An Inference Algorithm for the Static Verification of Pointer Manipulation

Pascal Fradet, Ronan Gaugne, Daniel Le Métayer

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An inference algorithm for the static verification of pointer manipulation

Pascal Fradet, Ronan Gaugne and Daniel Le Métayer

N ° 2895

juin 1996

THÈME 2

An inference algorithm for the static verification of pointer manipulation

Pascal Fradet, Ronan Gaugne and Daniel Le Métayer *

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n ° 2895 — juin 1996 — 45 pages

Abstract: The incorrect use of pointers is one of the most common source of bugs. As a consequence, any kind of static code checking capable of detecting potential bugs at compile time is welcome. This paper presents a static analysis for the detection of incorrect accesses to memory (dereferences of 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 analyser is derived from an axiomatisation of alias and connectivity properties which is shown to be sound with respect to the natural semantics of the language. It deals with dynamically allocated data structures and it is accurate enough to handle circular structures.

Key-words: alias analysis, debugging tool, Hoare logic, correctness proof.

(Résumé : tsvp)

* [fradet,gaugne,lemetayer]@irisa.fr

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

Un algorithme d'inférence pour la vérification statique de manipulation de pointeurs

Résumé : L'utilisation incorrecte de pointeurs est une des sources d'erreurs les plus répandues. Par conséquent, tout vérificateur statique de code capable de détecter des erreurs potentielles à la compilation est bienvenu. Cet article présente une analyse statique pour la détection d'accès incorrects à la mémoire (déréférences de pointeurs invalides). Un pointeur peut être invalide parce qu'il n'a pas été initialisé ou parce qu'il réfère à une cellule de la mémoire qui a été désallouée. L'analyseur est dérivé à partir d'une axiomatisation des propriétés d'alias et de connectivité qui est prouvée correcte par rapport à la sémantique naturelle du langage. Il prend en compte les structures de données dynamiques et il est suffisamment précis pour traiter les structures circulaires.

Mots-clé : analyse d'alias, outil de détection d'erreurs, logique de Hoare, preuve de correction.

1 Introduction

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

- Most widely used programming languages allow explicit pointer manipulations. The expressiveness provided by such features is appreciated by many programmers because it makes it 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.
- It is more economical to detect bugs at compile time than by running test cases. Testing is a very expensive activity: bugs have first to be discovered, then they must be localised within the source program. As a consequence, any kind of static code checking capable of detecting bugs at compile time is welcome. Type checking is an example of a static analysis technique which has proved greatly beneficial in terms of program development.

The technique described in this paper is applied to the detection of incorrect accesses to memory (dereferences of 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. Other applications are suggested in the conclusion. A large amount of literature is devoted to the analysis of pointers for compiler optimisations but there has been comparatively fewer contributions aiming at static bug detection. The main features of the analysis 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.
- The analyser is derived from an axiomatisation of alias and connectivity properties.

This contrasts for instance with `lint` 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 successfully cannot lead to an incorrect pointer dereference. The required approximation means that our checker can return warnings concerning safe programs. The checker can be seen as a static debugging tool, helping the programmer to focus on the pieces of code that cannot be trusted.

Even if it cannot be complete, such a tool must be as accurate as possible. Otherwise the user would 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 pointed to by another variable y).
- It is able to handle circular lists without introducing spurious aliases between different addresses in the list.

We focus in this paper on the formal definition of the inference algorithm and its relationship to the axiomatics and the natural semantics. The algorithm presented here is only a first step towards the design of an effective tool. Current work to get a more efficient algorithm is sketched in the conclusion.

In section 2 we present an inference system for proving properties about pointers such as (may and must) aliasing and reachability. We establish its correctness with respect to a natural semantics of the language. The inference system can be seen as a Hoare logic specialised for explicit pointer manipulation. This logic is not decidable in general and we define in section 3 appropriate restrictions to make the set of properties finite, which allows us to design a checking algorithm. Section 4 illustrates the algorithm with an example involving the construction and the destruction of a circular list. Section 5 outlines the generalisation of the technique to procedures and jumps. Section 6 reviews related work and suggests optimisations and other applications of the analysis. Appendix 1 contains the abstract syntax and the dynamic semantics of the subset of C considered in this paper. The correctness proofs of the main results are gathered into appendix 2 and appendix 3.

2 A Hoare Logic for Pointers

The syntax and semantics of the subset of C considered in this paper are provided in Fig. 11 and 12 in appendix 1. They are variations of definitions appearing in [3]. We use the exception value `illegal` to denote the result of a computation involving the dereference of an invalid pointer. The set of valid pointers of the store $\mathcal{S}_{\mathcal{D}}$ is $\mathcal{D}$. The effect of `alloc` (resp. `free`) is to add an address in (resp. to remove an address from) $\mathcal{D}$.

The first part of this paper is concerned with the analysis of blocks of instructions excluding procedure calls and gotos. This allows us to focus on the essential issues of pointer analysis and to keep the presentation simpler. 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 class of properties Prop considered in this paper is defined as follows:

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

In the sequel, we use the word "variable" to denote undef or an access chain (that is to say an identifier *id* of the program possibly prefixed by $*$ 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 meaning of properties is specified through a correspondence relation $\mathcal{C}_\mathcal{V}$ defined in Fig. 1. 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 (that is to say, pairs $(\mathcal{E}, \mathcal{S}_\mathcal{D})$ with $\mathcal{E}$ an environment, and $\mathcal{S}_\mathcal{D}$ a store) to the properties they satisfy. The intuition behind this correspondence 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.

$$
\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}(\texttt{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 1: Correspondence relation

Due to the presence of the negation and disjunction connectors, and the meaning of the = operator, our logic is able to deal with "must-alias" properties as well as "may-alias" properties. This allows us to retain a better level of accuracy, which is required to analyse

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

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

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

$$(v_1 \mapsto v_2) \wedge \neg(v_2 = v_3) \Rightarrow \neg(v_1 \mapsto v_3)$$

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

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

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

Figure 2: Partial order and equivalences on properties (w.r.t $\mathcal{V}$)

the kind of correctness-related properties we are interested in in this paper. We introduce a partial order on properties in Fig. 2. Note that $v_1 \mapsto w \wedge w \mapsto v_2 \Rightarrow v_1 \mapsto v_2$ holds only if w does not belong to the reference set (this follows the semantics of $\mapsto$, which is not transitive).
We define the transformation "$\overline{\ }$" which transforms a boolean C expression E into a property $\overline{E}$ in Prop. It is used to extract properties from tests in C programs. 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.

DEFINITION **2.1**

$$\begin{array}{rclrcl}
\overline{E_1\texttt{\&\&}E_2} & = & \overline{E_1} \wedge \overline{E_2} & \overline{E_1\texttt{||}E_2} & = & \overline{E_1} \vee \overline{E_2} \\
\overline{\texttt{!}(E_1\texttt{\&\&}E_2)} & = & \overline{\texttt{!}E_1} \vee \overline{\texttt{!}E_2} & \overline{\texttt{!}(E_1\texttt{||}E_2)} & = & \overline{\texttt{!}E_1} \wedge \overline{\texttt{!}E_2} \\
\overline{v_1\texttt{==}v_2} & = & v_1 = v_2 & \overline{v_1\texttt{!=}v_2} & = & \neg(v_1 = v_2) \\
\overline{E} & = & \mathit{True} & \mathit{otherwise} & &
\end{array}$$

$$\overline{\texttt{!}(v_1\texttt{!=}v_2)} = v_1 = v_2 \qquad \overline{\texttt{!}(v_1\texttt{==}v_2)} = \neg(v_1 = v_2)$$

The inference system for statements and expressions is presented in Fig. 3. Let us focus on the rules of Fig. 3 which depart from traditional Hoare logic.

- The rule for the conditional makes use of the transformation $\overline{E}$ in order to take the conditions on pointers into account when analysing the two branches. This degree of accuracy is necessary in order to prevent the analyser from generating too many spurious warnings.
- As expected, the rule for dereference ($*id$) includes a check that the pointer is valid.
- We assume that a preliminary transformation of the source program has replaced the assignments v`=alloc(T)` by the sequence {`z=alloc(T);`v`=z;free(z)`} where `z` is a

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

$$\frac{\{P\}\ E\ \{P\} \quad \{P \wedge \overline{E}\}\ S\ \{P\}}{\{P\}\ \texttt{while}\,(E)\ S\ \{P \wedge \overline{!E}\}}$$

$$\frac{\{P\}\ S_1\ \{P'\} \quad \{P'\}\ S_2\ \{Q\}}{\{P\}\ S_1\,;S_2\ \{Q\}}$$

$$\frac{\{P\}\ v_1\ \{P\} \quad \{P\}\ v_2\ \{P\} \quad P \Rightarrow Q[\![v_2/v_1]\!]^{\mathcal{V}}_P}{\{P\}\ v_1\texttt{=}v_2\ \{Q\}}$$

$$\{P\}\ \texttt{free}(v)\ \{Q\} \quad \text{if } P \Rightarrow Q[\![\texttt{undef}/\,*v]\!]^{\mathcal{V}}_P$$

$$\{P\}\ E\ \{P\} \quad \text{with } E{=}id,\ \&id$$

$$\{P\}\ *id\ \{P\} \quad \text{if } P \Rightarrow \neg(*id = \texttt{undef})$$

$$\{P\}\,\texttt{z = alloc}(T)\ \{P \wedge \bigwedge_{v \in \mathrm{Var}(P) - \{z, *z\}} \neg(z = v) \wedge \neg(*z = v) \wedge \neg(z = *z) \wedge (*z \mapsto \texttt{undef})\}$$

$$\frac{\{P_1\}\ S\ \{P_1'\} \quad \{P_2\}\ S\ \{P_2'\}}{\{P_1 \vee P_2\}\ S\ \{P_1' \vee P_2'\}} \quad \text{disjunction}$$

$$\frac{P \Rightarrow P' \quad \{P'\}\ S\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}} \quad \text{weakening}$$

Figure 3: Axiomatics of statements and expressions

new variable. This can always be done without altering the meaning of the program. The rule for `alloc` shows that the allocated address z is different from the values of all other variables and the pointer contained at address z is invalid. The effect of `free` is to set the deallocated cell to `undef`. So `free` is treated very much like the assignment.

