Static Debugging of C Programs: Detection of Pointer Errors in Recursive Data Structures

Ronan Gaugne

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Static debugging of C programs: detection of pointer errors in recursive data structures

Ronan Gaugne

N ° 3232

août 1997

THÈME 2

ISSN 0249-6399

Static debugging of C programs: detection of pointer errors in recursive data structures

Ronan Gaugne *

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n ° 3232 — août 1997 — 26 pages

Abstract: The incorrect use of pointers is one of the most common source of bugs in imperative languages. In this context, any kind of static code checking capable of detecting potential bugs at compile time is welcome. This paper presents a static debugging technique for the detection of incorrect accesses to memory (dereferences of invalid pointers). The analysed language is a subset of C. The tool is based on a static analyser extended with assertions inserted in the body of the program. Assertions are of two kinds:

- static assertions automatically verified by the analyser,
- dynamic assertions treated as assumptions by the analyser.

The technique deals with dynamically allocated data structures and it is accurate enough to handle circular structures.

Key-words: static debugging, static analysis, program verification, dangling pointers, assertion based debugging, Hoare logic

(Résumé : tsvp)

* gaugne@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 02 99 84 71 00 – Télécopie : (33) 02 99 84 71 71

Débogage statique de programmes C: détection des erreurs de pointeurs dans les structures de données récursives

Résumé : L'utilisation incorrecte de pointeurs est une des sources d'erreurs les plus répandues dans les langages impératifs. Dans ce contexte, tout vérificateur statique de code capable de détecter des erreurs potentielles à la compilation est bienvenu. Cet article présente une technique de débogage statique pour la détection d'accès incorrects à la mémoire (déréférences de pointeurs invalides). Le langage considéré est un sous-ensemble de C. L'outil est basé sur un analyseur statique étendu par des assertions qui sont de deux types:

- les assertions statiques qui sont vérifiées automatiquement par l'analyseur.
- les assertions hypothétiques qui sont traitées comme des hypothèses par l'analyseur.

La technique prend en compte les structures de données allouées dynamiquement et elle est suffisamment précise pour traiter les structures circulaires.

Mots-clé : débogage statique, analyse statique, vérification de programmes, pointeurs pendants, débogage par assertions, logique de Hoare

1 Introduction

The motivation for the work described in this paper comes from three observations:

- Most widely used programming languages allow explicit pointer manipulations. The expressiveness provided by such features is appreciated by many programmers because it makes possible to master low level details about memory allocation and reuse. However, the explicit use of pointers can be quite subtle and error prone. It is well known that one of the most common source of bugs in C is the incorrect use of pointers.
- Dynamic debugging tools are very accurate to locate errors in programs because they have access to the exact state of the memory. Their weakness is that they are execution dependent and it is not possible in general to test all existing executions of a program. On the opposite, static code checkers perform an exhaustive verification of programs at the cost of some approximations.
- In debugging, a widespread technique is the use of assertions inserted in the body of a program to specify its expected behavior. The importance of assertions in debugging is discussed in [17]. Verification of assertions is generally done by a theorem prover and can be a very time consuming task. An important challenge is to find a compromise between the expressiveness of the analysed properties and the mechanisation of the analysis.

The goal of the technique described in this paper is the detection of incorrect memory accesses through dereferencing invalid pointers. A pointer may be invalid because it has not been initialised or because it refers to a memory location which has been deallocated. The main features of the debugging technique described in this paper are the following:

- It is able to detect incorrect use of pointers within recursive data structures.
- It is formally based on a (natural) operational semantics of the language.
- It offers an interactive assertion-based interface to the users.

This contrasts for instance with `lint` [13] which returns warnings concerning the use of uninitialised variables but does not check dereferences of pointers in recursive data structures. To our knowledge, no formal definition of the `lint` checker has been published either.

Of course no static pointer analysis can be complete and we decide to err on the conservative side: we show that the execution of a program that has passed our checking process cannot lead to an incorrect pointer dereference. The required approximation means that our checker can return warnings concerning safe programs.

Even if it cannot be complete, such a tool must be as accurate as possible. The user would otherwise be swamped with spurious warnings and the tool would be of little help. In particular, the tool must be able to return useful information about recursive data structures in the heap. Two significant features of our checker with respect to data structures are the following:

- It is able to treat recursive data structures in a non uniform way (indicating, for example, that a pointer variable x refers to the tail of the list which is also pointed to by another variable y).
- It is able to handle circular lists without introducing spurious aliases between different addresses in the list.

Such accuracy has a potentially high computational cost. In order to design a realistic tool, we have to find a compromise between efficiency and accuracy. We choose a flexible solution by offering an interactive interface to the user. The interaction is based on two kinds of assertions:

- static assertions whose consistency with respect to the properties inferred by the analyser is automatically verified, and
- dynamic assertions treated as additional hypotheses by the analyser and verified during program execution.

Such a solution presents several advantages:

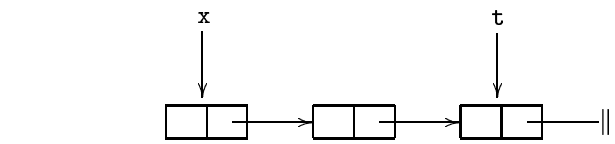
- The user can modify the trade-off between efficiency and accuracy. For example, he may decide to skip the analysis of a part of the program.
- Dynamic assertions permit to enhance the accuracy of the analyser by adding properties that cannot be inferred by the analyser.
- Static assertions are useful to locate precisely an error detected by the analyser.

In section 2, we briefly present the analyser that forms the basis of our debugger and recall the main results. Algorithmic concerns are discussed in section 3. The extension of the analyser with assertions is described in section 4. A dynamic verification of dynamic assertions is defined to lower the risk of errors that have passed the static checking phase due to the introduction of wrong assertions. Some examples illustrating different debugging scenarios are given in section 5. Section 6 reviews related work and section 7 suggests avenues for further research. The complete definition of the analyser described in section 2 and the abstract syntax and dynamic semantics of the subset of C considered in this paper are gathered in the appendix.

2 Specification of a static analyser

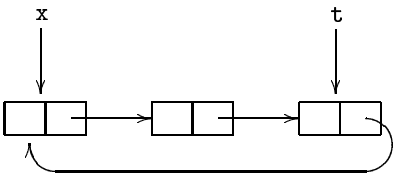
Let us first recall the static analysis described in [8], its possibilities in static debugging and the main correctness results. The goal of the analysis is the detection of pointers errors in presence of recursive data structures. The examples in Figure 1 and Figure 2 illustrate some of the possibilities of the analyser. For instance, in Figure 1, the analyser is able to prove that the dereference on `x` within the `while`-loop is always correct due to the condition `x!=t`.

The error on the double dereference on `x` is detected because the property `x=t` is true at the end of the loop. In Figure 2, the same program is applied on a circular list. In this case, the analyser is able to prove that the whole program is correct




```
while (x!=t)
     {
     ...
     x=*x.next;          ← correct dereference
     }
y=*(*x.next).next;       ←incorrect dereference
```

Figure 1: Iteration on a list




```
while (x!=t)
     {
     ...
     x=*x.next;          Correct program
     }
y=*(*x.next).next;
```

Figure 2: Iteration on a circular list

The analyser is defined with respect to a natural operational semantics of a subset of C. The language includes standard features of C (assignments, conditional, `while`-loop) and also instructions relative to the dynamic allocation and deallocation of memory cells (`alloc` and `free`). The syntax and semantics of the programming language considered in this paper are provided in Figure 15 and Figure 16 in the appendix. We use the exception value `illegal` to denote the result of a computation involving the dereference of an

invalid pointer. The language considered here does not include procedure calls and gotos but both are treated in the full version of the analysis described in [7]. We also ignore arithmetic operations on pointers and we assume that only one field of a record can be of type pointer. Due to this simplification, we can omit the field names in access chains without ambiguity (writing, for instance, $*v$ for `*v.cdr` if v is a variable of type `*list` with `list = struct car:int cdr:*list`).

