, and show how they can be combined to effectively debug programs. In section 5, we address the problem of the abstract interpretation of higher-order, *Pascal*-like imperative languages, and briefly sketch a non-standard, copy-in/copy-out semantics of these languages that is well suited to abstract debugging. Finally, in section 6, we present the prototype *Syntox* system that enables the abstract debugging of first-order *Pascal* programs by the determination of the range of scalar variables. We discuss implementation and complexity issues, show that even very simple properties such as the range of variables enable the determination of non-trivial bugs, and show how this system can also be used to safely suppress most array bound checks during the execution of *Pascal* programs.

2 Examples

The programs of figure 1 exemplify several common programming mistakes that can typically be discovered and reported by an abstract debugger.

For instance, program "For" will obviously exit on a run-time error when accessing T[0], unless $n < 0$ at point ①. Moreover, if the index i ranges from 1 to n instead of 0 to n, then the program will exit when accessing T[101] unless $n \leq 100$ at point ①. Similarly, program "While" will loop unless $b = \mathit{false}$ at point ①, and program "Fact" will loop unless $x \geq 0$ at point ①.

It might seem quite difficult to automatically discover these kinds of bugs. However, our abstract debugger *Syntox*, described in section 6, will automatically discover and report the above necessary conditions of correctness.

A compiler could use these conditions to issue a warning or generate a call to a specific error handler to do some clean-up and exit safely, or else could enter a special debugging mode to do a step-by-step execution of the program until the error actually occurs. The interesting fact about abstract debugging is that it predicts bugs *before* they actually happen, which permits a safe handling of these bugs.

Whenever possible, an abstract debugger finds the *origin* of bugs, rather than their occurrences, and *back-propagates* necessary conditions of correctness as far as possible in order to minimize the amount of information delivered to the programmer. This feature makes abstract debugging much more useful than traditional methods and global flow analyzers such as *Lint* for instance, which is well known for the large number of warnings it generates.

For example, it is much more interesting to know that variable n of program "For" must be lower than 100 at point ① than to know that i must be less than 100 at point ② since the former test can be done once and for all after n has been read, whereas the latter must be done for every access to "T". Moreover, if $n > 100$ at point ①, then it is *certain* that the program will either loop or exit on a run-time error later on.

As we shall see in the next section, *backward propagation* is an essential component of abstract debugging, since it is responsible for the "discovery" and the "factorization" of the correctness conditions of programs. As an example, consider the following sequence of *Pascal* statements, where "T" is an array of 100 integers:

```
      read(i);
①     j := i + 1;
②     k := j;
③     read(T[k])
```

Starting from the end of the sequence, a backward analysis will determine that $k \in [1, 100]$ at point ③, $j \in [1, 100]$ at point ②, and finally that $i \in [0, 99]$ at point ①. This information can then be combined with the forward data flow, which shows that the post-condition $i \in \mathbb{Z}$ of the first call to "read" does not imply the pre-condition $i \in [0, 99]$ determined by the backward analysis. Hence, a warning can be issued to inform the programmer that if $i \notin [0, 99]$ at point ①, then his program will *certainly* fail later on.

As stated in the introduction, an important feature of abstract debugging is that programmers can freely insert *invariant assertions* and *intermittent assertions* into their programs to either statically check that important invariance properties, such as calling conditions of library functions, are satisfied, or check under which conditions a program eventually reaches a control point while satisfying a given property.

Intermittent assertions allow for a very powerful form of debugging. As an example, if the intermittent assertion $i = 10$ is inserted at point ② of program "Intermittent", then *Syntox* shows that a necessary condition for the program to *eventually* reach control point ② with $i = 10$ is that $i \leq 10$ at point ①.

It is thus possible to determine the set of program states (and, in particular, of input states) from which a program eventually reaches a given control point, by simply inserting the intermittent assertion "*true*" at this point. So for instance, if the intermittent assertion "*true*" is inserted at point ② of program "Select", then *Syntox* shows that a necessary condition for the program to terminate is that $n \leq 10$ at point ①. Differently stated, if $n > 10$, then the program will certainly loop or exit on a run-time error.

Further more, if the invariant assertion "*false*" is inserted at point ③, then *Syntox* shows that $n < -10$ at point ① is a necessary condition for the program to terminate without control ever reaching point ③.

Now, if the intermittent assertion $s = 1$ is inserted at point ②, *Syntox* shows that a necessary condition for this assertion to eventually hold is that $n \in [-10, 10]$ at point ①.

To conclude this section, we can remark that invariant assertions are normally used to express "normal" conditions of the execution of a program, whereas intermittent assertions are either used to enforce termination or to determine necessary conditions for the failure of invariant assertions. For instance, if a necessary condition of correctness is reported for a given program, then it is interesting to negate, one after the other, every invariant assertion inserted into the source code and determine necessary conditions for control to reach the corresponding point and satisfy the negated assertion, i.e. necessary conditions for the assertion to be surely violated.

Finally, it should be clear that although the standard abstract interpretation framework determines flow-insensitive program properties, invariant assertions such as "*false*" can be used to restrict the control flow and examine the behavior of a program along specific execution paths.

3 Static debugging

In this section, we shall explain the mathematical framework behind abstract debugging, leaving decidability and computability issues to the next section. We shall refer to this mathematical framework as *static debugging*, reserving the term *abstract debugging* for its tractable counterpart. Static debugging is a combination of several different techniques, namely:

- Forward propagation
- Backward propagation
- Least fixed point computation
- Greatest fixed point computation

Forward propagation is the most classical technique. It has been used for a long time in data-flow analysis to propagate program properties by following the normal flow of programs. Backward propagation does the same as forward propagation but reflects the "backwards" execution of programs. Note that, contrary to a common belief, the backward semantics of a program is not more complex nor very different from the forward semantics, the only real difference being that it is not deterministic.