- The rule for assignment is more involved than the usual Hoare logic rule. This is because aliasing (in both sides of the assignment) has to be taken into account. The definition of $Q[\![v_2/v_1]\!]^{\mathcal{V}}_P$ can be found in Fig. 4. 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. In all cases except $v \mapsto v'$, the substitution $[\![v_2/v_1]\!]^{\mathcal{V}}_P$ is propagated through the property and applied to the variables which are aliases of v_1. The fact that x and y are aliases is expressed by $P \Rightarrow (\&x = \&y)$ in our setting (see the rule for $id[\![v_2/v_1]\!]^{\mathcal{V}}_P$ for instance). The case for $\mapsto$ is more involved because three properties are checked in order to show that $v \mapsto v'$ holds after an assignment $v_1 = v_2$:

(1) There is a path from $\tilde{v}$ to $\tilde{v}'$.

(2) The path is not affected by the assignment.

(3) The assignment does not introduce any element of $\mathcal{V}$ on the path.

Properties $(\tilde{v} \mapsto \tilde{v}')$ and $(\tilde{v} \mapsto w \mapsto \tilde{v}')$ ensure (1) and the disjunction $[\forall x \in \mathcal{V}, ...]$ establishes (3). Property (2) follows from $\neg(\tilde{v} = \&v_1)$ and $\neg(w = \&v_1)$. Due to our restriction on $\mathcal{V}$, all assigned variables v_1 belong to $\mathcal{V}$; thus v_1 cannot be on paths $\tilde{v} \mapsto \tilde{v}'$ or $w \mapsto \tilde{v}'$ except if $\tilde{v} = \&v_1$ or $w = \&v_1$. Since these two cases are excluded, the assignment cannot have any effect on these paths.

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

Figure 4: Definition of substitution with aliasing

The following theorems establish the soundness of the inference system:

THEOREM **2.2**

if $\{P\}\ S\ \{Q\}$ *can be proven using the rules of Fig. 3 then*

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

COROLLARY **2.3**

if $\{P\}\ S\ \{Q\}$ *can be proven using the rules of Fig. 3 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 2.3 is a direct consequence of Theorem 2.2. It shows that the logic can be used to detect illegal pointer dereferences. The proof of Theorem 2.2 is made by induction on the structure of proof tree. The most difficult part of the proof is the assignment case which relies on the following lemma:

LEMMA **2.4**

$$\begin{array}{l}\mathcal{C}_\mathcal{V}(Q,\ \mathcal{E},\ \mathcal{S}_\mathcal{D}) \\ \Longrightarrow\ Val(v[\![v_2/v_1]\!]^\mathcal{V}_Q,\ \mathcal{E},\ \mathcal{S}_\mathcal{D}) = Val(v,\ \mathcal{E},\ \mathcal{S}_\mathcal{D}[\,Val(v_2,\ \mathcal{E},\ \mathcal{S}_\mathcal{D})/Adr(v_1,\ \mathcal{E},\ \mathcal{S}_\mathcal{D})])\end{array}$$

Lemma 2.4 can be proven by inspection of the different cases in the definition of $v[\![v_2/v_1]\!]^\mathcal{V}_P$. The correctness of the dereference case $(*id)$ follows from the lemma:

LEMMA **2.5**

$$\mathcal{C}_\mathcal{V}(\neg(*v = \mathsf{undef}),\ \mathcal{E},\ \mathcal{S}_\mathcal{D}) \Rightarrow Val(v,\ \mathcal{E},\ \mathcal{S}_\mathcal{D}) \in \mathcal{D}$$

The proof of Theorem 2.2 is sketched in Appendix 2.

3 A Checking Algorithm

As a first stage to get an effective algorithm from the previous logic, we restrict the set of properties which may appear as pre/post-conditions. For a given program "Prog", let us call $\mathrm{Var}_{\mathrm{Prog}}$ the set of variables[1] occurring in Prog and their suffixes (plus undef). For the analysis of Prog, we take $\mathrm{Var}_{\mathrm{Prog}}$ as the reference set and consider only the properties involving variables in $\mathrm{Var}_{\mathrm{Prog}}$. Proceeding this way, we get a finite set of properties tailored to the program to be analysed.

In order to avoid the need for the last two rules of Fig. 3 (disjunction and weakening), we consider properties in atomic disjunctive normal form:

[1] We remind the reader that we use the word "variable" to denote an identifier of the program possibly prefixed by $*$ or &.

DEFINITION **3.1** *A property P is said to be in atomic disjunctive normal form (adnf) if it is of the form $\bigvee P_i$ where $P_i = A_1 \wedge \ldots \wedge A_n$, A_k being basic properties $(x = y)$, $(x \mapsto y)$ or negations of those, and each P_i is such that:*

$$\begin{array}{lll} \forall x, y \in Var_{Prog} & \textit{either } P_i \models\!\!\Rightarrow x = y \quad \textit{or} & P_i \models\!\!\Rightarrow \neg(x = y) \\ & \textit{either } P_i \models\!\!\Rightarrow x \mapsto y \quad \textit{or} & P_i \models\!\!\Rightarrow \neg(x \mapsto y) \end{array}$$

with $\models\!\!\Rightarrow$ defined as follows:

$$P \models\!\!\Rightarrow P \qquad P_1 \wedge P_2 \models\!\!\Rightarrow P_1 \qquad P_1 \wedge P_2 \models\!\!\Rightarrow P_2$$

The intuition is that a property in atomic disjunctive normal form records explicitly all basic properties for all possible memory states. As a consequence, implication boils down to the extraction of subproperties.
To handle *adnf* properties, this implication is extended in a natural way to disjunctions:

$$\frac{P_1 \models\!\!\Rightarrow P \quad P_2 \models\!\!\Rightarrow P}{P_1 \vee P_2 \models\!\!\Rightarrow P}$$
$$P_1 \models\!\!\Rightarrow P_1 \vee P_2 \qquad P_2 \models\!\!\Rightarrow P_1 \vee P_2$$

As usual when designing an algorithm from an inference system, we are facing a choice concerning 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. Here, we present the first option. The algorithm takes the form of an inference system whose rules are to be applied in order of appearance (see Fig. 5). It can be seen as a set of rules showing how to compute a post-condition from a pre-condition and a program. The main differences with respect to the logic presented in the previous section concern the rules for `if`, `while` and assignment.

The rule for `if` avoids the need for the weakening rule. The post-condition is the disjunction of the post-conditions of the alternatives.

The rule for `while` implements an iterative algorithm akin to traditional data-flow algorithms [1]. The iteration must converge because the sequence P_i is strictly increasing:

$$P_{i-1} \models\!\!\Rightarrow P_i \quad P_i \not\models\!\!\Rightarrow P_{i-1}$$

and the set of properties under consideration is finite.

The rule for assignment statements is by far the most complex. The analyser deals with properties of the form $\bigvee P_i$(*adnf*s). The rule for each P_i in the axiomatics is

$$\frac{\{P_i\}\ v_1\ \{P_i\} \quad \{P_i\}\ v_2\ \{P_i\} \quad P_i \Rightarrow Q_i[\![v_2/v_1]\!]^{\mathcal{V}}_{P_i}}{\{P_i\}\ v_1{=}v_2\ \{Q_i\}}$$

So, given P_i, the analyser has to compute a post-condition Q_i such that $P_i \Rightarrow Q_i[\![v_2/v_1]\!]^{\mathcal{V}}_{P_i}$; this is the rôle of the function $Assign^{v_1}_{v_2}$ (cf. Fig. 6). Furthermore, P_i is of the form $A_1 \wedge \ldots \wedge A_n$ (A_k being basic properties $(x = y)$, $(x \mapsto y)$ or negations of those). The

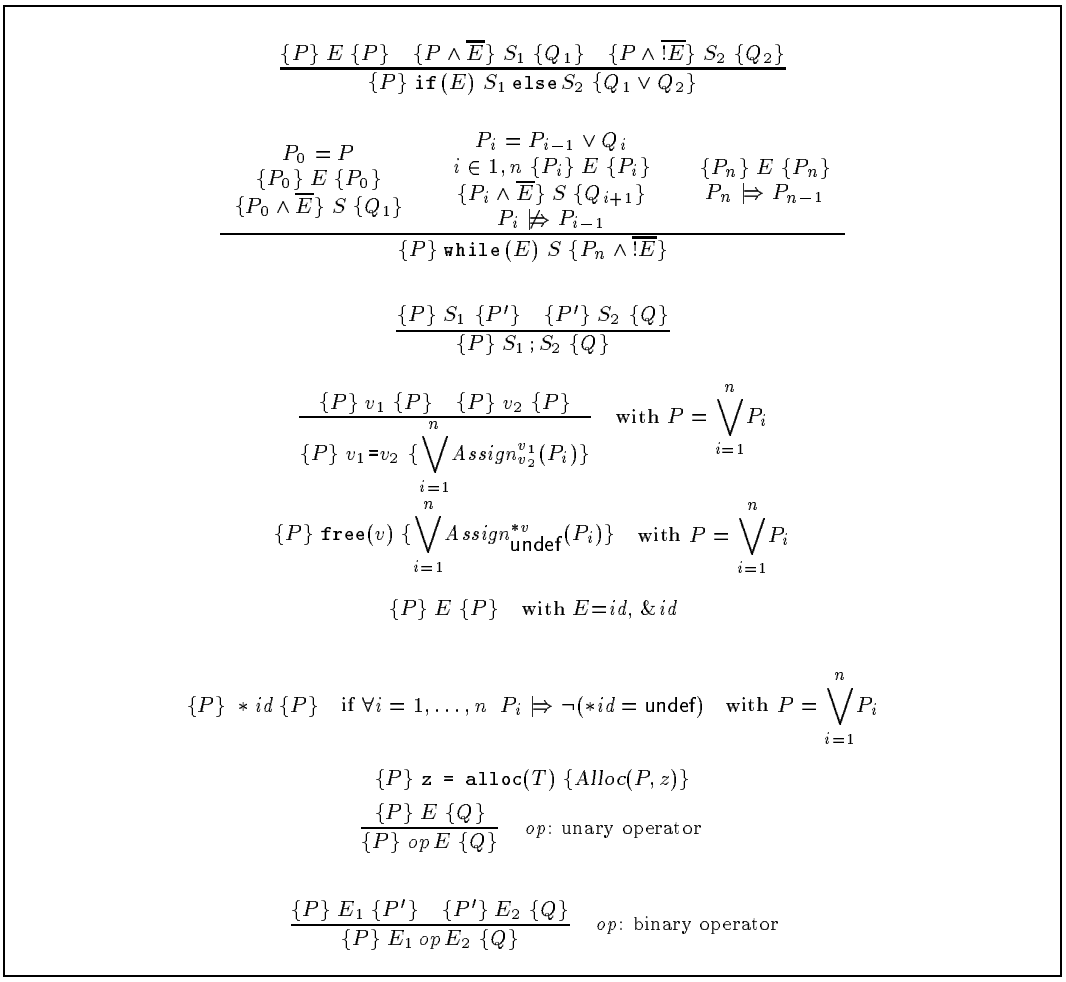

Figure 5: Rules of the analyzer

function $Produce_{v_2}^{v_1}$ (Fig. 6) determines properties B_k such that $A_k \Rightarrow B_k[\![v_2/v_1]\!]_{P_i}^{\mathcal{V}}$. By definition of substitution, we have $P_i \Rightarrow (B_1 \wedge \ldots \wedge B_n)[\![v_2/v_1]\!]_{P_i}^{\mathcal{V}}$ and the needed post-condition Q_i is therefore $B_1 \wedge \ldots \wedge B_n$.