The first part of this paper is concerned with the definition of the analyser. It is presented as a specialised Hoare logic for connectivity and aliasing properties of pointers. The sequents are triplets $\{P\}\ S\ \{Q\}$ where:

- S is a statement of the analysed program.
- P is a pre-condition expressed by a logical formula of our language of properties Prop (defined below). It characterises the state of the memory before the execution of S.
- Q is a post-condition in the same syntax as P. It characterises the state of the memory after the execution of S.

The class of properties Prop considered in this paper is defined in Figure 3.

$$
\begin{array}{lll}
\mathrm{P} & ::= & P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \neg P_1 \mid v_1 = v_2 \mid v_1 \mapsto v_2 \mid \mathrm{True} \mid \mathrm{False} \\
v & ::= & id \mid \&id \mid *id \mid \mathsf{undef} \\
\mathrm{P} & \in & \mathrm{Prop},\ v \in \mathrm{Var}
\end{array}
$$

Figure 3: Syntax of properties

In the sequel, we use the word "variable" to denote either undef or an access chain (that is to say an identifier *id* of the program possibly prefixed by a sequence of $*$ or &). P ranges over Prop, v ranges over the domain of variables Var and undef stands for the undefined location. As usual, $*v$ denotes the value contained at the address a where a is the value of v; $\&v$ is the address of v. The suffixes of a variable $*id$ are the variables id and $\&id$.

The semantics of properties is specified through a correspondence relation $\mathcal{C}_\mathcal{V}$ formalised in Figure 17 in the appendix. This semantics is parameterised with a set of variables $\mathcal{V} \subset$Var called the *reference set* in the sequel. This parameter can be used to tune the logic to get more or less accurate analyses. We impose only one constraint: $\mathcal{V}$ must contain the suffixes of all the variables assigned in the program (and the arguments of `free`). The correspondence relation $\mathcal{C}_\mathcal{V}(P, \mathcal{E}, \mathcal{S}_\mathcal{D})$ relates states of the memory to the properties they satisfy. The states of the memory are pairs $(\mathcal{E}, \mathcal{S}_\mathcal{D})$ where $\mathcal{E}$ is an environment (i.e. a function mapping identifiers to addresses), and $\mathcal{S}_\mathcal{D}$ is a store (i.e. function mapping addresses to values)). The intuition behind the correspondence relation is the following:

- $v_1 = v_2$ holds if the value of v_1 is equal to the value of v_2. In particular $*v_1 =$ undef means that the value of v_1 is an invalid pointer (which is the case if v_1 has not been initialised or if v_1 points to a cell which has been deallocated by `free`).

- $v_1 \mapsto v_2$ holds if the (address) value of v_2 is accessed from the (address) value of v_1 through at least one level of indirection and no (address) value of a variable of the reference set $\mathcal{V}$ appears in the path from v_1 to v_2. The last condition is crucial for the treatment of the assignment command.

Let us now turn to the axiomatisation itself. Space consideration prevents us from presenting all the system in detail (see Figure 19). We just consider three rules here to convey some intuition about the logic: the conditional, the dereference and the assignment.

- Conditional: The validity of the sequent $\{P\}$ `if` (E) S_1 `else` S_2 $\{Q\}$ depends on the validity of three Hoare sequents:
  - $\{P\}\ E\ \{P\}$ checks the validity of the derefences appearing in the test E, and
  - $\{P\}\ S_1\ \{Q_1\}$ and $\{P\}\ S_2\ \{Q_2\}$ correspond to the analysis of the two branches of the conditional.

  The rule for conditional is written:

$$\frac{\{P\}\ E\ \{P\} \quad \{P \wedge \overline{E}\}\ S_1\ \{Q_1\} \quad \{P \wedge \overline{!E}\}\ S_2\ \{Q_2\}}{\{P\}\ \texttt{if}\,(E)\ S_1\ \texttt{else}\ S_2\ \{Q_1 \vee Q_2\}}$$

  We can formulate two remarks about this rule:
  - the post-condition of the conditional is the disjunction of the post-conditions Q_1 and Q_2 of the two branches. This is because the analyser cannot determine statically which branch is going to be executed.
  - The test E is taken into account in the analysis of the two branches through the operation "$\overline{\ }$" which transforms a boolean C expression E into a property $\overline{E}$ in Prop. For example, the C operators `&&` and `||` are transformed into the logical "and" and "or" connectives. Of course, $\overline{E}$ is an approximation and it returns "True" if no pointer information can be extracted. The complete definition of the transformation "$\overline{\ }$" is given in Figure 21 in the appendix.

- Dereference: $\{P\}\ *v\ \{P\}$ if $P \Rightarrow \neg(*v = \mathsf{undef})$
  To check the validity of a dereference $*v$, it is sufficient to prove that $*v$ is not equal to the undefined value undef in the pre-condition. The verification is achieved via the partial order relation $\Rightarrow$ which is defined in Figure 18 in the appendix.

- Assignment: $\dfrac{\{P\}\ v_1\ \{P\} \quad \{P\}\ v_2\ \{P\} \quad P \Rightarrow Q[\![v_2/v_1]\!]_P^{\mathcal{V}}}{\{P\}\ v_1{=}v_2\ \{Q\}}$
  The rule corresponding to the assignment case is by far the most delicate to define. It needs a notion of substitution which must handle specific features of our language like aliasing and connectivity between pointer variables. The complete definition of the

substitution $[\![v_2/v_1]\!]^{\mathcal{V}}_P$ can be found in Figure 20 in the appendix. Roughly speaking, $Q[\![v_2/v_1]\!]^{\mathcal{V}}_P$ holds if Q holds when all occurrences of v_1 (and its initial aliases which are recorded in P) are replaced by v_2.

The following theorem establishes the soundness of the inference system:

THEOREM **2.1**

if $\{P\}\ S\ \{Q\}$ *then*

$$\forall \mathcal{E}, \forall \mathcal{S}_{\mathcal{D}}.\, \mathcal{C}_{\mathcal{V}}(P,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}})\ \textit{and}\ \mathcal{E} \vdash_{stat} <S, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}'_{\mathcal{D}'} \ \Rightarrow\ \mathcal{C}_{\mathcal{V}}(Q,\, \mathcal{E},\, \mathcal{S}'_{\mathcal{D}'})$$

This theorem expresses that for a valid sequent $\{P\}\ S\ \{Q\}$, if S is evaluated with initial state $(\mathcal{E}, \mathcal{S}_{\mathcal{D}})$ verifying the pre-condition property P then the resulting memory $(\mathcal{E}, \mathcal{S}'_{\mathcal{D}})$ verifies the post-condition property Q.

COROLLARY **2.2**

if $\{P\}\ S\ \{Q\}$ *then*

$$\forall \mathcal{E}, \forall \mathcal{S}_{\mathcal{D}}.\, \mathcal{C}_{\mathcal{V}}(P,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}) \ \Rightarrow\ \mathcal{E} \vdash_{stat} <S, \mathcal{S}_{\mathcal{D}}> \not\rightsquigarrow \texttt{illegal}.$$

Corollary 2.2 is a direct consequence of Theorem 2.1. It shows that the logic can be used to detect illegal pointer dereferences. The proof of Theorem 2.1 can be found in [7].