Static debugging uses the abstract interpretation framework pioneered by the Cousots in which the *operational semantics* of a program is defined by a *transition relation* τ over a set of program states S. This framework aims at computing safe approximations of *fixed points* of continuous functions over complete lattices, such as the lattice $(\mathbf{P}(S), \emptyset, S, \subseteq, \cup, \cap)$ of the subsets of S. In what follows, we shall denote by $\tau^+(X)$ the set $\{y \in S \mid \exists x \in X : x \xrightarrow{\tau} y\}$ of descendants of states in $X \subseteq S$, and by $\tau^-(Y)$ the set $\{x \in S \mid \exists y \in Y : x \xrightarrow{\tau} y\}$ of ancestors of states in $Y \subseteq S$. Moreover, we shall assume that every state s has a unique descendant, i.e. $|\tau^+(\{s\})| = 1$, which is the case for classical imperative languages, and that every output state $s \in S_{out}$ and every error state $s \in S_{err}$ is *stable*, i.e. $s \xrightarrow{\tau} s$.

Note that denotational semantics cannot be used here since the backward semantics of a program in this framework is not definable from the forward semantics.

Finally, we shall implicitly identify a *property* over S with the set of states for which this property holds.

It is now well known [13] that the set of descendants of a set of states $\Sigma \subseteq S$ by a finite number of program steps is the *least* fixed point (w.r.t. the subset ordering) of function Φ defined by:

$$\Phi \;=\; \lambda X . (\Sigma \cup \tau^+(X))$$

that is, the function which takes a set of states X and returns the union of Σ and the set of descendants of states in X. This fixed point always exists and can be iteratively computed as the following limit:

$$\bigcup_{i \geq 0} \Phi^i(\emptyset)$$

Similarly, the set of ancestors of a set of states Σ is the *least* fixed point of:

$$\lambda X . (\Sigma \cup \tau^-(X))$$

Finally, the set of states which do not lead to an error is the *greatest* fixed point of:

$$\lambda X . (\tau^-(X) - S_{err})$$

More generally, let Π be a property (i.e. a set of states $\Pi \subseteq S$) that one wishes to prove about a program. Two sets are of interest for static debugging:

- The set **always**(Π) of states whose descendants satisfy Π.
- The set **eventually**(Π) of states for which there exists at least one descendant satisfying Π.

It can be shown that:

$$\begin{cases} \textbf{always}(\Pi) & = & \textbf{gfp}\left(\lambda X . (\Pi \cap \tau^-(X))\right) \\ \textbf{eventually}(\Pi) & = & \textbf{lfp}\left(\lambda X . (\Pi \cup \tau^-(X))\right) \end{cases}$$

where **lfp** and **gfp** respectively denote the least fixed point and the greatest fixed point operators. Intuitively, since the least fixed point of a continuous function over a lattice can be computed by an increasing iterative computation starting from the least element $\emptyset$, then:

$$\textbf{eventually}(\Pi) \;=\; \Pi \cup \tau^-(\Pi) \cup (\tau^-)^2(\Pi) \cup \cdots$$

is the set of ancestors of the states satisfying Π. Similarly, since a greatest fixed point can be computed by a decreasing

```
program Select;
    var n, s : integer;
    function Select(n : integer) : integer;
    begin
        if n > 10 then
            Select := Select(n + 1)
        else if n > −10 then
            Select := Select(n − 1)
        else if n = −10 then
        ③   Select := 1
        else
            Select := −1
    end;
begin
    read(n);
①   s := Select(n);
    writeln(s); ②
end.
```

```
program Fact;
    var x, y : integer;
    function F(n : integer) : integer;
    begin
        if n = 0 then
            F := 1
        else
            F := n * F(n − 1)
    end;
begin
    read(x); ①
    y := F(x) ②
end.
```

```
program McCarthy;
    var m, n : integer;
    function MC(n : integer) : integer;
    begin
        if (n > 100) then
            MC := n − 10
        else
            MC := MC(MC(MC(MC(
                    MC(MC(MC(MC(
                     MC (n + 81)))))))))
    end;
begin
    read(n); ①
    m := MC(n);
    writeln(m) ②
end.
```

```
program While;
    var i : integer;
        b : boolean;
begin
    i := 0;
    read(b); ①
    while b and (i ≤ 100) do
    ②   i := i − 1
end.
```

```
program Intermittent;
    var i : integer;
begin
⓪   read(i); ①
    while (i ≤ 100) do
    ②   i := i + 1 ③
④
end.
```

```
program For;
    var i, n : integer;
        T : array [1..100] of integer;
begin
    read(n); ①
    for i := 0 to n do
    ②   read(T[i])
end.
```

Figure 1: Examples

iterative computation starting from the maximum element S, then:

$$\textbf{always}(\Pi) \;=\; \Pi \cap \tau^{-}(\Pi) \cap (\tau^{-})^{2}(\Pi) \cap \cdots$$

is the set of states satisfying Π whose ancestors satisfy Π. So for instance, **eventually**(S_{out}) is the set of states for which the program terminates, **eventually**(S_{err}) is the set of states leading to a run-time error, **always**$(S - S_{out})$ is the set of states which either cause the program to loop or to exit on a run-time error, and **always**$(S - S_{err})$ is the set of states which do not lead to a run-time error.

Note that under the assumption that $|\tau^{+}(\{s\})| = 1$ for every $s \in S$, it can be shown that:

$$\textbf{always}(\overline{\Pi}) \;=\; \overline{\textbf{eventually}(\Pi)}$$

where $\overline{\Pi}$ denotes the negation (or complement) of Π. However, since approximate lattices used to finitely compute safe approximations of program properties are usually not complemented, this equality is of no practical use. For instance, the complement of an interval, say $[1, 5]$, is not an interval in general.