The central task of $Produce_{v_2}^{v_1}$ is to find, for each variable x of $\mathrm{Var}_{\mathrm{Prog}}$, variables x' such that $x'[\![v_2/v_1]\!]_{P_i}^{\mathcal{V}} = x$. Two (non exclusive) cases arise:

- x is a $\mathrm{Var}_{\mathrm{Prog}}$ variable which is unaffected by the assignment (not in $Affected_{v_1}$) and $x[\![v_2/v_1]\!]_{P_i}^{\mathcal{V}} = x$.

$$
\begin{array}{lll}
Assign^{v_1}_{v_2}(P) = & \underline{\text{if}} & *v_2 \notin \text{Var}_{\text{Prog}} \\
& \underline{\text{then}} & Closure(Produce^{v_1}_{v_2}(Complete_{*v_2}(P))) \\
& \underline{\text{else}} & Closure(Produce^{v_1}_{v_2}(P))
\end{array}
$$

$$Produce^{v_1}_{v_2}(\bigvee P_i) = \bigvee Prod_{P_i}(P_i)$$

where

$$Prod_P(P_1 \wedge P_2) = Prod_P(P_1) \wedge Prod_P(P_2)$$

$$
\begin{array}{l}
Prod_P(x \text{ op } y) = \underline{\text{if}}\ \text{op} = \mapsto\ \underline{\text{and}}\ P \models x = \&v_1\ \underline{\text{then}}\ (\&v_1 \mapsto v_1) \wedge \bigwedge_{\substack{v \notin \text{Subst}_0 \\ \text{and } P \not\models \neg(v = v_2)}} \neg(\&v_1 \mapsto v) \\
\qquad \underline{\text{else if}}\ \text{op} = \neg \mapsto\ \underline{\text{and}}\ P \models x = \&v_1\ \underline{\text{then}}\ \text{True} \\
\qquad \underline{\text{else}} \bigwedge_{v,v' \in (Subst_i, Subst_j)_{i,j \in \{0,1\}}} \\
\qquad \{\ (v \text{ op } v') \quad \text{if } x = *^i v_2 \wedge y = *^j v_2 \quad \text{otherwise True} \\
\qquad\ \ (x \text{ op } v) \quad \text{if } y = *^j v_2 \wedge x \in \text{Var}_{\text{Prog}} - Affected_{v_1} \quad \text{otherwise True} \\
\qquad\ \ (v \text{ op } y) \quad \text{if } x = *^i v_2 \wedge y \in \text{Var}_{\text{Prog}} - Affected_{v_1} \quad \text{otherwise True} \\
\qquad\ \ (x \text{ op } y) \quad \text{if } x \in \text{Var}_{\text{Prog}} - Affected_{v_1} \wedge y \in \text{Var}_{\text{Prog}} - Affected_{v_1} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{otherwise True}\ \}
\end{array}
$$

$$
\begin{array}{lll}
Affected_{v_1} & = & \{x \in \text{Var}_{\text{Prog}} \mid \exists y \text{ suffix of } x, P \models y = \&v_1\} \\
Subst_i & = & \{x \in Affected_{v_1} \mid x[\![v_2/v_1]\!]^{\mathcal{V}}_P = *^i v_2\} \quad i = 0 \text{ or } i = 1
\end{array}
$$

$$\text{op} \in \{=, \neg =, \mapsto, \neg \mapsto\}$$

$$
\begin{array}{lllll}
Complete_{*x}(P) = & \underline{\text{if}} & P \models (x = \text{undef}) & \underline{\text{then}} & Closure(P \wedge (*x = \text{undef})) \\
& \underline{\text{elseif}} & P \models (x \mapsto y) \wedge (\&y = x) & \underline{\text{then}} & Closure(P \wedge (*x = y)) \\
& \underline{\text{elseif}} & P \models (x \mapsto y) & \underline{\text{then}} & Closure(P \wedge (*x = y)) \vee Insert(P, *x, y) \\
& & & \underline{\text{else}} & Add(P, *x)
\end{array}
$$

Figure 6: Functions for the assignment rule

- $x = *^i v_2$ ($i = 0$ or $i = 1$): x may be the result of the substitution of several variables. Prior to $Produce^{v_1}_{v_2}$, the analyser computes the set $Subst_0$ (resp. $Subst_1$) of Var_{Prog} variables x' such that $x'[\![v_2/v_1]\!]^{\mathcal{V}}_{P_i} = v_2$ (resp. $*v_2$). So, when $x = *^i v_2$ we have $x'[\![v_2/v_1]\!]^{\mathcal{V}}_{P_i} = x$ for all x' in $Subst_i$.

From there, basic properties can be rewritten in the form $(x'\ op\ y')[\![v_2/v_1]\!]^{\mathcal{V}}_{P_i}$. For example, let $A = x\ op\ y$ with x not in $Affected_{v_1}$ and $y = *^i v_2$ then

$$x[\![v_2/v_1]\!]^{\mathcal{V}}_{P_i} = x \text{ and } \forall v \in Subst_i \quad v[\![v_2/v_1]\!]^{\mathcal{V}}_{P_i} = y$$

so, by definition of substitution, $A \Rightarrow \bigwedge_{v \in Subst_i}(x\ op\ v)[\![v_2/v_1]\!]^{\mathcal{V}}_{P_i}$. When $op =$ " $\mapsto$ " or "$\neg \mapsto$" we also have to check that $P_i \not\models \neg(x = \&v_1)$ to be able to apply the definition of substitution (see Fig. 4). The three other cases in the definition of $Produce^{v_1}_{v_2}(x\ op\ y)$ are similar. Note that basic properties involving a variable affected by the assignment and different from $*^i v_2$ are removed (i.e. *True* is produced).

It can be shown that $Produce^{v_1}_{v_2}$ yields a post-condition in *adnf* provided the pre-condition is in *adnf* and $*v_2$ is in Var_{Prog}. Otherwise, $*v_2$ must first be added to the pre-condition

$Closure(P)$ is defined as the normal form of the $\succ$ relation defined as follows:

$$P' \wedge (a \text{ op } b) \wedge (a = a') \wedge (b = b') \succ P \wedge (a' \text{ op } b')$$

$$Insert(P, *x, y) = \mathrm{NF}_1^{*x}(Closure(Replace(Mk\text{-}node(P, *x), x \mapsto y, *x \mapsto y)))$$

with:

$$Mk\text{-}node(P, *x) = P \wedge (x \mapsto *x) \wedge (*x = *x) \wedge \bigwedge_{z \in P} (\neg(*x = z))$$

$$\begin{aligned} &Replace(P \wedge (v \mapsto v'), (v \mapsto v'), (w \mapsto w')) \\ &\quad = \underline{\text{if}}\ P = P' \wedge (v_1 \mapsto v_1')\ \underline{\text{and}}\ P' \not\Rightarrow (v = v_1) \wedge (v' = v_1') \\ &\qquad \underline{\text{then}}\ Replace(P' \wedge (v_1 \mapsto v_1'), (v_1 \mapsto v_1'), (w \mapsto w')) \wedge \neg(v \mapsto v') \\ &\qquad \underline{\text{else}}\ P \wedge (w \mapsto w') \wedge \neg(v \mapsto v') \end{aligned}$$

NF_1^{*x} normal form of the $\succ_1^{*x}$ relation defined as follows:

$$P = P' \wedge (a \mapsto b) \wedge \neg(b = c) \succ_1^{*x} P \wedge \neg(a \mapsto c)$$

$$P = P' \wedge (*x \mapsto a) \wedge (b \mapsto a) \wedge \neg(b = *x) \wedge \neg(b = x) \succ_1^{*x} P \vee Replace(P, b \mapsto a, b \mapsto *x)$$

$$P = P' \wedge (*x \mapsto a) \wedge \neg(b \mapsto a) \wedge \neg(b = x) \succ_1^{*x} P \wedge \neg(b \mapsto *x)$$

and $\mathrm{NF}_1^{*x}(P_1 \vee P_2) = \mathrm{NF}_1^{*x}(P_1) \vee \mathrm{NF}_1^{*x}(P_2)$

$$Add(P, *x) = \mathrm{NF}_2^{*x}(End(Closure(Mk\text{-}node(P, *x)), *x))$$

with:

$$End(P, *x) = P \wedge \bigwedge_{z \in P} (\neg(*x \mapsto z))$$

NF_2^{*x} normal form of the $\succ_2^{*x}$ relation defined as follows:

$$P \succ_2^{*x} (P \wedge \neg(b \mapsto *x)) \vee (P \wedge (b \mapsto *x)) \text{ if } b \in \{v \in \mathrm{Var}(P) \mid \nexists w\, P \not\Rightarrow (v \mapsto w)\}$$

$$P = P' \wedge (a \mapsto b) \wedge \neg(b = *x) \succ_2^{*x} P \wedge \neg(a \mapsto *x)$$

and $\mathrm{NF}_2^{*x}(P_1 \vee P_2) = \mathrm{NF}_2^{*x}(P_1) \vee \mathrm{NF}_2^{*x}(P_2)$

Figure 7: Normalisation functions for the assignment rule

using the function *Complete*. The consequences of our restriction to the fixed set of variables $\mathrm{Var}_{\mathrm{Prog}}$ are to be found in this function. $Complete_{*v_2}$ relies on connectivity relations (such as $v_2 \mapsto x$) but nevertheless has to introduce disjunctions to deal with the lack of information on $*v_2$. The functions in Fig. 7 are used to normalise properties in *adnf*s with respect to the extended set of variables.