3 A static debugger for pointer errors

The system presented in the previous section is a specification of a static analyser. The next step consists in the derivation of an algorithm from the specification. The derivation follows the method presented in [10] to get a decidable and deterministic proof system. Choices have to be made among algorithmic alternatives during this step. In particular, we have to decide the direction of the analysis: it can be top-down and return the post-condition from the pre-condition or bottom-up and do the opposite.

Of course, the derivation of the algorithm from the specification involves some approximations. For example, we need to restrict the reference set $\mathcal{V}$ to be finite. Another approximation concerns the order relation $\Rightarrow$ which is very expensive to check; it is approximated by $\models\!\!\Rightarrow$ and properties are turned into a normal form to preserve the most complete information. The interested reader can find more information on this subject in [8]. The definition of $\models\!\!\Rightarrow$ is shown in Figure 4. A derivation of a correct top-down algorithm is presented in [8].

The algorithm produced as the result of this derivation is a useful static analyser for pointer errors. We have proved that programs which pass the check cannot lead to an

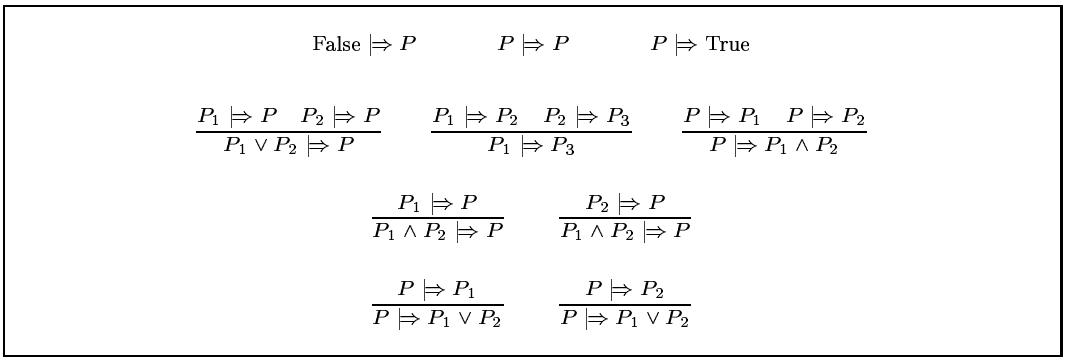


Figure 4: Deduction order over properties

incorrect pointer dereference. When an error is detected, the guilty dereference is returned to the programmer. Furthermore, due to the presence of disjunctions in properties, the analyser is able to distinguish between incorrect dereferences and unsafe ones. A dereference is reported to be incorrect when the analyser has proved that it certainly causes an error. An incorrect dereference is detected with the rule

$$\{P\} \ *id \ \{\text{incorrect dereference:} *id\} \quad \text{if } P \models\!\Rightarrow *id = \mathsf{undef}$$

A dereference is reported to be unsafe when the analyser has proved that it may cause an error. Unsafe dereferences are detected with the rule

$$\{P\} \ *id \ \{\text{unsafe dereference:} *id\} \quad \text{if } P \not\models\!\Rightarrow *id \neq \mathsf{undef}$$

Unsafe dereference reports are warnings. They can occur because of the approximations of our algorithm. For example, in general the length of a path between two variables cannot be known statically. Let us consider that the property $*x \mapsto y$ is implied by the pre-condition of the assignment `x = *x.next`. As we do not know the length of the path between $*x$ and y, the post-condition is split in the disjunction $*x \mapsto y \vee *x = y$. Furthermore, if y is equal to undef, further dereferences on x will be unsafe. Another case of approximation is when the source of a pointer error occurs in a branch of a conditional. As the analyser cannot determine statically which branch is going to be executed, we have seen in the previous section that the post-condition is a disjunction. The source of the bug thus is handled within a subterm of the disjunction.

As we have shown above, disjunctions provide a very high accuracy for the analysis. The counterpart is that the size of properties is exponential in space with respect to the number of variables of the reference set. Several solutions for the optimisation of the algorithm are discussed in section 7. These optimisations lead to important gains, which is crucial for the design of a realistic tool. However, we think that it is important to let the user decide the trade-off between accuracy and efficiency.

Another significant issue is the relevance of the information returned by the debugger. Our analyser is able to provide accurate reports on erroneous dereferences. But, in general, this is not sufficient to pinpoint the source of the bug. We argue that this part of a debugging session needs a narrow collaboration between the tool and the user. We formalise this interaction with user defined assertions. Such an interaction is useful for the location of errors; it also represents a realistic compromise between accuracy and efficiency: the user has the possibility to enhance the power of the analyser by writing additional hypotheses. He can also avoid the analysis of some part of the program as shown below.

4 Debugging with assertions

So far, we have focussed on the information returned to the user by the analyser. We present now the dual information flow in the dialogue with the user. Our system is extended in a natural way with a notion of user defined assertions without introducing any modification to the formalism defined previously. We distinguish between two kinds of assertions:

- static assertions, to be checked by the analyser via a decidable order relation, and
- dynamic assertions, which are considered by the analyser as additional hypotheses in the same way as tests in conditional and `while`-loop. They represent extra information about the execution context that are provided by the user. This kind of assertion can be verified at run time using the concretisation function $\mathcal{C}_{\mathcal{V}}$.

$$\frac{\{P\}\ S\ \{Q\} \quad P \models\!\!\Rightarrow A}{\{P\}\ [A <= S]\ \{Q\}}\ \text{[SPRE]} \qquad \frac{\{P\}\ S\ \{Q\} \quad Q \models\!\!\Rightarrow A}{\{P\}\ [S => A]\ \{Q\}}\ \text{[SPOST]}$$

$$\frac{P \wedge A \not\models\!\!\Rightarrow \text{False} \quad \{P \wedge A\}\ S\ \{Q\}}{\{P\}\ [A \vdash S]\ \{Q\}}\ \text{[DPRE]} \qquad \frac{\{P\}\ S\ \{Q\} \quad Q \wedge A \not\models\!\!\Rightarrow \text{False}}{\{P\}\ [S \dashv A]\ \{Q \wedge A\}}\ \text{[DPOST]}$$

Figure 5: Axiomatics of annotations

The syntax of an annotated statement for a static assertion A inserted before (resp. after) a C statement S is $[A <= S]$ (resp. $[S => A]$). The rules for static assertions are presented in Figure 5. We have two different rules, [SPRE] and [SPOST] corresponding to assertions inserted before or after a program point. The analyser checks that A can be deduced with the relation $\models\!\!\Rightarrow$ from the pre-condition P (resp. the post-condition Q). The important point is that the verification of static assertions is fully automatic because the order relation $\models\!\!\Rightarrow$ defined in the previous section is decidable.