The two sets **always**(Π) and **eventually**(Π) can be used to perform the static debugging of programs as follows. Suppose for instance that a programmer wants to prove that a set of properties $\{\pi_k\}_{k\in K_a}$ *always* hold at control points $\{c_k\}_{k\in K_a}$. We shall assume that every state other than an error state consists in a pair $\langle c, m\rangle$ of a control point c and a memory state m. Then, the *global invariant assertion* to prove is:

$$\Pi_a \;=\; \{\langle c, m\rangle \in S \mid \forall k \in K_a : c = c_k \Longrightarrow m \in \pi_k\}$$

that is, Π_a enforces π_k at point $c_k \in K_a$ and is "*true*" everywhere else. Similarly, if the programmer wants to prove that at least one control point of a set $\{c_k\}_{k\in K_e}$, $K_a \cap K_e = \emptyset$, will be *eventually* reached at run-time with a memory state satisfying $\{\pi_k\}_{k\in K_e}$, then the *global intermittent assertion* to prove is:

$$\Pi_e \;=\; \{\langle c, m\rangle \in S \mid \exists k \in K_e : c = c_k \;\wedge\; m \in \pi_k\}$$

that is, Π_e enforces π_k at point $c_k \in K_e$ and is "*false*" everywhere else. The *program invariant*, which represents the set of program states that can be reached during program executions that are correct with respect to the programmer's specifications, is then the limit $\mathbf{I}$ of the decreasing chain I_k defined by $\mathrm{I}_0 = S$ and iteratively computed by applying the following steps in sequence:

1) Compute the set I_{k+1} of descendants of the states in I_k by computing the *least* fixed point of:

$$\lambda X . (I_k \cap (S_{in} \cup \tau^+(X)))$$

where S_{in} is the set of input states.

2) Compute the set I_{k+2} of states in I_{k+1} whose descendants satisfy Π_a by computing the *greatest* fixed point of:

$$\lambda X . (I_{k+1} \cap \Pi_a \cap \tau^-(X))$$

3) Compute the set I_{k+3} of states in I_{k+2} for which there exists at least one descendant satisfying Π_e by computing the *least* fixed point of:

$$\lambda X . (I_{k+2} \cap (\Pi_e \cup \tau^-(X)))$$

Step 1 is a forward analysis, whereas steps 2 and 3 are backward analyses. Note that the chain $(I_k)_{k\geq 0}$ can be infinitely strictly decreasing, since each step can refine the previous one. For instance, steps 2 and 3 can back-propagate a condition on the input states (i.e. "remove" a few input states) that will be then propagated forward by step 1, etc.

When the invariant **I** has been reached, it is very easy to determine the source of potential bugs in the program and issue warnings to the programmer.

First, every input state s that is not in **I** is an "erroneous" state and any execution starting from this state will lead to a violation of at least one of the programmer's invariants. Moreover, it is easy to see that every state $s' \notin \mathbf{I}$ that is a descendant of a state $s \in \mathbf{I}$ is erroneous, since it follows from the forward data flow but is not part of the backward flow (section 2).

The characterization of this set of erroneous "frontier states" is the negation of the correctness condition reported to the programmer.

Finally, note that in real programming languages, the hypothesis $|\tau^+(\{s\})| = 1$ for every state $s \in S$ does not hold, and some states can have several descendants. This is the case, for instance, for input statements and procedure calls (since local variables have undefined values upon procedure entry), but the same property holds for logic programs, which are intrinsically non-deterministic. It can be shown that the framework remains valid but that the invariant **I** becomes a *necessary* condition of correctness and is no longer a *sufficient* condition.

4 Abstract debugging

Since each invariant I_k is not computable in general, abstract interpretation techniques must be used to finitely compute a safe approximation of **I**. The standard framework, defined in Cousot [8], consists in defining a Galois connection (α, γ) between the exact lattice $(\mathbf{P}(S), \emptyset, S, \subseteq, \cup, \cap)$ and a finitely represented approximate lattice $(\mathbf{P}^\#(S), \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ used to represent safely approximated program properties.

The *abstraction function* $\alpha : \mathbf{P}(S) \rightarrow \mathbf{P}^\#(S)$ maps sets of states to their best approximation in the abstract lattice, whereas the *concretization* or *meaning* function $\gamma : \mathbf{P}^\#(S) \rightarrow \mathbf{P}(S)$ maps every abstract property to its "meaning", i.e. the set of states satisfying the property. Note that α loses information, whereas γ does not, that is:

$$\gamma \circ \alpha \supseteq 1_{\mathbf{P}(S)} \qquad \text{and} \qquad \alpha \circ \gamma = 1_{\mathbf{P}^\#(S)}$$

The advantage of using Galois connections is that if a function $\Phi^\#$ is a *safe approximation*:

$$\Phi^\# \sqsupseteq \alpha \circ \Phi \circ \gamma$$

of a semantic function Φ, e.g. $\Phi = \lambda X . (S_{in} \cup \tau^+(X))$, then the least fixed point (for instance) of $\Phi^\#$ is automatically a safe approximation of the least fixed point of Φ, that is:

$$\gamma(\mathbf{lfp}(\Phi^\#)) \sqsupseteq \mathbf{lfp}(\Phi)$$

Note that α and γ are never actually implemented and only serve the purpose of establishing the semantic correctness of $\Phi^\#$. Also, note that the semantic functions Φ and $\Phi^\#$ depend on the program. However, it is easy to see that $\Phi^\#$ is built-up on a standard way from "abstract primitives" which are program independent and are the only semantic functions actually implemented in an abstract debugger.