Let us consider the following pre-condition:
$P = (y \mapsto *y) \wedge (x \mapsto z) \wedge \neg(x = y) \wedge \neg(x = *y) \wedge \neg(z = y) \wedge \neg(z = *y) \wedge \ldots$
and the assignment $y = x$. The post-condition is computed by $Assign_x^y(P)$. From the definitions in Fig. 6, we get:

$$\begin{aligned} Affected_y &= \{y, *y\} \text{ (set of variables with a suffix alias of } y) \\ Subst_0 &= \{y\} \quad \text{(set of variables equated to } x \text{ by substitution)} \end{aligned}$$

$$
\begin{array}{l}
\mathit{Alloc}(P,z) = \mathit{Closure}(\mathit{Clear}(\ P,z) \ \wedge (\&z \mapsto z \mapsto *z \mapsto \mathsf{undef}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \bigwedge_{v \in P} (\neg(v \mapsto \&z)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \bigwedge_{v \in P - \{\&z\}} (\neg(\&z = v) \wedge \neg(v \mapsto z)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \bigwedge_{v \in P - \{z\}} (\neg(\&z \mapsto v) \wedge \neg(z = v) \wedge \neg(v \mapsto *z)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \bigwedge_{v \in P - \{*z\}} (\neg(*z = v) \wedge \neg(z \mapsto v)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \bigwedge_{v \in P} (\neg(*z \mapsto v))\ ) \\
\qquad\qquad\qquad\qquad\qquad\qquad P \models \neg(v = \mathsf{undef}) \\
\mathit{Clear}(\bigvee P_i, z) = \bigvee(P_i/z) \\
P_i/z = \bigwedge\{x \ \mathrm{op}\ y \mid x, y \notin \{\&z, z, *z\}\} \wedge \bigwedge\{v = v \mid v \in \{\&z, z, *z\}\}
\end{array}
$$

Figure 8: Functions for the "alloc" rule

$$\mathit{Subst}_1 \quad = \quad \{*y\} \quad \text{(set of variables equated to } *x \text{ by substitution)}$$

Let us assume that the variable $*x$ is not in $\mathrm{Var}_{\mathrm{Prog}}$; $\mathit{Produce}_x^y$ cannot build any property on $*y$ from P (since $*y[\![x/y]\!]_P^{\mathcal{V}} = *x$). The variable $*x$ must be added to P using $\mathit{Complete}_{*x}(P)$. Since $P \models (x \mapsto z)$, we have $\mathit{Complete}_{*x}(P) = \mathit{Closure}(P \wedge (*x = z)) \vee \mathit{Insert}(P, *x, z)$. The disjunction is necessary because the length of the path between x and z is unknown, so $*x$ may either be equal to z or stand on the path between x and z. $\mathit{Closure}(P \wedge (*x = z))$ adds all missing properties of $*x$ (identical to properties of z) and yields an *adnf*. $\mathit{Insert}(P, *x, z)$ adds the property $(*x \mapsto z)$. It is more involved because other variables pointing to z may interfere. If P implies $(v \mapsto z)$, sharing may occur between paths from v to z and x to z. In particular, if v and x point to cells having the same value (i.e. $*x = *v$) then $(v \mapsto z)$ must be split into $(v \mapsto *x) \wedge (*x \mapsto z)$. This is done by the second rule of NF_1^{*x} in Fig. 7.
After this step, $\mathit{Produce}_x^y$ evaluates the post-condition in a natural way, and we get:

$$
\begin{array}{rl}
\mathit{Assign}_x^y(P) = & [(x = y) \wedge (*y = z) \wedge (x \mapsto z) \wedge (y \mapsto *y) \wedge (y \mapsto z) \wedge \ldots] \\
& \vee[(x = y) \wedge (x \mapsto *y) \wedge (*y \mapsto z) \wedge (y \mapsto *y) \wedge \neg(y \mapsto z) \ldots]
\end{array}
$$

The following theorems establish the correctness of the analyser.

THEOREM **3.2**

If P is in adnf and $\{P\}\ S\ \{Q\}$ can be proven using the inference system of Fig. 5 then Q is in adnf.

THEOREM **3.3**

If $\{P\}\ S\ \{Q\}$ can be proven using the inference system of Fig. 5 then $\{P\}\ S\ \{Q\}$ can be proven using the inference system of Fig. 3.

Theorem 3.2 shows that the atomic disjunctive normal form representation is invariant which ensure that the analyser will be able to produce a result for all input programs. The proof of theorem 3.3 is made by induction on the structure of proof of the premise. The difficult part is the rule for assignment which follows from the lemma:

LEMMA **3.4** $\quad P \Rightarrow Assign_{v_2}^{v_1}(P)[\![v_2/v_1]\!]_P^{\mathcal{V}}$

Proofs can be found in Appendix 3.