The syntax for an annotated statement with dynamic assertion A is $[A- > S]$ (resp. $[S < -A]$) and its semantics is such that A is directly included in the pre-condition (reps. the post-condition). The rules for dynamic assertions [DPRE] and [DPOST] are presented

in Figure 5. As in the static assertions case, we can insert dynamic assertions before or after program points. Dynamic assertions are used to add hypotheses for the analyser. Their natural role is to provide information which cannot be inferred by the analyser. They also can be useful to avoid analysing some branches of the program by imposing test values at conditional points. It is then sometimes possible to avoid analysing `while` loops, which may reduce greatly the time required for the analysis. In this case, dynamic assertions can be seen as a kind of code relaxation (elimination of some parts of a program to avoid their execution during a debugging session). The interest of dynamic assertions is illustrated in section 5. The particularity of dynamic assertions is that the verification of their consistency with respect to the actual abstract state of the memory is limited. The proof of $P \wedge A \not\Rightarrow$False which checks this consistency is unable to ensure that all executions of the analysed program will lead to a concrete memory state which verifies the property A. This is because the analyser may not be able to prove the user defined dynamic assertions. In order to avoid the risk of errors that have passed the static checking phase due to the introduction of wrong assertions, we have extended the operational semantics of the language to integrate a dynamic verification. The extension of the semantics is shown in Figure 6. The verification is achieved through the use of the concretisation function $\mathcal{C}_{\mathcal{V}}$.

$$\frac{\mathcal{E} \vdash_{stat} <S, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}'_{\mathcal{D}'} \quad \mathcal{C}_{\mathcal{V}}(H, \mathcal{E}, \mathcal{S}_{\mathcal{D}})}{\mathcal{E} \vdash_{stat} <[H -> S], \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}'_{\mathcal{D}'}}$$

$$\frac{\mathcal{E} \vdash_{stat} <S, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}'_{\mathcal{D}'} \quad \mathcal{C}_{\mathcal{V}}(H, \mathcal{E}, \mathcal{S}'_{\mathcal{D}'})}{\mathcal{E} \vdash_{stat} <[S <- H], \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}'_{\mathcal{D}'}}$$

Figure 6: Operational semantics for dynamic assertions

$$\frac{\{P\}\ E\ \{P\} \quad \{P \wedge \overline{E}\}\ [S_1 =>> A]\ \{Q_1\} \quad \{P \wedge \overline{!E}\}\ [S_2 =>> A]\ \{Q_2\}}{\{P\}\ [\texttt{if}\ (E)\ S_1\ \texttt{else}\ S_2 =>> A]\ \{Q_1 \vee Q_2\}}$$

$$\frac{\{P\}\ E\ \{P\} \quad \{P \wedge \overline{E}\}\ [S =>> A]\ \{P\} \quad P \models\!\Rightarrow A}{\{P\}\ [\texttt{while}\ (E)\ S =>> A]\ \{P \wedge \overline{!E}\}}$$

$$\frac{\{P\}\ [S_1 =>> A]\ \{P'\} \quad \{P'\}\ [S_2 =>> A]\ \{Q\}}{\{P\}\ [S_1; S_2 =>> A]\ \{Q\}}$$

$$\frac{\{P\}\ S\ \{Q\} \quad Q \models\!\Rightarrow A}{\{P\}\ [S =>> A]\ \{Q\}}$$

Figure 7: Axiomatics of static invariants

In order to extend the user-friendliness of the interaction, we introduce a third kind of assertion, static invariants, which are in fact a particular case of static assertions. They allow the user to verify that a property holds throughout a part of a program. The syntax for a static invariant A in a subprogram S is $[S =>> A]$. The rules for static invariants are presented in Figure 7. We have not defined the counterpart of static invariants in the dynamic case because it does not seem to be useful at this stage. If this facility turned out to be useful, it would not be difficult to handle dynamic invariants, neither to provide corresponding operational semantics rules.

Since assertions have to be written by the programmer, it is important to provide a suitable high level language to define them. The particularities of the two kinds of assertions lead to the following remarks:

- The language of dynamic assertions must be directly translatable in the language of Prop since these assertions are included in the property inferred by the analyser.
- The language of static assertions is less constrained. Static assertions are verified with respect to properties inferred by the analyser via the $\models\!\!\Rightarrow$ relation but they are not mixed with them. It is then possible to define a more expressive language including, for example, recursively defined predicates. The introduction of recursivity permits to simulate a notion a transitivity on the $\mapsto$ predicate which was not possible in our language of properties in the analyser because of accuracy considerations. This extension allows us to define very natural predicates, "Path(v_1, v_2)" and "Loop(v)" which are defined in Figure 9. The only constraint is to verify that the validity of sequents of the form $P \Rightarrow A$ with $P \in$ Prop and $A \in$ SProp remains decidable. This result is established in Theorem 4.1

Languages of static and dynamic assertions are presented in Figure 8. Figure 9 describes the translation of the interface language into the language of properties Prop used by the analyser in addition of the definition of the two recursive predicates "Path" and "Loop". Note that the definition of the predicate "Path" includes a free variable w. This definition can be seen as a Horn clause where variables are implicitly universally quantified at the outermost level. This quantification is bounded because the values of all variables are assumed to be in the reference set.

DProp ::= Equal(v_1, v_2) | NEqual(v_1, v_2) | Alias(v_1, v_2) | NAlias(v_1, v_2)
| Direct(v_1, v_2) | NDirect(v_1, v_2)
| Dprop and Dprop | Dprop or Dprop
SProp ::= Dprop | Path(v_1, v_2) | Loop(v)
| SProp and SProp | SProp or SProp
v, v_1, $v_2 \in$ Var.

Figure 8: Language of annotations

Equal(v_1,v_2)	$\equiv$	$v_1 = v_2$
NEqual(v_1,v_2)	$\equiv$	$\neg(v_1 = v_2)$
Alias(v_1,v_2)	$\equiv$	$\&v_1 = \&v_2$
NAlias(v_1,v_2)	$\equiv$	$\neg(\&v_1 = \&v_2)$
Direct(v_1,v_2)	$\equiv$	$v_1 \mapsto v_2$
NDirect(v_1,v_2)	$\equiv$	$\neg(v_1 \mapsto v_2)$
Path(v_1,v_2)	$=$	Direct(v_1,v_2) or Direct(v_1, w) and Path(w, v_2)
Loop(v)	$=$	Path(v,v)

Figure 9: Meaning of annotations

The axiomatisation of the analyser extended with rules of Figure 5 still verifies Theorem 2.1. An important property to ensure is:

THEOREM **4.1**

The provability of $P \models\!\!\Rightarrow A$ with $P \in$Prop and $A \in$SProp is decidable.

The proof of this theorem is direct: we consider a finite set of variables, thus implicit quantifications are finitely bounded. Our logic is then equivalent to a propositional logic and hence is decidable.
Because of the presence of disjunctions in SProp, the verification of $P \models\!\!\Rightarrow A$ can be quite expensive. We do not treat this problem here but the interested reader can find more information on ways to tackle this problem in [11].

5 Examples

This section presents several examples illustrating the role of assertions in static debugging. We first consider a scenario of debugging using static assertions. The Josephus algorithm described in [15] and given in Figure 10 consists in two loops. The first one builds a circular list and the second one removes at each iteration one element from the circular list until only one is left (which points to itself.) We have slightly modified the code of the program to introduce a pointer error. We have suppressed the assignment `*t.next = *(*t.next).next` of the second loop. The role of this assignment is to remove the element captured by the pointer `x` from the circular list. Hence the variable `x` is kept inside the circular list. The statement `free(x)` breaks the circularity of the list and at the next iteration of the loop, the analyser detects that the dereference on `t` in the `for` loop is no more safe. The warning provided by the analyser concerns this dereference on `t` as shown in Figure 11. From the programmer point of view, this warning is surprising at this place because the list pointed to by `t` is supposed to be always circular in the `while` loop. A solution to locate the source of the error is to add the static invariant Loop(`t`) at the `while` loop to check that `t` always

```
typedef struct node *link;
struct node {
  int key;
  link next;
};
main()
{
  int i, n, m;
  link t, x;
  scanf("%d %d", &n, &m);
  t = (link) alloc(sizeof(struct node));
  *t.key = 1; x = t;
  for (i = 2; i <= n; ++i) {
      *t.next = (link) alloc(sizeof(struct node));
      t = *t.next;
      *t.key = i;
    }
  *t.next = x;
  while (t != *t.next) {
    for (i = 1; i <= m-1; ++i)
      t = *t.next;
    printf("%d ", *t.next.key);
    x = *t.next;
    *t.next = *(*t.next).next;
    free(x);
  }
  printf("%d\n", *t.key);
}
```