For instance, program "Intermittent" of figure 1 is associated with the forward system of semantic equations, corresponding to the least fixed point equation $X = \Phi^\#(X)$:

$$\begin{array}{lcl} x_0 & = & \top \\ x_1 & = & [\![\mathbf{read}(i)]\!](x_0) \\ x_2 & = & [\![i \leq 100]\!](x_1) \sqcup [\![i \leq 100]\!](x_3) \\ x_3 & = & [\![i := i+1]\!](x_2) \\ x_4 & = & [\![i > 100]\!](x_1) \sqcup [\![i > 100]\!](x_3) \end{array}$$

where, for example, $[\![i \leq 100]\!]$ denotes the "abstract test" primitive, which must satisfy, for every $x \in \mathbf{P}^\#(S)$:

$$[\![i \leq 100]\!](x) \sqsupseteq \alpha\big(\{i \in \gamma(x) : i \leq 100\}\big)$$

Similarly, when the *intermittent* assertion $i = 10$ is inserted at point ②, the backward system of semantic equations to be solved iteratively, starting from $\bot$, is:

$$\begin{array}{lcl} x_0 & = & [\![\mathbf{read}(i)]\!]^{-1}(x_1) \\ x_1 & = & [\![i \leq 100]\!](x_2) \sqcup [\![i > 100]\!](x_4) \\ x_2 & = & \alpha(\{10\}) \sqcup [\![i := i+1]\!]^{-1}(x_3) \\ x_3 & = & [\![i \leq 100]\!](x_2) \sqcup [\![i > 100]\!](x_4) \\ x_4 & = & x_4 \end{array}$$

where $[\![i := i+1]\!]^{-1}$ denotes the "backward abstract assignment" primitive which, in this particular case, is equivalent to the $[\![i := i-1]\!]$ forward primitive. Finally, if the *invariant* assertion "$i \geq 0$" is inserted at point ②, the backward system of semantic equations to be solved, starting from $\top$, becomes:

$$\begin{array}{lcl} x_0 & = & [\![\mathbf{read}(i)]\!]^{-1}(x_1) \\ x_1 & = & [\![i \leq 100]\!](x_2) \sqcup [\![i > 100]\!](x_4) \\ x_2 & = & \alpha(\{0,1,2,3,\ldots\}) \sqcap [\![i := i+1]\!]^{-1}(x_3) \\ x_3 & = & [\![i \leq 100]\!](x_2) \sqcup [\![i > 100]\!](x_4) \\ x_4 & = & x_4 \end{array}$$

Note that the last two systems are obtained by a trivial inversion of the forward system and by adding the appropriate unions (resp. intersections) to the right-hand sides of the equations, e.g. "$\cdots \sqcup \alpha(\{10\})$", to take into account the programmer's intermittent (resp. invariant) assertions.

When the approximate lattice is of finite height, iterative computations of solutions of these systems always terminate, but when the lattice is of infinite height, or when its height is finite but very large, *speed-up techniques* must be used.

These techniques, known as *widening* and *narrowing* [4, 5, 8, 11, 13], allow the determination of *safe approximations of approximate fixed points*, while enforcing *finite* iterative computations. They allow trade-offs to be made between computation time and precision. We recall that a widening operator ∇ is a safe approximation of the union, that is:

$$\forall x, y \in \mathbf{P}^{\#}(S) \ : \ x \nabla y \ \sqsupseteq \ x \sqcup y$$

and is such that for every increasing chain $(x_i)_{i \geq 0}$, the chain $(x'_i)_{i \geq 0}$ defined by:

$$x'_0 \ = \ x_0 \qquad \text{and} \qquad x'_{i+1} \ = \ x'_i \nabla x_{i+1} \quad (i \geq 0)$$

is always eventually stable. A narrowing operator Δ satisfies:

$$\forall x, y \in \mathbf{P}^{\#}(S) \ : \ x \ \sqsupseteq \ y \ \Longrightarrow \ x \ \sqsupseteq \ x \Delta y \ \sqsupseteq \ y$$

and is such that for every decreasing chain $(x_i)_{i \geq 0}$, the chain $(x'_i)_{i \geq 0}$ defined by:

$$x'_0 \ = \ x_0 \qquad \text{and} \qquad x'_{i+1} \ = \ x'_i \Delta x_{i+1} \quad (i \geq 0)$$

is always eventually stable. For every continuous semantic function $\Phi^{\#}$, it can be shown that the increasing chain:

$$x_0 \ = \ \bot \qquad \text{and} \qquad x_{i+1} \ = \ x_i \nabla \Phi^{\#}(x_i) \quad (i \geq 0)$$

is eventually stable and that its limit $\phi^{\#}$ is a *post-fixed point* of $\Phi^{\#}$ (i.e. $\Phi^{\#}(\phi^{\#}) \sqsubseteq \phi^{\#}$) and, therefore, a safe approximation of the least fixed point of $\Phi^{\#}$. Similarly, it can be shown that the decreasing chain:

$$x_0 \ = \ \phi^{\#} \qquad \text{and} \qquad x_{i+1} \ = \ x_i \Delta \Phi^{\#}(x_i) \quad (i \geq 0)$$

starting from any post-fixed point $\phi^{\#}$ of $\Phi^{\#}$ is always a safe approximation of the *least* fixed point of $\Phi^{\#}$ and, choosing $\phi^{\#} = \top$, that this limit is a safe approximation of the *greatest* fixed point of $\Phi^{\#}$.

These properties show that safe approximations of least fixed points of continuous functions can be finitely computed by a combination of a *widening phase* starting from $\bot$, followed by a *narrowing phase* starting from the result of the previous phase, and that safe approximations of greatest fixed points can be computed by a single *narrowing phase* starting from $\top$.