4 Checking a program manipulating circular data structures

In this section, we illustrate our analysis with the Josephus program (Figure 9) borrowed from [24], page 22. The program first builds a circular list, then proceeds through the list, counting through $m-1$ items and deleting the next, until only one is left (which points to itself). Figure 10 shows some steps of the analysis. The post-conditions computed by the algorithm are displayed as graphs to the right of the corresponding statement. Two variables v_1 and v_2 are in the same node in the graph G_i if $P_i \models\!\!\Rightarrow v_1 = v_2$. They are in different nodes if $P_i \models\!\!\Rightarrow \neg(v_1 = v_2)$ (remember that one of the two implications must hold). As expected, arcs represent the $\mapsto$ relations. Iteration steps are described up to convergence. The numbers of steps necessary for the first `for` loop, the second `for` loop and the `while` loop are respectively 2, 3 and 2. The post-conditions of `*t.next = x` after the first `for` loop shows that t points to a circular list (which can be of length 1, 2 or more, each case corresponding to one of the three properties of the disjunction) and there is no other aliasing in the state. The post-conditions of `t = *t.next` in the second `for` loop show that this circular structure is an invariant of the loop. We stress the fact that the use of the condition `t!=*t.next` (through $\overline{\texttt{t!=*t.next}} = \neg(t = *t)$) in the treatment of the `while` loop is crucial to prove that the `free` statement cannot create a pending pointer. Its effect is to filter out the first property of the disjunction in the pre-condition of the `while` loop. Also the post-condition of the whole program implies that t must point to itself. This program does not follow faithfully the syntax defined in Figure 11 because a double dereference appears in the `while` loop, just before the `free` instruction (`*t.next = *(*t.next).next;`). A straightforward transformation can be applied to remove multiple dereferences and get programs which are fully compliant to the syntax used in this paper.

```
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 9: Josephus program

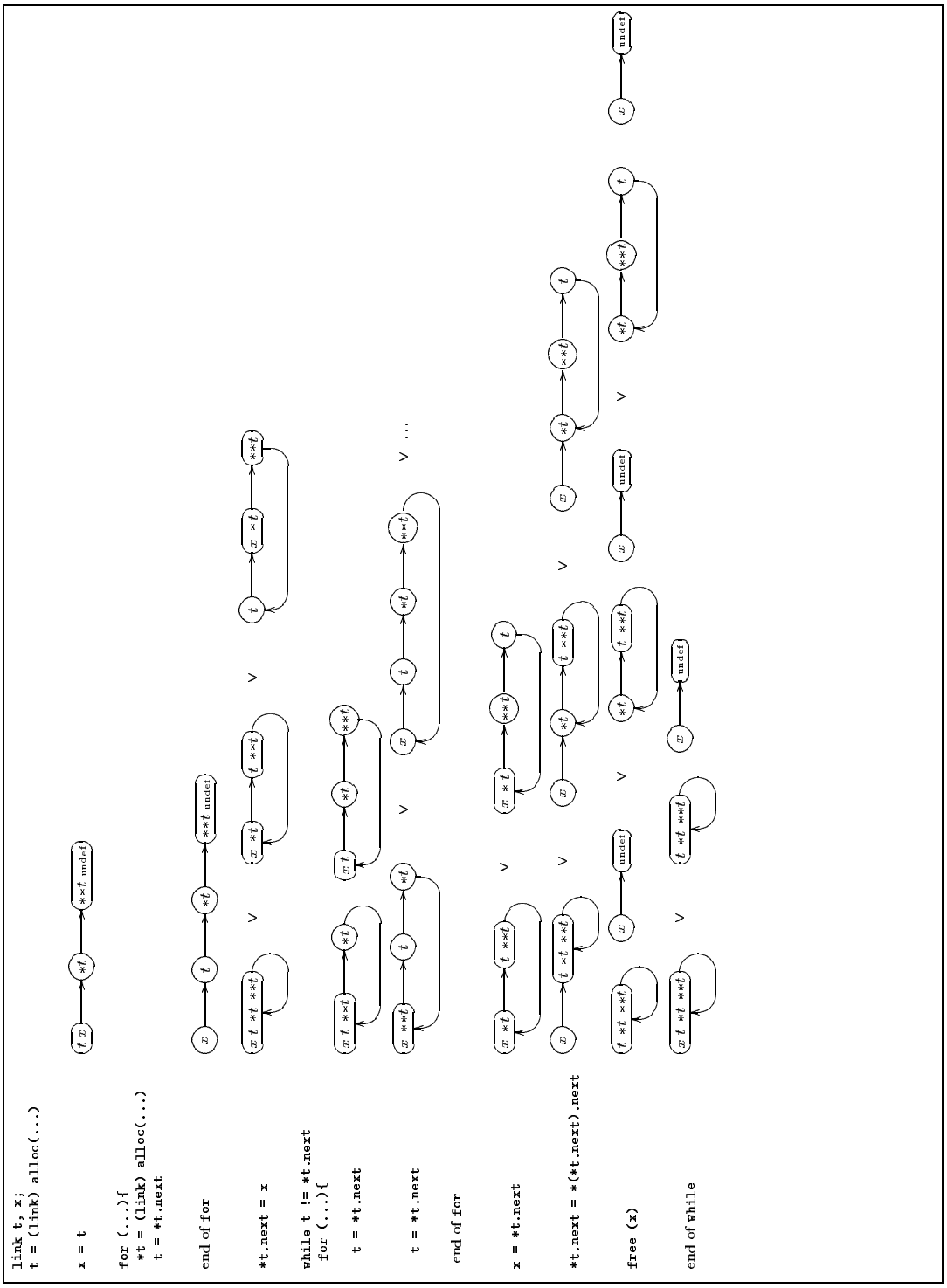

Figure 10: Analysis of Josephus program

5 Extensions

The analysis is readily extended to deal with a large subset of C. We sketch in this section how declarations, procedures and goto's can be accommodated. We suppose that declarations are made only at the beginning of a function block and that static and dynamic scoping are equivalent. This simplifies the axiomatisation (e.g. interaction between goto's and declarations). Both restrictions can be enforced by simple program transformations (e.g. renaming of variables).

Declarations of pointer variables

Let us consider a block with a pointer variable declaration of the form { *(type)*`* x;`E}. The Hoare-like rule for this declaration is

$$\frac{\{P[y/x] \wedge (x = \mathsf{undef}) \bigwedge_{\forall z \in Var(P[y/x])} \neg(\&x = z)\}\ E\ \{Q[y/x]\}}{\{P\}\ (type)\texttt{* x;}E\ \{Q\}} \quad y \text{ being a fresh variable}$$

The implementation of this rule in the analyser is straightforward. The declared variables are renamed in the pre-condition using fresh variables and properties of the newly allocated local variables are added. On exit, all the properties on declared variables (x) are removed and previously renamed variables (y) take back their original name (x).

We can get the general rule for declarations by extending the above rule to several declarations and by considering for each declared identifier all its prefixes occurring in the body. Also, since pointers can appear in compound types (e.g. structures), the rule must extract all the variables (access chains) denoting pointers from declarations.

Procedures

We focus here on the specific problems introduced by procedures and we restrict our attention to the case of one recursive parameterless procedure. The rules presented below can be straightforwardly generalised to mutually recursive procedures. It is also relatively simple to transform any C function into a parameterless procedure using local variables declarations and global variables to pass the parameters. The general rule for procedures can then be deduced from the rule for declarations and the rule for parameterless procedures.

The Hoare logic rule for a recursive procedure `f` with body S is

$$\frac{\{P\}\ \texttt{f()}\ \{Q\} \vdash \{P\}\ S\ \{Q\}}{\{P\}\ \texttt{f()}\ \{Q\}}$$

which means that the body is analysed with $\{P\}$`f()`$\{Q\}$ as an induction hypothesis on recursive calls `f()`. This rule is refined to compute iteratively the pre-condition as follows.

A procedure call with pre-condition P_0 and no induction hypothesis is treated by analysing the body with the induction hypothesis $\{P_0\}$`f()`{False}.

$$\frac{\{P_0\}\ \texttt{f()}\ \{\text{False}\} \vdash \{P_0\}\ S\ \{Q_n\}}{\{P_0\}\ \texttt{f()}\ \{Q_n\}}$$

Two cases arise with recursive calls. If the pre-condition of the induction hypothesis implies the pre-condition of the call then the iteration ends.

$$P_n \models\!\Rightarrow P \quad \{P_n\}\ \texttt{f()}\ \{Q_n\} \vdash \{P\}\ S\ \{Q_n\}$$

Otherwise, the body is analysed again with a new (greater) pre-condition.

$$P_i \not\models\!\Rightarrow P \quad \frac{\{P_i \vee P\}\ \texttt{f()}\ \{Q_i\} \vdash \{P_i \vee P\}\ S\ \{Q_n\}}{\{P_i\}\ \texttt{f()}\ \{Q_i\} \vdash \{P\}\ \texttt{f()}\ \{Q_n\}}$$

The strictly increasing chain of pre-conditions and the bounded number of possible properties ensure termination.

As described the algorithm reanalyses the body of the procedure for each call. An alternative could be to analyse the body only once to provide summary of the effects of the procedure for all possible pre-condition. In our approach, this could be done using a generic (variable) pre-condition. This summary would then be applied to each different call context. However, the effects of a procedure depend too much on the actual pre-condition (notably the alias relation) to expect concise and useful summaries. Actually, it is likely that this approach would be as costly as reanalyses. This problem has already been pointed out in the context of alias analysis ([26]).

Intuitively, only a subset of the properties are relevant for the analysis. Our approach is to determine for each procedure the relevant part of the pre-condition. The analysis remains context sensitive but is done only for context calls with different relevant parts. An example of invariant property is $x = y$ where no suffix of x or y is an alias of any variable assigned in the body of the procedure. Let P_i be the set of such properties in P and $P = P_r \wedge P_i$. If we prove $\{P_r\}\ \texttt{f()}\ \{Q_r\}$ we can deduce $\{P\}\ \texttt{f()}\ \{Q_r \wedge P_i\}$. Actually, much more factorisation is possible. As noted in [20], a procedure `f` has the same effect on all alias pairs which contain variable x (in the scope of `f`) and any non-visible variable. This property can be formally justified and exploited in our framework. We expect that relevant pre-conditions concern only few variables and will be shared by many context calls.

In order to extend the approach to procedure calls through pointers, we must determine for each call a (super)set of possible callees. A special analysis can be used for this purpose but another solution in our framework is to allow functions addresses in Var_{Prog}. Our analyser could naturally infer the needed information as connectivity relations between function pointers and function addresses.

Control Transfers.

Apart from `goto`, three instructions alter the flow of control in C: `break`, `continue` and `return`. Each of them can be replaced by a forward `goto`. The `break` statement (resp. `continue`) is a `goto` to a label just after (resp. before) the smallest enclosing loop or switch.

The return statement can be suppressed by adding a new parameter to each function and replacing each call `f(x1,...,xn)` by `f(&r,x1,...,xn), r` (`r` being a fresh local variable) and each return statement `return (e)` by `r=e; goto end_f;`. We therefore describe the extension of our approach to forward gotos, the most common form of control transfer in C programs.

Several proposals have been made to extend Hoare's logic to goto statements. We choose here a notation close to Arbib and Alagić's [4]. The rules add to the normal post-condition an environment where every label occurring in the statement is associated with a condition. For example

$$\{P\}\ E\ \{Q \mid l_1 : R_1, ..., l_n : R_n\}$$

expresses the fact that, when P holds on entry, then Q holds on normal exit while R_i holds on any exit from E via `goto` l_i. Goto and labeled statements are treated by the two following rules

$$\{P\}\ \texttt{goto}\ l_i\ \{\text{False} \mid l_i : P\}$$

$$\frac{\{P\}\ S_1\ \{P_i \mid lenv_1\} \quad \{P_i\}\ S_2\ \{Q \mid lenv_2\}}{\{P\}\ S_1; \texttt{li:}S_2\ \{Q \mid (lenv_1 - l_i : P_i) \cup lenv_2\}}$$
where $lenv_1 = l_1 : R_1, ..., l_i : P_i, ..., l_p : R_p$

Note that the union of the two label environments enforces that the same label occurring in both environments is associated with the same property. The other rules must be changed to take into account the label environment. For example, the rule for ";" becomes

$$\frac{\{P\}\ S_1\ \{P_1 \mid l_1 : R_1, ..., l_n : R_n\} \quad \{P_1\}\ S_2\ \{Q \mid l_1 : R_1, ..., l_n : R_n\}}{\{P\}\ S_1; S_2\ \{Q \mid l_1 : R_1, ..., l_n : R_n\}}$$
where $S_1; S_2$ do not contain labels $l_1, ..., l_n$.

The refinement of these rules is along the same lines as the rules for `if` or `switch` rules. The disjunction of all the pre-conditions of statements `goto` l_i is used as the pre-condition of the statement l_i`:E`.

The framework is powerful enough to axiomatise the general goto's of C. As can be expected, in the presence of backward gotos, pre-conditions will be found by iteration. However, a reasonably efficient analysis of general goto's is more complicated and will not be described here.

6 Conclusion

The work described in this paper stands at the crossroad of three main trends of activities:

- the design of semantic based debugging tools,

- alias analysis,
- the axiomatisation of languages with explicit pointer manipulation.

We sketch related work in each of these areas in turn.

- There are relatively few papers about the design of program analysers to help in the program development process. Most related contributions [6, 13, 15, 25] and tools [19] can provide information about uninitialised variables but are unable to track illegal accesses in recursive data structures. Other techniques like [14, 18] perform different kinds of analyses (like aspects, program slicing) which are complementary to the work described here.
- There is an extensive body of literature on alias analysis but most of the contributions are concerned with may-alias analysis and are targeted towards compiler optimisations [11, 12]. The alias pairs (x, y) of [11] correspond to $\&x = \&y$ here and the x points-to y relationship of [12] is equivalent to $x = \&y$. One of the most precise published alias analysis is the framework described in [11]. Our analysis is not directly comparable to this one in terms of precision: on one hand, the symbolic access paths used in [11] provide a much more accurate may-alias information (because numerical coefficients are used to record precise positions in a structure); on the other hand, our properties include both may-alias and must-alias information which allows us to gain accuracy in certain situations (the significance of must-alias properties to get more accurate may-alias properties is stressed in [2]). This extra level of precision is required to the analysis of correctness-related properties (for instance, the example treated in Section 4 could not be analysed successfully without a form of must-alias information).
- Axiomatisation of pointer and alias relations has been studied for Pascal (see e.g. [7, 8, 22]). Most contributions in this area focus on generality and completeness issues and do not consider automatisation. An exception is the work by Luckham and Suzuki [21] which presents an axiom-based verifier for Pascal programs. The language of properties encompasses ours but is too rich to make the analysis fully automatic. The verifier (actually a theorem prover) depends heavily on user-supplied properties such as loop invariants.

The work whose spirit is the closest to our approach is the analysis framework presented in [23]. Environments are described as sets of assertions specified as Horn clauses. They define optimal analyses which exploit all the information available. Our $=$ relation is close to their universal static predicate $eq_\forall$ but they do not have a counterpart for our $\mapsto$ relation (because they do not attempt to track pointer equality in recursively defined structures, which is the main issue of this paper) and they do not consider disjunctive properties. Also they do not study the link of the analysis with an operational semantics of the language (or, to be more precise, the semantics of their language is expressed logically in terms of predicate transformers).

The approach followed in this paper does not stand at the same level as usual presentations of static analyses. Our starting point, the axiomatics of Fig. 3, is a specification of the property under consideration which is not biased towards a specific analysis technique. Programs are associated with pre/post-conditions relations but no transformation function is provided to compute one from the other; in fact, even the direction in which proofs are to be carried out is left unspecified. The main goal of the transformation leading to the system of Fig. 5 is precisely to introduce a direction for the analysis and to derive transfer functions from the pre/post-conditions relations[2]. We have presented a forward analysis here but we could as well have chosen the derivation of a backward analyser. The analyser of Fig. 5 itself can be rephrased as an abstract interpretation of the operational semantics. The abstract domain is the disjunctive completion of a lattice of matrices (associating each pair (v_1, v_2) with truth values of the basic relations $=$ and $\mapsto$). This domain has some similarities with the path matrices used in [17] for the analysis of a restricted form of regular acyclic structures. The abstraction and concretisation functions follow directly from the correspondence relation of Fig. 1. Instead of a correctness proof of the analyser with respect to the axiomatics as suggested here, the soundness of the analysis would then be shown as a consequence of the soundness of the abstract interpretation of the basic rules with respect to the operational semantics (see [10] for an illustration of this approach). Again, the most difficult rule is the assignment. It is not clear whether the overall effort would be less important but the formulation in terms of abstract interpretation would make it easier to show the optimality of the analyser (in terms of precision) [9]. Also, the approximation techniques studied in this framework can be applied to get more efficient analysers. So, the two approaches are complementary: we have focussed in this paper on the derivation of an analysis from the axiomatisation of a property, emphasizing a clear separation between logical and algorithmic concerns. Hoare logic is an ideal formalism at this level because it makes it possible to leave unspecified all the details which are not logically relevant. On the other hand, abstract interpretation is a convenient framework for describing analyses themselves as well as studying approximation and algorithmic issues.

The algorithm presented in section 3 is only a first step towards the design of an effective analyser. Its worst case complexity is clearly exponential in terms of the number of variables in the program. The main source of inefficiency is the use of disjunctions to represent the lack of information incurred when dereferencing a variable v when $*v \notin \mathrm{Var}_{\mathrm{Prog}}$. We are currently investigating several complementary optimisations to improve the situation:

- Approximating properties to reduce the size of the abstract domain and the complexity of the primitive operations on properties. One solution leads to a representation of properties as matrices of a three- value domain (instead of sets of matrices of a boolean domain as suggested in this paper).
- Computing only the necessary part of each property using a form of *lazy type inference* [16].

[2] In fact, the transformation also performs an approximation, mapping the set of variables into a finite subset, but this issue could have been dealt with separately.

- Using (standard) types to filter properties which cannot be true. Exploiting this extra information usually reduces dramatically the size of the properties manipulated by the algorithm.

We are also studying the use of the pointer analysis described here to enhance the information flow analysis proposed in [5]. Other applications of this analysis include the detection of unsafe programming styles (which rely on specific implementation choices like the order of evaluation of subexpressions) or memory leaks. A different perspective of this work could be its use as a specialised interactive theorem prover for a restricted form of Hoare logic.

References

[1] A. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley publishing company, 1988.

[2] R. Altucher and W. Landi, *An extended form of must-alias analysis for dynamic allocation*, in 22^{nd} Annual ACM Symp. on Principles of Programming Languages POPL'95, Jan. 1995, pp.74-85.

[3] L. Andersen, *Program analysis and specialisation for the C programming language*, Ph.D Thesis, DIKU, University of Copenhagen, May 1994.

[4] M.A. Arbib, S. Alagić, Proof rules for gotos, *Acta Informatica 11,2*, pp.139-148,1979.

[5] J.-P. Banâtre, C. Bryce, D. Le Métayer, *Compile-time detection of information flow in sequential programs*, proc. European Symposium on Research in Computer Security, Springer Verlag, LNCS 875, pp. 55-74.

[6] J.F. Bergeretti and B. Carré, *Information-flow and data-flow analysis of while-programs*, in ACM Transactions on Programming Languages and Systems,Vol. 7, No. 1, Jan. 85, pp. 37-61.

[7] A. Bijlsma, *Calculating with pointers*, in Science of Computer Programming 12 (1989) 191-205, North-Holland.

[8] R. Cartwright and D. Oppen, *The logic of aliasing*, in Acta Informatica 15, 365-384, 1981 ACM TOPLAS, Vol. 7, 1985, pp. 299-310.

[9] P. Cousot and R. Cousot, *Systematic design of program analysis frameworks*, in 6^{th} Annual ACM Symp. on Principles of Programming Languages POPL'79, Jan. 79, pp. 269-282.

[10] A. Deutsch, *A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations*, in Proc. of the IEEE 1992 Conf. on Computer Languages, Apr. 92, pp. 2-13.

[11] A. Deutsch, *Interprocedural may-alias analysis for pointers: Beyond k-limiting*, in SIGPLAN'94 Conf. on Programming Language Design and Implementation PLDI'94, Jun. 1994, pp. 230-241.

[12] M. Emami, R. Ghiya and L. Hendren, *Context-sensitive interprocedural points-to analysis in the presence of function pointers*, in SIGPLAN'94 Conf. on Programming Language Design and Implementation PLDI'94, Jun. 1994, pp. 242-256.

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

[14] J. Field, G. Ramalingam and F. Tip, *Parametric program slicing*, in 22^{th} Annual ACM Symp. on Principles of Programming Languages POPL'95, Jan. 95, pp. 379-392.

[15] L. Fosdick and L. Osterweil, *Data flow analysis in software reliability*, ACM Computing surveys, 8(3), Sept. 1976.

[16] 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.

[17] L. Hendren and A. Nicolau, *Parallelizing programs with recursive data structures*, in IEEE Transactions on Parallel and Distributed Systems, Jan. 90, Vol. 1(1), pp. 35-47.

[18] D. Jackson, *Aspect: an economical bug-detector*, in Proceedings of 13^{th} International Conference on Software Engineering, May 1994, pp. 13-22.

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

[20] W. Landi and B. Ryder, *A safe approximate algorithm for interprocedural pointer aliasing*, in Proceedings of the ACM SIGPLAN'92 Conf. on Programming Language Design and Implementation PLDI'92, Jun. 1992, pp. 235-248.