Figure 10: Josephus program

```
:
while (t!=*t.next)
  {   for (...)
          {
          t=*t.next;          Warning: unsafe dereference *t.next
          }
      x=*t.next;
      free(x);
  }
```

Figure 11: Detection of an unsafe dereference

points to a circular list. Figure 12 shows the result of the analysis with such assertion. The

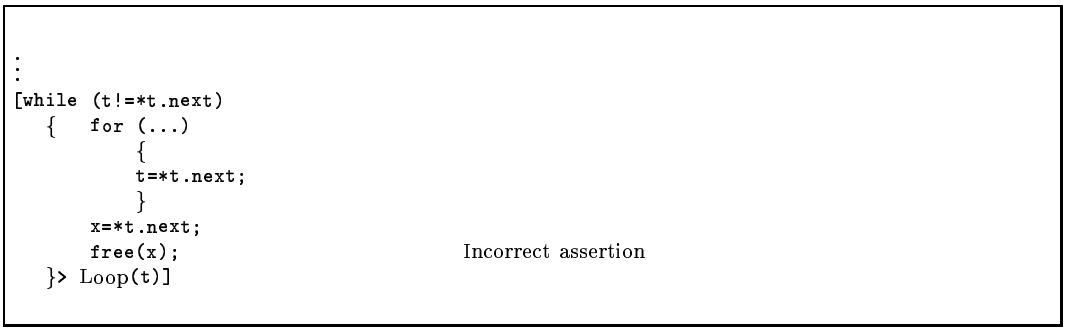


Figure 12: Detection of the wrong static assertion

error is localised on the `free` statement.

Let us now consider an example of an application of dynamic assertions. First, we would like to stress the fact that we implicitly focus on debugging with a forward analyser. In that case, dynamic assertions are propagated forward and we can observe their effects on the state of the program. Keeping this idea in mind, a natural application of these assertions is the localisation of the error in Figure 13. The error is reported after the conditional and is due to the `free` statement on the variable `x` which causes `*x.next` to be undefined. For the programmer which has not found the source of the error, it would be useful to detect if the bug is caused within one of the two branches of the conditional. A dynamic

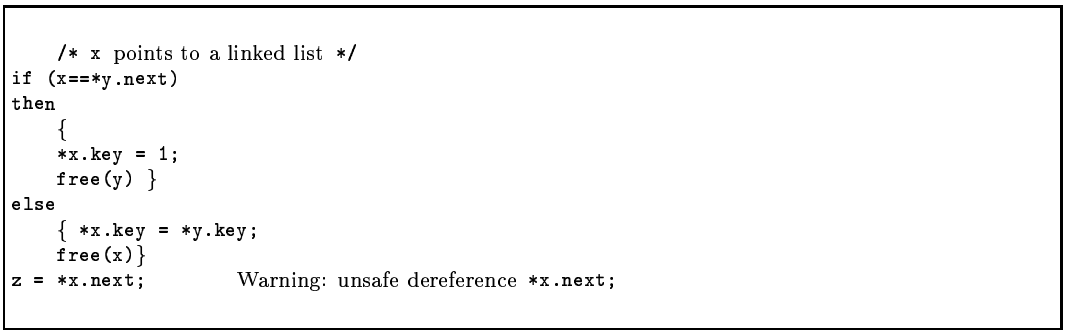


Figure 13: Error reported after a conditional.

assertion inserted before the conditional permits to relax one of the two branches and then to determine more precisely the origin of the bug. In Figure 14, the analyser finds that the annotated program is correct. Hence, the source of the bug is confined in the `else` branch of the conditional. If the code written in the `else` branch was longer, the detection process of the exact location of the error could use static assertions `*x.next= undef` to check the validity of the dereference.

```
[Equal(x,*y.next) ⊢ if (x==*y.next)
then
    {
    *x.key = 1;
    free(y) }
else
    { *x.key = *y.key;
    free(x)}]
z = *x.next;                                    Correct program
```

Figure 14: Localisation of the error.

Each kind of assertion has its specific use. Static assertions are used by the programmer to verify that a property holds at some program point. Dynamic assertions are useful to observe the influence of a property from a particular program point. Furthermore, they establish a link between static debugging and dynamic debugging. Dynamic assertions can be used to check if some properties hold or if some pieces of code are accessed at run time. We believe that this intimate integration of static analysis and dynamic analysis is a significant advantage of our framework.

6 Related work

In [4], Ducassé proposes a classification of debugging tools. Our work can be classified in her framework as a "verification with respect to specifications" strategy. In our setting, specifications gather the axiomatisation of pointer analysis and the dynamic assertions added by the programmer. In this particular debugging strategy, Ducassé identifies several techniques. Among them, our system handles the following:

- Symbolic derivation of assertions, which corresponds to the inference of properties of the analyser complemented with dynamic assertions.
- Consistency checking with respect to assertions, which corresponds to the verification of static assertions.
- Concrete evaluation of assertions, which corresponds to the dynamic verification of dynamic assertions.

Most related tools provide information about uninitialised variables but are unable to track illegal accesses in recursive data structures.

LCLint [6] and Aspects [12] are also tools using formal specifications to check programs but they both focus on the specification of the behaviour of procedures. They do not handle recursive data structures.

Typestates [16] present a special form of type checking performed by the NIL compiler. They express the degree of definition of pointer variables (uninitialised, initialised, defined) but not in recursive data structures.

Purify [9] and the work of Austin, Breach and Sohi [1] are less directly connected to our work but they illustrate a complementary aspect of the debugging activity. They are dynamic tools performing an instrumentation of the program code (object code for Purify and source code in [1]) to protect memory accesses (among others) so that runtime errors are detected as soon as possible during the execution. Like all testing tools, reported errors are only those occuring with the tested inputs. Our static debugger formally integrates some features of dynamic debugging via dynamic assertions verified at run time.

Another interesting debugging tool is Syntox [3], a static debugging tool for tracking out-of-bounds array indices in Pascal like languages. It is formally based on abstract interpretation and uses assertions added by the programmer. Assertions in Syntox are equivalent to our dynamic assertions but no solution is proposed to deal with incorrect assertions.

In the area of static debugging, the work whose spirit is the closest to our approach is the debugging framework ESC (Extended Static Checking) presented in [5]. They choose to detect a large variety of simple errors like nil dereferences or out-of-bounds array indices in Modula-3. The user annotates the program being checked with specifications corresponding to our dynamic assertions. A *verification condition generator* transforms the program and the specifications into a logical formula which is passed to an automatic theorem prover. The problem of incorrect specifications is not treated and the power of the tool when no specification is given is not discussed. As in other tools presented here, we find no equivalent for our static assertions.

Another related area is program verification. Most systems like [2] and the work of Luckham and Suzuki [14] focus on generality and completeness issues. The languages of properties they use encompass ours but they are too rich to support fully automatic analyses. Verification of properties is achieved by a theorem prover and depends heavily on user-supplied properties such as loop invariants.

7 Conclusion

We have presented a static debugger for the detection of pointer errors. The tool relies on a static analyser extended with facilities for interacting with the user. The interaction is based on a language of static and dynamic assertions, which offers a wide panoply of debugging strategies. We believe that the main contributions of our work are the following:

- The methodology has a firm semantics basis: an operational semantics is used to derive a correct analysis algorithm from the specification. Two different kind of assertions are formally defined and integrated within this formalism.
- The technique is powerful enough to track of pointer errors within recursive data structures.