Note that, when working with systems of equations, it is not necessary to apply widening operators to each equation, but only to a set W of equations such that every cycle in the dependency graph of the system is cut by at least an element $w \in W$, called a *widening point*. For instance, for the first system above, only the third equation needs to be replaced by:

$$x_2 \ = \ x_2 \ \nabla \ \left([\![i \leq 100]\!](x_1) \ \sqcup \ [\![i \leq 100]\!](x_3) \right)$$

Of course, since widening operators lead to a loss of precision, the smaller the cardinal of W, the better, but since the problem of finding minimum sets W is NP-complete, safe heuristics have to be developed to select "good" sets of widening points and iteration strategies (section 6.3).

Finally, note that the approximation of **I** implies that the correctness conditions determined by an abstract debugger are *necessary*, but not always sufficient.

5 Higher-order languages

As stated in the introduction, one of the constraints of abstract debugging is to efficiently determine precise informations about the dynamic behavior of programs in order to be able to detect bugs. It is thus desirable that the abstraction used to approximate program invariants be highly "tunable" and arbitrarily precise. Earlier works [13] show that this is the case for "simple", first-order languages without reference parameters. However, reference parameters and procedure parameters are the source of two distinct problems: aliasing and environment sharing, which both lead to very unprecise and expensive abstract interpretations.

Intuitively, environment sharing, i.e. the fact that the same variable can be accessible, at the same time, to different procedure activations in the run-time stack of a block-structured program with local procedures, implies that the abstract assignment of non-local variables, to be safe, must be "additive" rather than "destructive" as is normally the case ([5], p. 105).

Similarly, if the partition of the set of identifiers accessible to a given procedure activation into subsets of identifiers having the same address is not known exactly, then the abstract assignment primitive has an exponential complexity, since the assignment must be simulated for every possible partition, and is also very unprecise ([5], p. 106).

We have thus designed a non-standard, copy-in/copy-out semantics of higher-order imperative languages, and shown that this semantics is equivalent to the standard, stack-based semantics (as can be found for instance in Aho et al. [1]) of second-order *Pascal* programs with jumps to local and non-local labels.

An early version of this semantics for first-order *Pascal* programs can be found in Bourdoncle [3] and the version for higher-order imperative languages can be found in Bourdoncle [5], p. 113–196.

We have shown that this semantics is also equivalent to the standard semantics for an undecidable sub-class of higher-order programs containing, in particular, programs with exceptions but without local procedures (which allows for the treatment of the `setjmp` and `longjmp` primitives of *C*).

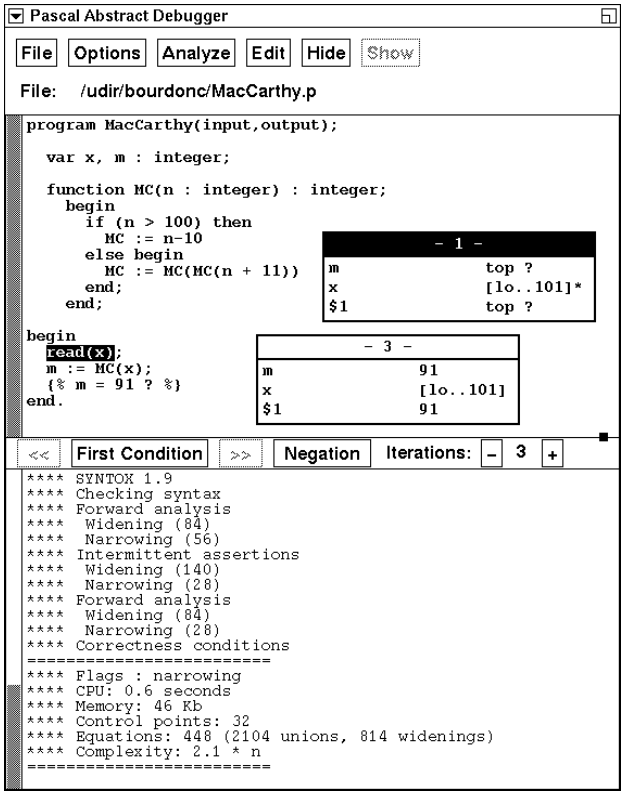

Figure 2: The *Syntox* system

Moreover, we have shown that for programs for which the two semantics coincide, it is possible to determine variable aliasing *exactly*, that is, it is possible to determine, for every procedure, all the possible run-time partitions of the local and global variables of this procedure into subsets of variables having the same address. The knowledge of these partitions is much richer than what can be found for instance in Banning [2] since there may be a procedure activation where x and y are aliases and another activation where y and z are aliases, but no activation where x, y and z are aliases simultaneously. As a matter of fact, the classical problem of finding all possible *alias pairs* of a program is an abstraction (namely, graph union and transitive closure) of the "real" aliasing information.

6 The Syntox system

The *Syntox* system is an interprocedural abstract debugger that implements the ideas of section 3 and 4, and the non-standard semantics of section 5 for a subset of *Pascal*. This system, which is only a research prototype, can be used to find bugs which are related to the range of scalar variables, such as array indexing, range sub-types, etc. The lattice of program properties used is thus non-relational, but we have shown ([5], p. 197–216) that any relational lattice can be chosen, and that results can be arbitrarily precise, even in the presence of aliasing, local procedures passed as parameters, jumps to non-local labels and exceptions (which do not exist in *Pascal*).

Relational lattices can be used to determine properties, such as linear inequalities of the form $i \leq 2 * j + 1$, that exist between different variables of a program [10]. Even though the interval lattice is quite simple, we shall see in section 6.5 that it allows to determine non-trivial bugs.