[21] 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.

[22] J. Morris, *A general axiom of assignment* and *Assignment and linked data structures*, in Theoretical Foundations of Programming Methodology, M. Broy and G. Schmidt (eds), pp. 25-41, 1982.

[23] S. Sagiv, N. Francez, M. Rodeh and R. Wilhelm, *A logic-based approach to data flow analysis problems*, in Programming Language Implementation and Logic Programming PLILP'90, LNCS 456, pp. 277-292, 1990.

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

[25] 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.

[26] R.P. Wilson and M.S. Lam, *Efficient context-sensitive pointer analysis for C programs* in Proceedings of the ACM SIGPLAN'95 Conf. on Programming Language Design and Implementation PLDI'95, Jun. 1995, pp. 1-12.

Appendix 1 Abstract syntax and dynamic semantics of a subset of C

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
exp	::=	*id*	Variable (*id* $\in$ Id)
	|	$*id$	Pointer dereference
	|	$\&id$	Address operator
	|	**alloc** (type)	Runtime allocation
	|	*op* exp	Unary operator
	|	exp *op* exp	Binary operator
lexp	::=	*id*	
	|	$*id$	

Figure 11: Abstract syntax of a subset of C

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

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

Figure 12: Dynamic semantics for statements and expressions

Appendix 2 Proof of Theorem 2.2

Before embarking in the proof of Theorem 2.2, we first show some usefull lemmas.

LEMMA **6.1**

$$\mathcal{C}_{\mathcal{V}}(P,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}) \wedge (P \Rightarrow Q) \Rightarrow \mathcal{C}_{\mathcal{V}}(Q,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}).$$

The proof is done by considering each rule in Fig.2. Let us consider the rule

$$v_1 \mapsto v_2 \Rightarrow (v_2 = *v_1) \vee (*v_1 \mapsto v_2).$$

We have to prove

$$\mathcal{C}_{\mathcal{V}}(v_1 \mapsto v_2,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}) \Rightarrow \mathcal{C}_{\mathcal{V}}((v_2 = *v_1) \vee (*v_1 \mapsto v_2),\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}})$$

From definition of $\mathcal{C}_{\mathcal{V}}$ in Fig.1, we get

$$\begin{aligned}
\exists \alpha_1 \ldots \alpha_k,\ \alpha_1 &= \mathit{Val}(v_1,\, \mathcal{E},\, S'_{\mathcal{D}'}) && (1)\\
\alpha_k &= \mathit{Val}(v_2,\, \mathcal{E},\, S'_{\mathcal{D}'}) && (2)\\
\forall 1 \leq i < k &\quad S'_{\mathcal{D}'}(\alpha_i) = \alpha_{i+1} && (3)\\
\forall 1 < i < k &\quad \forall x \in \mathcal{V}, \alpha_i \neq \mathit{Val}(x,\, \mathcal{E},\, S'_{\mathcal{D}'}) && (4)
\end{aligned}$$

Two cases arise: $k = 2$ or $k \geq 2$. If $k = 2$,

$$(1) \wedge (2) \wedge (3) \Rightarrow \mathcal{S}_{\mathcal{D}}(\,\mathit{Val}(v_1,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}})) = \mathit{Val}(v_2,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}})$$

From definition of *Val* in Fig.1,

$$\mathcal{S}_{\mathcal{D}}(\,\mathit{Val}(v_1,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}})) = \mathit{Val}(*v_1,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}})$$

which implies

$$\mathit{Val}(v_2,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}) = \mathit{Val}(*v_1,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}})$$

and from definition of $\mathcal{C}_{\mathcal{V}}$,

$$\mathcal{C}_{\mathcal{V}}(v_2 = *v_1,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}).$$

If $k \geq 2$, we want to prove

$$\begin{aligned}
\exists \alpha'_1 \ldots \alpha'_{k'},\ \alpha'_1 &= \mathit{Val}(*v_1,\, \mathcal{E},\, S'_{\mathcal{D}'})\\
\alpha'_{k'} &= \mathit{Val}(v_2,\, \mathcal{E},\, S'_{\mathcal{D}'})\\
\forall 1 \leq i < k' &\quad S'_{\mathcal{D}'}(\alpha'_i) = \alpha'_{i+1}\\
\forall 1 < i < k' &\quad \forall x \in \mathcal{V}, \alpha'_i \neq \mathit{Val}(x,\, \mathcal{E},\, S'_{\mathcal{D}'})
\end{aligned}$$

we just have to take $\alpha'_i = \alpha_{i+1}$ and $k' = k - 1$.
Other cases are proved in the same way.

LEMMA **6.2**

$$\mathcal{E} \vdash_{exp} <E, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <b, \mathcal{S}'_{\mathcal{D}'}>,\ b \neq 0 \Rightarrow\ \mathcal{C}_{\mathcal{V}}(\overline{E},\ \mathcal{E},\ S'_{\mathcal{D}'})$$
$$\mathcal{E} \vdash_{exp} <E, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <b, \mathcal{S}'_{\mathcal{D}'}>,\ b = 0 \Rightarrow\ \mathcal{C}_{\mathcal{V}}(\overline{!E},\ \mathcal{E},\ S'_{\mathcal{D}'})$$

The proof is done by induction on the structure of E and the definition of "$\overline{\ }$". Only boolean expression involving pointer variables are to be considered (for others expressions E, we have $\overline{E}$ =True).
Let us consider one case:
Let E be id_1==id_2 and let us assume $\mathcal{E} \vdash_{exp} <id_1\texttt{==}id_2, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <b, \mathcal{S}'_{\mathcal{D}'}>,\ b \neq 0$.
We want to show

$$\mathcal{C}_{\mathcal{V}}(\overline{id_1\texttt{==}id_2},\ \mathcal{E},\ S'_{\mathcal{D}'}).$$

From the definition 2.1,

$$\overline{id_1\texttt{==}id_2} = (id_1 = id_2)$$

so it is equivalent to show

$$\mathcal{C}_{\mathcal{V}}(id_1 = id_2,\ \mathcal{E},\ S'_{\mathcal{D}'}).$$

The semantic function $\mathcal{O}$ is not specified in our system. It gives sense to unary and binary operators like ==. Here, the value of b is determined in a natural way by this semantic function: $\mathcal{O}(\texttt{op})(id_1,\ id_2)$ returns a value different from 0 if and only if the values of id_1 and id_2 are equal in the state $(\mathcal{E}, \mathcal{S}'_{\mathcal{D}'})$.
So, we get, with $b \neq 0$

$$\mathcal{S}'_{\mathcal{D}'}(\mathcal{E}(id_1)) = \mathcal{S}'_{\mathcal{D}'}(\mathcal{E}(id_2)).$$

But, from the definition of *Val* in Fig.1

$$\mathcal{S}'_{\mathcal{D}'}(\mathcal{E}(id_1)) = Val(id_1,\ \mathcal{E},\ S'_{D'})$$

and

$$\mathcal{S}'_{\mathcal{D}'}(\mathcal{E}(id_2)) = Val(id_2,\ \mathcal{E},\ S'_{D'})$$

which imply

$$Val(id_1,\ \mathcal{E},\ S'_{D'}) = Val(id_2,\ \mathcal{E},\ S'_{D'}).$$

So, from the definition of $\mathcal{C}_{\mathcal{V}}$ in Fig.1,

$$\mathcal{C}_{\mathcal{V}}(id_1 = id_2,\ \mathcal{E},\ S'_{\mathcal{D}'}).$$

Other cases are proved in the same way.

DEFINITION **6.3**

$$Adr(v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = Val(\&v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})$$

LEMMA **6.4**

$$\mathcal{E} \vdash_{exp} <v, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <val, \mathcal{S}'_{\mathcal{D}'}> \Rightarrow \mathit{Val}(v,\ \mathcal{E},\ S'_{\mathcal{D}'}) = val$$
$$\mathcal{E} \vdash_{lexp} <v, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <a, \mathcal{S}'_{\mathcal{D}'}> \Rightarrow \mathit{Adr}(v,\ \mathcal{E},\ S'_{\mathcal{D}'}) = a$$

The proof is done by induction on the structure of v. Let us consider the case $v = id$.
We have

$$\mathcal{E} \vdash_{exp} <id, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <\text{val}, \mathcal{S}'_{\mathcal{D}'}>$$

and we want to prove

$$\mathit{Val}(id,\ \mathcal{E},\ S'_{\mathcal{D}'}) = \text{val}.$$

From [var] rule in Fig.12, we have

$$\mathcal{E} \vdash_{exp} <id, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <\mathcal{S}_{\mathcal{D}}(\mathcal{E}(id)), \mathcal{S}_{\mathcal{D}}> \quad \text{if } \mathcal{E}(id) \in \mathcal{D} \quad .$$

So, here we have

$$\text{val} = \mathcal{S}_{\mathcal{D}}(\mathcal{E}(id))$$

and

$$\mathcal{S}_{\mathcal{D}} = \mathcal{S}'_{\mathcal{D}'}.$$

From definition of *Val* in Fig.1, we get

$$\mathit{Val}(id,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = \mathcal{S}_{\mathcal{D}}(\mathcal{E}(id)).$$

There is another case to consider, if $\mathcal{E}(id) \notin \mathcal{D}$.
Then, from the [illegal] rule of $\vdash_{exp}$ definition in Fig.12, we have

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

So, here we have

$$\text{val} = \perp$$

and

$$\mathcal{S}_{\mathcal{D}} = \texttt{illegal}$$

From definition of *Val* in Fig.1, we get

$$\mathit{Val}(id,\ \mathcal{E},\ \texttt{illegal}) = \perp.$$

LEMMA **6.5**

$$\mathcal{C}_{\mathcal{V}}(\neg(*v = \mathsf{undef}),\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \Rightarrow \mathit{Val}(v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \in \mathcal{D}$$

To prove this lemma, let us suppose

$$Val(v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \notin \mathcal{D}.$$

Then from

$$Val(*v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = \mathcal{S}_{\mathcal{D}}(\ Val(v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}))$$

we get

$$Val(*v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = \bot.$$

From

$$Val(\mathsf{undef},\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = \bot$$

we get

$$Val(\mathsf{undef},\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = Val(*v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}).$$

So, from definition of concretisation function, we get

$$\mathcal{C}_{\mathcal{V}}(\mathsf{undef} = *v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})$$

which is equivalent to

$$\underline{\text{not}}(\mathcal{C}_{\mathcal{V}}(\neg(\mathsf{undef} = *v),\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})).$$

We have proved the contrapositive of the lemma.

LEMMA **6.6**

$$\mathcal{C}_{\mathcal{V}}(P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \quad \Rightarrow \quad Val(v[\![v_2/v_1]\!]^{\mathcal{V}}_P,\ \mathcal{E},\ S_{\mathcal{D}}) = Val(v,\ \mathcal{E},\ S_{\mathcal{D}}[Val(v_2,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})/Adr(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})])$$

We prove this lemma by induction on the structure of v:
Let us consider the case $v = id$

- if $P \Rightarrow \&id = \&v_1$
  $\mathcal{C}_{\mathcal{V}}(P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \Rightarrow Adr(id,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = Adr(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})$ from the definition of correspondence relation (Fig.1) and the definition of *Adr*.
  $v[\![v_2/v_1]\!]^{\mathcal{V}}_P = v_2$ from the definition of $[\![]\!]^{\mathcal{V}}_P$ (Fig.4).
  Then we get $Val(v_2,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = Val(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}[Val(v_2,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})/Adr(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})])$.