- The use of a basis static analyser makes it possible to avoid complex interactions with general purpose theorem provers.
- Static and dynamic aspects of debugging are intimately integrated in our framework.

We are considering several improvements for our basis analyser like the integration of the results of extra analyses like liveness analyses and dependence analyses. An important source of inefficiency in our analysis is the use of disjunctions, which causes the size of properties to be exponential in space with respect to the number of variables of the program. In order to alleviate this problem, we have designed a new language of properties Prop' that does not include disjunctions while preserving a high level of accuracy by using more involved predicates. The formal correspondence between Prop' and Prop has been established (a property of Prop is approximated by a property of Prop') and an implementation of the tool based on this restricted set of properties is under way (this restriction has no impact on the interface language used for the interaction with the user).

References

[1] T. Austin, S. Breach and G. Sohi, *Efficient detection of all pointer and array access errors*, in Proceedings of the ACM SIGPLAN'94 Conf. on Programming Language Design and Implementation PLDI'94, Jun. 1994, pp. 290-301.

[2] R. S. Boyer and J. Moore, *A Computational Logic*, Academic Press, New York, NY, 1979.

[3] F. Bourdoncle, *Abstract debugging of higher-order imperative languages*, in Proceeding of the ACM SIGPLAN'93 Conf. on Programming Language Design and Implementation PLDI'93, Jun. 93.

[4] M. Ducassé, *A pragmatic survey of automated debugging*. P. Fritzson, editor, Proceedings of the first International Workshop on Automated and Algorithmic Debugging. May 1993. LNCS 749, Springer-Verlag.

[5] D. L. Detlefs, *An Overview of the Extended Static Checking System*, `http://www.research.digital.com/SRC/esc/Esc.html`.

[6] D. Evans, *Using specifications to check source code*, in Technical Report, MIT Lab for computer science, Jun. 1994.

[7] P. Fradet, R. Gaugne and D. Le Métayer, *An inference algorithm for the static verification of pointer manipulation*, INRIA Research Report 2895, Jun. 1996.

[8] P. Fradet, R. Gaugne and D. Le Métayer, *Static detection of pointer errors: an axiomatisation and a checking algorithm*, proc. European Symposium on Programming, ESOP'96, LNCS 1058, Apr. 1996.

[9] R. Hastings and B. Joyce, *Purify: Fast Detection of Memory Leaks and Access Errors*, in Proc. of the Winter Usenix Conference, Jan. 92.

[10] C. L. Hankin, D. Le Métayer, *Deriving algorithms from type inference systems: Application to strictness analysis*, proc. ACM Symposium on Principles of Programming Languages, 1994, pp. 202-212, Jan. 1994.

[11] I. Stéphan, *Nouvelles fondations pour la programmation en logique disjonctive*, thèse de doctorat, Université de Rennes 1, décembre 1995.

[12] D. Jackson, *Aspect: an economical bug-detector*, in Proceedings of 13^{th} International Conference on Software Engineering, May 1992, pp. 13-22.

[13] S. Johnson, *Lint, a C program checker*, Computer Science technical report, Bell Laboratories, Murray Hill, NH, July 1978.

[14] D. Luckham and N. Suzuki, *Verification of array, record, and pointer operations in Pascal*, in ACM Transactions on Programming Languages and Systems, Vol. 1, No.2, Oct. 1979, pp. 226-244.

[15] R. Sedgewick, *Algorithms*, Addison-Wesley publishing company, 1988.

[16] R. Strom and D. Yellin, *Extending typestate checking using conditional liveness analysis*, in IEEE Transactions on Software Engineering, Vol. 19, No 5, May. 93, pp. 478-485.

[17] J. Voas and K. Miller, *Putting Assertions in Their Place*, in Proc. of the Int'l. Symposium on Software Reliability Engineering, November 6–9, 1994, Monterey, CA.

Appendix

pgm	::=	stmt	
stmt	::=	**if** (exp) stmt **else** stmt	If-else
	|	**while** (exp) stmt	While loop
	|	stmt ; stmt	Sequence
	|	lexp = exp	Assignment
	|	**free** (lexp)	Runtime deallocation
	|	[Dprop $\vdash$ stmt]	Dynamic pre-assertion
	|	[stmt $\dashv$ Dprop]	Dynamic post-assertion
	|	[Sprop **|** stmt]	Static pre-assertion
	|	[stmt **|** Sprop]	Static post-assertion
	|	[stmt **>** Sprop]	Static invariant
exp	::=	*id*	Variable ($id \in$ Id)
	|	$*id$	Pointer dereference
	|	$\&id$	Address operator
	|	**alloc** (type)	Runtime allocation
lexp	::=	*id*	
	|	$*id$	

Figure 15: Abstract syntax of the programming language

[if-true] $$\frac{\mathcal{E} \vdash_{exp} <E, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <b, \mathcal{S}'_{\mathcal{D}'}> \quad \mathcal{E} \vdash_{stat} <S_1, \mathcal{S}'_{\mathcal{D}'}> \rightsquigarrow \mathcal{S}''_{\mathcal{D}''} \quad b \neq 0}{\mathcal{E} \vdash_{stat} <\texttt{if}\,(E)\ S_1\ \texttt{else}\,S_2, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}''_{\mathcal{D}''}}$$

[if-false] $$\frac{\mathcal{E} \vdash_{exp} <E, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <b, \mathcal{S}'_{\mathcal{D}'}> \quad \mathcal{E} \vdash_{stat} <S_2, \mathcal{S}'_{\mathcal{D}'}> \rightsquigarrow \mathcal{S}''_{\mathcal{D}''} \quad b = 0}{\mathcal{E} \vdash_{stat} <\texttt{if}\,(E)\ S_1\ \texttt{else}\,S_2, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}''_{\mathcal{D}''}}$$

[while-true] $$\frac{\mathcal{E} \vdash_{exp} <E, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <b, \mathcal{S}'_{\mathcal{D}'}> \quad \mathcal{E} \vdash_{stat} <S; \texttt{while}\,(E)\ S, \mathcal{S}'_{\mathcal{D}'}> \rightsquigarrow \mathcal{S}''_{\mathcal{D}''} \quad b \neq 0}{\mathcal{E} \vdash_{stat} <\texttt{while}\,(E)\ S, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}''_{\mathcal{D}''}}$$

[while-false] $$\frac{\mathcal{E} \vdash_{exp} <E, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <b, \mathcal{S}'_{\mathcal{D}'}> \qquad b = 0}{\mathcal{E} \vdash_{stat} <\texttt{while}\,(E)\ S, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}'_{\mathcal{D}'}}$$

[seq] $$\frac{\mathcal{E} \vdash_{stat} <S_1, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}'_{\mathcal{D}'} \quad \mathcal{E} \vdash_{stat} <S_2, \mathcal{S}'_{\mathcal{D}'}> \rightsquigarrow \mathcal{S}''_{\mathcal{D}''}}{\mathcal{E} \vdash_{stat} <S_1; S_2, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}''_{\mathcal{D}''}}$$

[assign] $$\frac{\mathcal{E} \vdash_{lexp} <v_1, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <a_1, \mathcal{S}'_{\mathcal{D}'}> \quad \mathcal{E} \vdash_{exp} <v_2, \mathcal{S}'_{\mathcal{D}'}> \rightsquigarrow <\mathrm{val}_2, \mathcal{S}''_{\mathcal{D}''}>}{\mathcal{E} \vdash_{stat} <v_1 = v_2, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}''_{\mathcal{D}''}[\mathrm{val}_2/a_1]}$$