6.1 Interval lattice

The interval lattice $\mathbf{I}(\mathbb{Z}_b)$ used by *Syntox* is a safe approximation of $\mathbf{P}(\mathbb{Z}_b)$ where $\mathbb{Z}_b$ denotes the set of integers between $\omega^- = -2^{b-1}$ and $\omega^+ = 2^{b-1}-1$. This lattice being of height 2^b, fixed point computations can require up to 2^b iterations. Therefore, we use the standard widening and narrowing operators of Cousot [8, 13] defined, for every $x \in \mathbf{I}(\mathbb{Z}_b)$, by:

$$\bot \nabla x = x \nabla \bot = x, \qquad \bot \Delta x = x \Delta \bot = \bot$$

and, for every $[a_1, b_1], [a_2, b_2] \in \mathbf{I}(\mathbb{Z}_b)$, by:

$$\begin{aligned} [a_1, b_1] \ \nabla \ [a_2, b_2] \ = \\ & [\textbf{if } a_2 < a_1 \textbf{ then } \omega^- \textbf{ else } a_1, \\ & \ \textbf{if } b_2 > b_1 \textbf{ then } \omega^+ \textbf{ else } b_1] \\ [a_1, b_1] \ \Delta \ [a_2, b_2] \ = \\ & [\textbf{if } a_1 = \omega^- \textbf{ then } a_2 \textbf{ else } \mathit{min}(a_1, a_2), \\ & \ \textbf{if } b_1 = \omega^+ \textbf{ then } b_2 \textbf{ else } \mathit{max}(b_1, b_2)] \end{aligned}$$

It is easy to see that these operators enforce the convergence of iterative computations of least fixed points in four steps at most. For instance, the values taken by x_2 during the iterative computation of the solution of the first system of section 4 are:

$$\bot, \quad [0,0], \quad [0,0] \nabla ([0,0] \sqcup [1,1]) \ = \ [0, \omega^+]$$

for the widening phase and:

$$[0, \omega^+], \quad [0, \omega^+] \Delta ([0,0] \sqcup [0,100]) \ = \ [0,100]$$

for the narrowing phase, which gives the optimum results: $x_2 = [0,100]$ and $x_4 = [10,10]$. Therefore, the approximate computation of a least fixed point over $\mathbf{I}(\mathbb{Z}_b)$ using a widening phase and a narrowing phase is only four times more complex than constant propagation!

Finally, note that more sophisticated widening and narrowing operators can be easily designed [4] to integrate ad-hoc heuristics which are appropriate for the class of programs considered, as long as they satisfy the generic requirements stated above.

6.2 Language restrictions

For historical reasons, *Syntox* does not yet allow procedures to be passed as parameters to other procedures, but the theoretical results of section 5 show that this feature could be added without major problems. Variant records and the "with" construct are not allowed in programs. Only the most standard *Pascal* library functions are predefined. Programs with pointers to heap-allocated objects are accepted, but are not always

handled safely with respect to aliasing; other works on the abstract interpretation of heap-allocated data structures such as Deutsch [15, 16] could be used to handle pointer-induced aliasing. Records are accepted, but no information is given on their fields. This decision was made to simplify the design of the debugger, but records can be handled without much trouble. Jumps to local and non-local labels are fully supported.

6.3 Algorithms and complexity

Two different fixed point computation algorithms, described in Bourdoncle [6], are used by *Syntox*. Both algorithms are based on a "weak topological ordering" decomposition of the dependency graph of the system of equations which generalizes topological ordering to directed graphs containing cycles. In particular, we have shown that the hierarchical decomposition of a reducible graph [1] obtained by computing its limit graph is a weak topological ordering, and that any weak topological ordering of a dependency graph gives an admissible set of widening points (section 4) as well as two "good" iteration strategies, that is, algorithms for iteratively and asynchronously solving the system of equations [9].

The first strategy is used to compute intraprocedural fixed points and takes advantage of the fact that the intraprocedural dependency graph is known in advance to achieve an excellent complexity. Theoretical results [5, 6] show that the complexity of this strategy is the product of the height h of the abstract lattice by the sum of the individual *depths* of the n nodes in the decomposition of the graph. The maximum complexity is thus n when the graph is acyclic, and is at most:

$$\frac{h \cdot n \cdot (n+1)}{2}$$

Note that the use of widening and narrowing operators over the lattice of intervals leads to the same complexity with $h = 4v$, where v is the number of variables, and that $h = 4$ is a good approximation of the empirical "average" complexity.

The second strategy is used during the interprocedural analysis, for which the dependency graph is not known in advance, and is based on a depth-first visit of the interprocedural call graph. We have shown that the overall complexity of a fixed point computation over a program with n control points, c procedure calls, p procedures and l intraprocedural loops is at most:

$$h \cdot n \cdot (c+p+l) \quad = \quad \rho \cdot h \cdot n^2$$

where $\rho \leq 1$ is the sum $l/n + c/n$ of the densities of intraprocedural loops and procedure calls in the program, and of the inverse of the average size n/p of procedures. However, practice shows that complexity is rarely quadratic, except for programs which consist in tightly coupled mutually recursive procedures or ad-hoc programs such as program "McCarthy" of figure 1, which is equivalent, in terms of complexity, to a program with 10 mutually recursive procedures or to a linear program 100 times longuer.

```
program BinarySearch;
    type index = 1..100;
    var  n : index;   key : integer;
         T : array [index] of integer;
    function Find(key : integer) : boolean;
         var m, left, right : integer;
    begin
         left := 1;  right := n;
         repeat
              m := (left + right) div 2;
              if  (key < T[m])  then
                   right := m - 1
              else
                   left := m + 1
         until (key = T[m]) or  (left > right);
         Find := (key = T[m])
    end;
begin
    read(n, key); writeln("Found = ", Find(key))
end.
```

Figure 3: Binary search

6.4 Implementation

Syntox consists of approximately 20.000 lines of *C*, 4.000 of which implement a user-friendly interface under the *X Window* system. The system has its own integrated editor shown in figure 2. Once a program has been successfully parsed and analyzed, the user can click on any statement and the debugger pops up a window displaying the abstract memory state right after the execution of this statement. If needed, the window can be dragged to a permanent position on the screen. When a procedure has reference parameters, *Syntox* gives a description of all the possible "alias sets" of this procedure [3]. Intermittent and invariant assertions can be inserted before any statement.