[free] $$\frac{\mathcal{E} \vdash_{exp} <v, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <a, \mathcal{S}_{\mathcal{D}}>}{\mathcal{E} \vdash_{stat} <\texttt{free}(v), \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}_{\mathcal{D}'}} \quad a \in \mathcal{D},\ \mathcal{D}' = \mathcal{D} - \{a\}$$

[illegal] $$\mathcal{E} \vdash_{stat} <S, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \texttt{illegal} \quad \text{otherwise (access to } a \notin \mathcal{D})$$

Definition of $\vdash_{exp}$

[var] $$\mathcal{E} \vdash_{exp} <id, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <\mathcal{S}_{\mathcal{D}}(\mathcal{E}(id)), \mathcal{S}_{\mathcal{D}}> \quad \mathcal{E}(id) \in \mathcal{D}$$

[indr] $$\frac{\mathcal{E} \vdash_{exp} <id, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <a, \mathcal{S}'_{\mathcal{D}'}>}{\mathcal{E} \vdash_{exp} <*id, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <\mathcal{S}'_{\mathcal{D}'}(a), \mathcal{S}'_{\mathcal{D}'}>} \quad a \in \mathcal{D}'$$

[address] $$\frac{\mathcal{E} \vdash_{lexp} <id, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <a, \mathcal{S}'_{\mathcal{D}'}>}{\mathcal{E} \vdash_{exp} <\&id, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <a, \mathcal{S}'_{\mathcal{D}'}>}$$

[alloc] $$\mathcal{E} \vdash_{exp} <\texttt{alloc}(T), \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <a, \mathcal{S}'_{\mathcal{D}'}> \quad a \notin \mathcal{D},\ \mathcal{D}' = \mathcal{D} + \{a\},\ \mathcal{S}'_{\mathcal{D}'} = \mathcal{S}_{\mathcal{D}} + \{a \rightarrow \bot\}$$

[unary] $$\frac{\mathcal{E} \vdash_{exp} <E, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <\mathrm{val}, \mathcal{S}'_{\mathcal{D}'}>}{\mathcal{E} \vdash_{exp} <op\,E, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <\mathcal{O}(op)(\mathrm{val}), \mathcal{S}'_{\mathcal{D}'}>}$$

[binary] $$\frac{\mathcal{E} \vdash_{exp} <E_1, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <\mathrm{val}_1, \mathcal{S}'_{\mathcal{D}'}> \quad \mathcal{E} \vdash_{exp} <E_2, \mathcal{S}'_{\mathcal{D}'}> \rightsquigarrow <\mathrm{val}_2, \mathcal{S}''_{\mathcal{D}''}>}{\mathcal{E} \vdash_{exp} <E_1 op\,E_2, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <\mathcal{O}(op)(\mathrm{val}_1, \mathrm{val}_2), \mathcal{S}''_{\mathcal{D}''}>}$$

[illegal] $$\mathcal{E} \vdash_{exp} <E, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <\bot, \texttt{illegal}> \quad \text{otherwise (access to } a \notin \mathcal{D})$$

Definition of $\vdash_{lexp}$

[var] $$\mathcal{E} \vdash_{lexp} <id, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <\mathcal{E}(id), \mathcal{S}_{\mathcal{D}}>$$

[indr] $$\frac{\mathcal{E} \vdash_{exp} <id, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <a, \mathcal{S}'_{\mathcal{D}'}>}{\mathcal{E} \vdash_{lexp} <*id, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <a, \mathcal{S}'_{\mathcal{D}'}>}$$

$\mathcal{S}_{\mathcal{D}} : (\mathcal{D} \rightarrow \mathrm{Val}) + \{\texttt{illegal}\}$, $\mathcal{E} : \mathrm{Id} \rightarrow \mathrm{Adr}$,
$\mathcal{D} \subset$Adr, Val = Base+Adr, Base = Bool+Int+...
$id \in$ Id, $a \in$ Adr, val $\in$ Val

Figure 16: Dynamic semantics for statements and expressions

$$
\begin{array}{rcl}
\mathcal{C}_{\mathcal{V}}(P,\,\mathcal{E},\,\texttt{illegal}) & = & \mathit{false} \\
\mathcal{C}_{\mathcal{V}}(v_1 = v_2,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) & = & \mathit{Val}(v_1,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) = \mathit{Val}(v_2,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) \\
\mathcal{C}_{\mathcal{V}}(v_1 \mapsto v_2,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) & = & \exists \alpha_1 \ldots \alpha_k,\, \alpha_1 = \mathit{Val}(v_1,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}),\, \alpha_k = \mathit{Val}(v_2,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}), \\
 & & \forall i\ (1 \leq i < k)\ \mathcal{S}_{\mathcal{D}}(\alpha_i) = \alpha_{i+1} \\
 & & \forall i\ (1 < i < k),\ \forall v \in \mathcal{V},\ \alpha_i \neq \mathit{Val}(v,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) \\
\mathcal{C}_{\mathcal{V}}(P_1 \wedge P_2,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) & = & \mathcal{C}_{\mathcal{V}}(P_1,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}})\ \underline{\text{and}}\ \mathcal{C}_{\mathcal{V}}(P_2,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) \\
\mathcal{C}_{\mathcal{V}}(P_1 \vee P_2,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) & = & \mathcal{C}_{\mathcal{V}}(P_1,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}})\ \underline{\text{or}}\ \mathcal{C}_{\mathcal{V}}(P_2,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) \\
\mathcal{C}_{\mathcal{V}}(\neg P,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) & = & \underline{\text{not}}\ (\mathcal{C}_{\mathcal{V}}(P,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}})) \\
\mathcal{C}_{\mathcal{V}}(\text{True},\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) & = & \mathit{true} \\
\mathcal{C}_{\mathcal{V}}(\text{False},\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) & = & \mathit{false} \\
 & & \\
\mathit{Val}(\textsf{undef},\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) & = & \bot \\
\mathit{Val}(\&id,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) & = & \mathcal{E}(id) \\
\mathit{Val}(id,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) & = & \mathcal{S}_{\mathcal{D}}(\mathcal{E}(id)) \\
\mathit{Val}(*id,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}) & = & \mathcal{S}_{\mathcal{D}}(\mathit{Val}(id,\,\mathcal{E},\,\mathcal{S}_{\mathcal{D}}))
\end{array}
$$

Figure 17: Correspondence relation

$$
(v_1 = v_2) \wedge P \;\Rightarrow\; P[v_2/v_1] \qquad \&{*}v = v \qquad {*}\&v = v \qquad v_1 = v_2 \Rightarrow {*}v_1 = {*}v_2
$$

$$
v = v \qquad v_1 = v_2 \wedge v_2 = v_3 \Rightarrow v_1 = v_3 \qquad v_1 = v_2 \Rightarrow v_2 = v_1
$$

$$
v \mapsto {*}v \qquad v_1 \mapsto v_2 \Rightarrow (v_2 = {*}v_1) \vee ({*}v_1 \mapsto v_2) \qquad x = \textsf{undef} \Rightarrow {*}x = \textsf{undef}
$$

$$
v_1 \mapsto w \wedge w \mapsto v_2 \Rightarrow v_1 \mapsto v_2 \quad \text{with } w \notin \mathcal{V}
$$

$$
\text{False} \Rightarrow P \qquad P \Rightarrow P \qquad P \Rightarrow \text{True}
$$

$$
\frac{P_1 \Rightarrow P \quad P_2 \Rightarrow P}{P_1 \vee P_2 \Rightarrow P} \qquad \frac{P_1 \Rightarrow P_2 \quad P_2 \Rightarrow P_3}{P_1 \Rightarrow P_3} \qquad \frac{P \Rightarrow P_1 \quad P \Rightarrow P_2}{P \Rightarrow P_1 \wedge P_2}
$$

$$
P_1 \wedge P_2 \Rightarrow P_1 \qquad P_1 \wedge P_2 \Rightarrow P_2 \qquad P_1 \Rightarrow P_1 \vee P_2 \qquad P_2 \Rightarrow P_1 \vee P_2
$$