The analysis of a program is done in several steps. The first step consists in writing the *intraprocedural semantic equations* associated with each procedure of the program. The forward system of equations directly follows from the syntax of the program, and the backward equations are built by a trivial inversion of the forward system as described in section 4.

The debugger then repeatedly performs a forward analysis and two backward analyses (one for each kind of assertion) and stops after a user-selectable number of passes. The default is to perform a forward analysis, two backward analyses and a final forward analysis. Each analysis consists of a fixed point computation (either a least fixed point or a greatest fixed point) with a widening phase and a narrowing phase, which implies that a complete analysis is, in practice, $4 \times 4 = 16$ times more complex than constant propagation.

Note that when the program has recursive procedures, the interprocedural call graph is dynamically unfolded during the analysis, and each procedure activation is duplicated accord-

ing to the value of its *token* [3, 4, 5, 19, 23] which consists in the static calling site of the activation and the set of all its aliases. When this duplication is too costly in terms of time and memory, it is possible to avoid it, at the cost of a loss of precision.

6.5 Results

Although the lattice of intervals used by *Syntox* is rather simple, the use of tokens to unfold the interprocedural call graph and the use of widening and narrowing techniques allow for the discovery of very subtle and non-trivial bugs. As an example, consider program "McCarthy" of figure 1. This program implements, for $k = 9$, a generalization MC_k of McCarthy's 91 function defined by:

$$MC_k(n) = \begin{cases} n - 10 & \textbf{if } n > 100 \\ (MC_k)^k(n + 10k - 9) & \textbf{if } n \leq 100 \end{cases}$$

It can be shown that this function has the following meaning:

$$MC_k(n) = \begin{cases} n - 10 & \textbf{if } n > 100 \\ 91 & \textbf{if } n \leq 100 \end{cases}$$

If the invariant assertion $n \leq 101$ is inserted at point ①, *Syntox* proves that $m = 91$ at point ②, which shows that if a call $MC_9(n)$ terminates and $n \leq 101$, then $MC_9(n) = 91$. Even more interesting, if the intermittent assertion $m = 91$ is inserted at point ②, then *Syntox* shows that a necessary condition for this property to hold is that $n \leq 101$ at point ①. Finally, if the correct value 81 is replaced by any value less than 80, for instance 71, and that the intermittent assertion "*true*" is inserted at point ②, then *Syntox* shows that a necessary condition for the program to terminate is that $n \geq 101$ at point ①. Experimentally, it can be shown that this erroneous program loops for every value $n \leq 100$ by calling MC_9 with a finite set of arguments n. For instance, the call $MC_9(0)$ leads to a cycle of length 81. Interestingly, this bug was effectively discovered while attempting to generalize McCarthy's 91 function with an incorrect formula.

Another use of *Syntox* is to prove that every array access in a program is statically correct, so that a compiler need not generate code to check that array indices are correct at runtime. We have been able to show automatically that every array access is statically correct in particular implementations of HeapSort and BinarySearch (figure 3), and that most accesses (i.e. all but one or two) are also correct in other implementations of various sorting algorithms. The experimental comparison between these programs compiled with and without array bound checking shows a speed-up ranging from 30% to 40%.

These results are clearly superior to previous ones. For instance, Harrison [17] computes the *greatest* fixed point of the forward system of semantic equations, which has no semantic justification and gives poor results (section 3). Moreover, since he does not use a narrowing operator, his analysis can be extremely costly. More recently, Markstein et al. [21] use strength reduction, code motion and subexpression elimination to move range checks outside loops, and Gupta [18] uses "monotonicity" to improve their results. These works are partial attempts to perform backward analyses, but they are not semantically founded, and rely on ad-hoc heuristics related to the way induction variables are computed. On the contrary, our method works without such assumptions, can be applied to arbitrary recursive procedures, back-propagates assertions much further, and gives better results. For instance, every array access in programs "Matrix" and "Shuttle" of Markstein et al. [21] is statically proven correct by *Syntox*.

Program	Size	Memory	Time
Fact	24	44 kb	0.5 sec
Select	61	64 kb	0.9 sec
Ackermann	72	99 kb	1.9 sec
QuickSort	92	98 kb	2.1 sec
HeapSort	96	108 kb	2.4 sec
McCarthy_9	176	230 kb	5.4 sec
McCarthy_{30}	1184	3387 kb	153.3 sec

Figure 4: Statistics

Finally, note that *Syntox* can be used to statically check that variables having *range sub-types*, such as 1..100, are used consistently. As a matter of fact, range sub-types act as "permanent invariant assertions" and have proven to be very useful for abstract debugging.

The time and memory requirements for the abstract debugging of programs are reasonable. The table in figure 4 shows the size of differents examples (i.e. the total number of control points after having unfolded the interprocedural call graph), the allocated memory in kbytes, and the analysis time in seconds for a DEC 5000/200 Ultrix workstation.

These results show that, in practice, the amount of time and memory required is closer to the linear case than the quadratic case, except for very complex programs, and therefore invalidate a common belief according to which static analysis of programs would be exponential.

7 Conclusions and future work

We have presented a new static, semantic-based approach to the debugging of programs, called "abstract debugging", that allows programmers to use invariant and intermittent assertions to statically and formally check the validity of a program, test its behavior along certain execution paths, and find the origin of bugs rather than their occurrences.

This method is based on the operational abstract interpretation framework and cannot be translated to the denotational framework used by systems such as *SPARE* [25].

We have shown that abstract debugging can be efficiently implemented with a worst-case quadratic complexity, and shown that non-trivial bugs can be automatically discovered even when the lattice of abstract properties is fairly simple.

Finally, we have presented the prototype abstract debugger *Syntox* which can be used to debug first-order *Pascal* programs. The methods we have developed are not restricted to *Pascal*, and could be easily applied to other "safe" imperative languages such as *Modula-3* or safe subsets of *C++*, and to functional or logic programming languages.

Although we have not tried to debug large, real-life programs with *Syntox*, all experiments done to date indicate that the time and space complexity of abstract debugging lies somewhere between linear and quadratic, and that only intrinsically complex programs tend to be complex to analyze. We are therefore confident that this technique can be effectively applied to reasonably sized, real-life programs.

8 Acknowledgements

I wish to thank Patrick and Radhia Cousot for their support and helpful comments on this work.

References

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman: "Compilers — Principles, Techniques and Tools", Addison-Wesley Publishing Company (1986)

[2] John P. Banning: "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables", *Proc. of the 6th ACM Symp. on POPL* (1979) 29–41

[3] François Bourdoncle: "Interprocedural Abstract Interpretation of Block Structured Languages with Nested Procedures, Aliasing and Recursivity", *Proc. of the International Workshop PLILP'90*, Lecture Notes in Computer Science 456, Springer-Verlag (1990) 307–323

[4] François Bourdoncle: "Abstract Interpretation By Dynamic Partitioning", *Journal of Functional Programming*, Vol. 2, No. 4 (1992) 407–435

[5] François Bourdoncle: "Sémantiques des langages impératifs d'ordre supérieur et interprétation abstraite", *Ph.D. dissertation*, Ecole Polytechnique (1992)

[6] François Bourdoncle: "Efficient Chaotic Iteration Strategies with Widenings", *Proc. of the International Conf. on Formal Methods in Programming and their Applications*, Lecture Notes in Computer Science, Springer-Verlag (1993) *to appear*

[7] Keith D. Cooper: "Analyzing Aliases of Reference Formal Parameters", *Proc. of the 12th ACM Symp. on POPL* (1985) 281–290

[8] Patrick and Radhia Cousot: "Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", *Proc. of the 4th ACM Symp. on POPL* (1977) 238–252

[9] Patrick Cousot: "Asynchronous iterative methods for solving a fixpoint system of monotone equations", Research Report IMAG-RR-88, Université Scientifique et Médicale de Grenoble (1977)

[10] Patrick Cousot and Nicolas Halbwachs: "Automatic discovery of linear constraints among variables of a program", *Proc. of the 5th ACM Symp. on POPL* (1978) 84–97

[11] Patrick Cousot: "Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis. Analyse sémantique de programmes", *Ph.D. dissertation*, Université Scientifique et Médicale de Grenoble (1978)

[12] Patrick and Radhia Cousot: "Static determination of dynamic properties of recursive procedures", *Formal Description of Programming Concepts*, North Holland Publishing Company (1978) 237–277

[13] Patrick Cousot: "Semantic foundations of program analysis" in Muchnick and Jones Eds., *Program Flow Analysis, Theory and Applications*, Prentice-Hall (1981) 303–343

[14] Alan J. Demers, Anne Neirynck and Prakash Panangaden: "Computation of Aliases and Support Sets", *Proc. of the 14th ACM Symp. on POPL* (1987) 274–283

[15] Alain Deutsch: "On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications", *Proc. of the 17th ACM Symp. on POPL* (1990)

[16] Alain Deutsch: "A Storeless Model of Aliasing and its Abstractions using Finite Representations of Right-Regular Equivalence Relations", *Proc. of the IEEE'92 International Conf. on Computer Languages*, IEEE Press (1992)

[17] William H. Harrison: "Compiler Analysis of the Value Ranges for Variables", *IEEE Transactions on software engineering*, Vol. SE-3, No. 3, (1977) 243–250

[18] Rajiv Gupta: "A Fresh Look at Optimizing Array Bound Checking", *Proc. of SIGPLAN '90 Conf. on Programming Language Design and Implementation* (1990) 272–282

[19] Neil D. Jones and Steven Muchnick: "A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures", in *Proc. of the 9th ACM Symp. on POPL* (1982)

[20] William Landi and Barbara G. Ryder: "Pointer-induced Aliasing: A Problem Classification", *Proc. of the 18th ACM Symp. on POPL* (1991) 93–103

[21] Victoria Markstein, John Cocke and Peter Markstein: "Optimization of Range Checking", *Proc. of the SIGPLAN'82 Symp. on Compiler Construction* (1982) 114–119

[22] Jan Stransky: "A lattice for Abstract Interpretation of Dynamic (Lisp-like) Structures", Information and Computation 101 (1992) 70–102

[23] Micha Sharir and Amir Pnueli: "Two Approaches to Interprocedural Data Flow Analysis" in Muchnick and Jones Eds., *Program Flow Analysis, Theory and Applications*, Prentice-Hall (1981) 189–233

[24] Andrew P. Tolmach and Andrew W. Appel: "Debugging Standard ML Without Reverse Engineering", *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, ACM Press (1990) 1–12

[25] G.A. Venkatesh and Charles N. Fisher: "SPARE: A Development Environment For Program Analysis Algorithms", *IEEE Transactions on software engineering*, Vol. 18, No. 4, (1992) 304–318