Figure 18: Partial order and equivalences on properties (w.r.t $\mathcal{V}$)

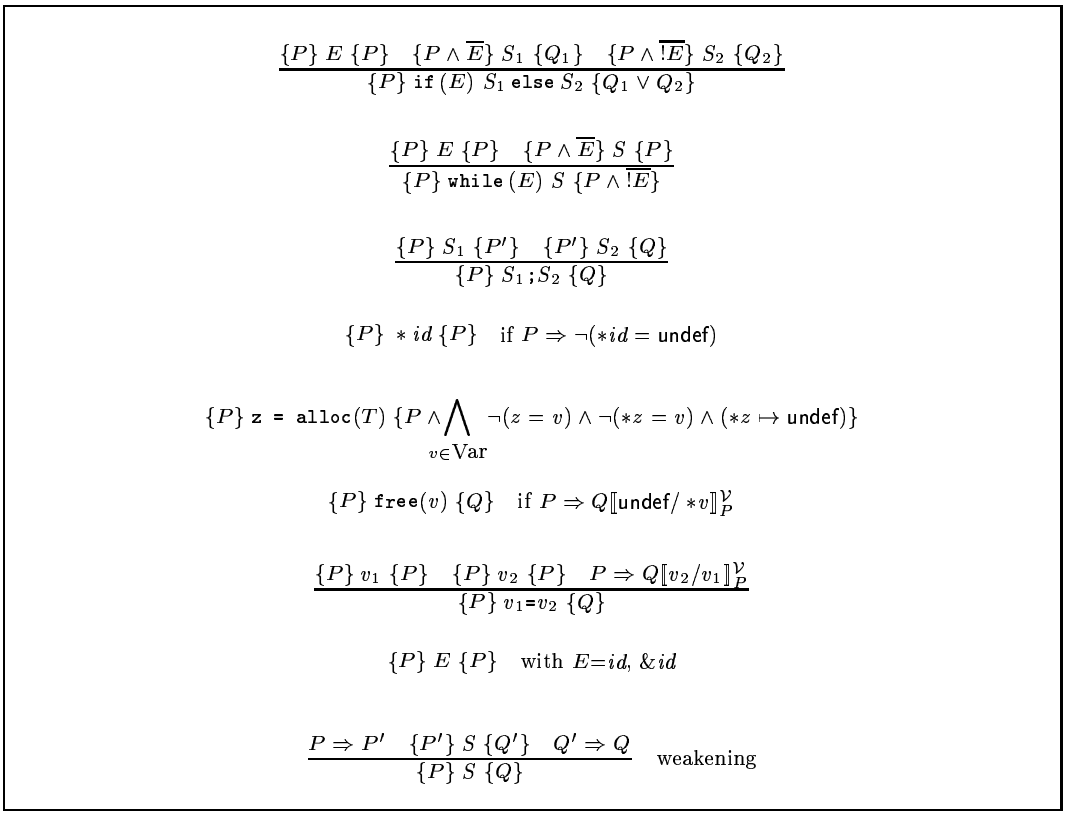


Figure 19: Axiomatics of statements and expressions

$$
\begin{array}{rcl}
(Q_1 \wedge Q_2)[\![v_2/v_1]\!]_P^{\mathcal{V}} & = & (Q_1[\![v_2/v_1]\!]_P^{\mathcal{V}}) \wedge (Q_2[\![v_2/v_1]\!]_P^{\mathcal{V}}) \\
(Q_1 \vee Q_2)[\![v_2/v_1]\!]_P^{\mathcal{V}} & = & (Q_1[\![v_2/v_1]\!]_P^{\mathcal{V}}) \vee (Q_2[\![v_2/v_1]\!]_P^{\mathcal{V}}) \\
(\neg Q)[\![v_2/v_1]\!]_P^{\mathcal{V}} & = & \neg(Q[\![v_2/v_1]\!]_P^{\mathcal{V}}) \\
(v = v')[\![v_2/v_1]\!]_P^{\mathcal{V}} & = & v[\![v_2/v_1]\!]_P^{\mathcal{V}} = v'[\![v_2/v_1]\!]_P^{\mathcal{V}} \\
(v \mapsto v')[\![v_2/v_1]\!]_P^{\mathcal{V}} & = & [((\tilde{v} \mapsto \tilde{v}') \wedge \neg(\tilde{v} = \&v_1)) \\
 & & \vee((\tilde{v} \mapsto w \mapsto \tilde{v}') \wedge \neg(\tilde{v} = \&v_1) \wedge \neg(w = \&v_1))] \\
 & & \wedge[\forall x \in \mathcal{V},\ (\neg(\tilde{v} \mapsto \tilde{x}) \vee (\tilde{x} = \tilde{v}') \vee\ \neg(\tilde{x} \mapsto \tilde{v}'))] \\
 & & \text{with } \tilde{x} = x[\![v_2/v_1]\!]_P^{\mathcal{V}},\ \tilde{v} = v[\![v_2/v_1]\!]_P^{\mathcal{V}} \text{ and } \tilde{v}' = v'[\![v_2/v_1]\!]_P^{\mathcal{V}} \\
\mathrm{True}[\![v_2/v_1]\!]_P^{\mathcal{V}} & = & \mathrm{True} \\
\mathrm{False}[\![v_2/v_1]\!]_P^{\mathcal{V}} & = & \mathrm{False} \\
\&id[\![v_2/v_1]\!]_P^{\mathcal{V}} & = & \&id \\
id[\![v_2/v_1]\!]_P^{\mathcal{V}} & = & v_2 \text{ if } P \Rightarrow (\&id = \&v_1) \\
 & = & id \text{ if } P \Rightarrow \neg(\&id = \&v_1) \\
*id[\![v_2/v_1]\!]_P^{\mathcal{V}} & = & v_2 \text{ if } P \Rightarrow (id[\![v_2/v_1]\!]_P^{\mathcal{V}} = \&v_1) \\
 & = & *(id[\![v_2/v_1]\!]_P^{\mathcal{V}}) \text{ if } P \Rightarrow \neg(id[\![v_2/v_1]\!]_P^{\mathcal{V}} = \&v_1) \\
\mathsf{undef}[\![v_2/v_1]\!]_P^{\mathcal{V}} & = & \mathsf{undef}
\end{array}
$$

Figure 20: Definition of substitution with aliasing

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\overline{E_1 \texttt{\&\&} E_2} & = & \overline{E_1} \wedge \overline{E_2} & \overline{E_1 \texttt{||} E_2} & = & \overline{E_1} \vee \overline{E_2} & \overline{!(v_1 \texttt{!=} v_2)} & = & v_1 = v_2 \\
\overline{!(E_1 \texttt{\&\&} E_2)} & = & \overline{!E_1} \vee \overline{!E_2} & \overline{!(E_1 \texttt{||} E_2)} & = & \overline{!E_1} \wedge \overline{!E_2} & \overline{!(v_1 \texttt{==} v_2)} & = & \neg(v_1 = v_2) \\
\overline{v_1 \texttt{==} v_2} & = & v_1 = v_2 & \overline{v_1 \texttt{!=} v_2} & = & \neg(v_1 = v_2) & & & \\
\overline{E} & = & \mathrm{True} & \text{otherwise} & & & & &
\end{array}
$$

Figure 21: Definition of $\overline{E}$

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399