- if $P \Rightarrow \neg(\&id = \&v_1)$
  $\mathcal{C}_{\mathcal{V}}(P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \Rightarrow Adr(id,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \neq Adr(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})$ by definition of correspondence relation. (Fig.1)
  $v[\![v_2/v_1]\!]^{\mathcal{V}}_P = v$ by definition of $[\![]\!]^{\mathcal{V}}_P$. (Fig.4)
  Then we get $Val(v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = Val(v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}[Val(v_2,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})/Adr(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})])$.

LEMMA **6.7**

$$\mathcal{C}_{\mathcal{V}}(Q[\![v_2/v_1]\!]^{\mathcal{V}}_P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})\ \textit{and}\ \mathcal{C}_{\mathcal{V}}(P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \Rightarrow \mathcal{C}_{\mathcal{V}}(Q,\ \mathcal{E},\ S_{\mathcal{D}}[Val(v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})/Adr(v_1,\ \mathcal{E},\ S_{\mathcal{D}})])$$

The proof is straightforward (by induction on Q).

THEOREM 2.2

if $\{P\}\, S\, \{Q\}$ can be proven using the rules of Fig. 3 then

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

The proof of Theorem 2.2 is made by induction on the structure of proof tree of $\{P\}\, S\, \{Q\}$. We consider in detail two interesting cases for S here: the assignment statement and the `while` loop. We sketch the proof for the remaining cases.

Correctness of assignment statement.

We have

$$\{P\}\ v_1 := v_2\ \{Q\}.$$

We consider $(\mathcal{E}, \mathcal{S}_{\mathcal{D}})$ such that

$$\mathcal{C}_{\mathcal{V}}(P,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}).$$

From rule [assign] in Fig.12, we get

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

We want to prove

$$\mathcal{C}_{\mathcal{V}}(Q,\, \mathcal{E},\, \mathcal{S}''_{\mathcal{D}''}[\mathrm{val}_2/a_1]).$$

From the rule of Fig.12

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

we can deduce with lemma 6.4:

$$\mathrm{Adr}(v_1,\, \mathcal{E},\, S'_{\mathcal{D}'}) = a_1$$

and

$$\mathit{Val}(v_2,\, \mathcal{E},\, \mathcal{S}''_{\mathcal{D}''}) = \mathrm{val}_{\in}$$

Let us consider property P in disjunctive normal form, i.e. in the form $\bigvee_i P_i$. Let us first suppose $i = 1$. (the generalisation to $\bigvee_i$ is straightforward)

From the rule of Fig.3

$$\frac{\{P\}\ v_1\ \{P\} \quad \{P\}\ v_2\ \{P\} \quad P \Rightarrow Q[\![v_2/v_1]\!]^{\mathcal{V}}_P}{\{P\}\ v_1{=}v_2\ \{Q\}}$$

and the rule of Fig.12 just above, we get the following hypothesis:

1. $P \Rightarrow Q[\![v_2/v_1]\!]^{\mathcal{V}}_P$
2. $\mathcal{C}_{\mathcal{V}}(P,\ \mathcal{E},\ \mathcal{S}''_{\mathcal{D}''})$

Let us show the wanted property:

$$\mathcal{C}_{\mathcal{V}}(Q,\ \mathcal{E},\ \mathcal{S}''_{\mathcal{D}''}[\mathrm{val}_{\in}/\dashv_{\infty}]).$$

From (1) $\wedge$ (2) and lemma 6.1 we get

$$\mathcal{C}_{\mathcal{V}}(Q[\![v_2/v_1]\!]^{\mathcal{V}}_P,\ \mathcal{E},\ \mathcal{S}''_{\mathcal{D}''})$$

Let us show:

$(\mathcal{C}_{\mathcal{V}}(Q[\![v_2/v_1]\!]^{\mathcal{V}}_P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \wedge \mathcal{C}_{\mathcal{V}}(\mathcal{P},\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})) \Rightarrow \mathcal{C}_{\mathcal{V}}(\mathcal{Q},\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}[\mathit{Val}(\sqsubseteq_{\in},\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})/\mathrm{Adr}(\sqsubseteq_{\infty},\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})])$

by induction on Q:

- $Q = (v = v')$:
  We need to prove

  $\mathcal{C}_{\mathcal{V}}(v[\![v_2/v_1]\!]^{\mathcal{V}}_P = v'[\![v_2/v_1]\!]^{\mathcal{V}}_P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \Rightarrow \mathcal{C}_{\mathcal{V}}(v = v',\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}[\mathit{Val}(v_2,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})/\mathrm{Adr}(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})])$

  From definition of concretisation in Fig.1 and $\mathcal{C}_{\mathcal{V}}(Q[\![v_2/v_1]\!]^{\mathcal{V}}_P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})$, we get

  $$\mathit{Val}(v[\![v_2/v_1]\!]^{\mathcal{V}}_P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = \mathit{Val}(v'[\![v_2/v_1]\!]^{\mathcal{V}}_P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})$$

  Let us show

  $\mathit{Val}(v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}[\mathit{Val}(v_2,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})/\mathrm{Adr}(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})]) = \mathit{Val}(v',\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}[\mathit{Val}(v_2,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})/\mathrm{Adr}(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})])$

  Lemma 6.6 implies

  $$\mathit{Val}(v[\![v_2/v_1]\!]^{\mathcal{V}}_P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = \mathit{Val}(v,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}[\mathit{Val}(v_2,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})/\mathrm{Adr}(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})])$$

  and

  $$\mathit{Val}(v'[\![v_2/v_1]\!]^{\mathcal{V}}_P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = \mathit{Val}(v',\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}[\mathit{Val}(v_2,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})/\mathrm{Adr}(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})])$$

- $Q = (v \mapsto v')$
  We need to prove

$$\mathcal{C}_{\mathcal{V}}((v \mapsto v')[\![v_2/v_1]\!]^{\mathcal{V}}_P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \Rightarrow \mathcal{C}_{\mathcal{V}}(v \mapsto v',\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}[\mathit{Val}(v_2,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})/\mathrm{Adr}(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})])$$

  which is equivalent to:

$$\mathcal{C}_{\mathcal{V}}(([[[(\tilde{v} \mapsto \tilde{v}') \wedge \neg(\tilde{v} = \&v_1)) \vee ((\tilde{v} \mapsto w \mapsto \tilde{v}') \wedge \neg(\tilde{v} = \&v_1) \wedge \neg(w = \&v_1))]$$
$$\wedge[\forall x \in \mathcal{V},\ (\neg(\tilde{v} \mapsto \tilde{x}) \vee (\tilde{x} = \tilde{v}') \vee\ \neg(\tilde{x} \mapsto \tilde{v}'))]] \vee [(\tilde{v} = \&v_1) \wedge (\tilde{v}' = v_2)]), \mathcal{E}, \mathcal{S}_{\mathcal{D}})$$
$$\Rightarrow$$

$$\begin{aligned}
\exists \alpha_1 \ldots \alpha_k,\ \alpha_1 &= \mathit{Val}(v,\ \mathcal{E},\ S'_{\mathcal{D}'}) && (1)\\
\alpha_k &= \mathit{Val}(v',\ \mathcal{E},\ \mathcal{S}'_{\mathcal{D}'}) && (2)\\
\forall 1 \le i < k &\quad \mathcal{S}'_{\mathcal{D}'}(\alpha_i) = \alpha_{i+1} && (3)\\
\forall 1 < i < k &\quad \forall x \in \mathcal{V}, \alpha_i \neq \mathit{Val}(x,\ \mathcal{E},\ \mathcal{S}'_{\mathcal{D}'}) && (4)
\end{aligned}$$

  The abstract property appearing in this concretisation can be written in the form of a disjunction of three terms:
  $[(\tilde{v} \mapsto \tilde{v}') \wedge \neg(\tilde{v} = \&v_1) \wedge \forall x \in \mathcal{V},\ (\neg(\tilde{v} \mapsto \tilde{x}) \vee (\tilde{x} = \tilde{v}') \vee \neg(\tilde{x} \mapsto \tilde{v}'))]$ (A)
  $\vee[((\tilde{v} \mapsto w \mapsto \tilde{v}') \wedge \neg(\tilde{v} = \&v_1) \wedge \neg(w = \&v_1)) \wedge \forall x \in \mathcal{V},\ (\neg(\tilde{v} \mapsto \tilde{x}) \vee (\tilde{x} = \tilde{v}') \vee \neg(\tilde{x} \mapsto \tilde{v}'))]$ (B)
  $\vee[(\tilde{v} = \&v_1) \wedge (\tilde{v}' = v_2)]$ (C)
  To prove $\mathcal{C}_{\mathcal{V}}((A) \vee (B) \vee (C)) \Rightarrow ((1) \wedge (2) \wedge (3) \wedge (4))$, we first prove $\mathcal{C}_{\mathcal{V}}(A) \Rightarrow ((1) \wedge (2) \wedge (3) \wedge (4))$:

$$\begin{aligned}
\mathcal{C}_{\mathcal{V}}(\tilde{v} \mapsto \tilde{v}',\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \Rightarrow\ \exists \alpha_1 \ldots \alpha_k, &\quad \alpha_1 = \mathit{Val}(\tilde{v},\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}),\ \alpha_k = \mathit{Val}(\tilde{v}',\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}),\\
&\quad \forall 1 \le i < k,\ S_{\mathcal{D}}(\alpha_i) = \alpha_{i+1},\\
&\quad \forall 1 < i < k,\ \forall x \in \mathcal{V}, \alpha_i \neq \mathit{Val}(x,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})
\end{aligned}$$

  From the definition of $\mathcal{V}$ we have: $\&v_1 \in \mathcal{V}$, so

$$\forall i,\ 1 < i < k,\ \alpha_i \neq\ \mathrm{Val}(\&v_1, \mathcal{E}, S_{\mathcal{D}})$$

  and

$$\forall i,\ 1 < i < k,\ S'_{\mathcal{D}'}(\alpha_i) = \alpha_{i+1}$$

  but

$$\begin{aligned}
\mathcal{C}_{\mathcal{V}}(\neg(\tilde{v} = \&v_1),\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) &\Rightarrow \mathit{Val}(\tilde{v},\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \neq \mathit{Adr}(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})\\
&\Rightarrow \alpha_1 \neq \mathit{Adr}(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})
\end{aligned}$$

  so

$$S'_{\mathcal{D}'}(\alpha_1) = \alpha_2 \Rightarrow (3)$$

  Lemma 6.6 implies

$$\alpha_1 = \mathit{Val}(v[\![v_1/v_2]\!]^{\mathcal{V}}_P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = \mathit{Val}(v,\ \mathcal{E},\ S'_{\mathcal{D}'})$$

  Applying the same reasoning to α_k, we get $(1) \wedge (2)$

Let us suppose $x \in \mathcal{V}$ and $\alpha_j = Val(x, \mathcal{E}, S'_{\mathcal{D}'}) \neq \alpha_k$ (i)
From lemma 6.6, we get

$$Val(x[\![v_1/v_2]\!]^{\mathcal{V}}_P, \mathcal{E}, \mathcal{S}_{\mathcal{D}}) = Val(x, \mathcal{E}, S'_{\mathcal{D}'})$$

So,

$$\alpha_1 = Val(\tilde{v}, \mathcal{E}, \mathcal{S}_{\mathcal{D}}) \mapsto \ldots \mapsto \alpha_j = Val(\tilde{x}, \mathcal{E}, \mathcal{S}_{\mathcal{D}})$$

and

$$\forall k' \in [2, j-1],\ \forall z \in \mathcal{V},\ \alpha_{k'} \neq Val(z, \mathcal{E}, \mathcal{S}_{\mathcal{D}}) \qquad (ii)$$

In the same way:

$$\alpha_j = Val(\tilde{x}, \mathcal{E}, \mathcal{S}_{\mathcal{D}}) \mapsto \ldots \mapsto \alpha_k = Val(v', \mathcal{E}, S'_{\mathcal{D}'})$$

and

$$\forall k' \in [j+1, k-1],\ \forall z \in \mathcal{V},\ \alpha_{k'} \neq Val(z, \mathcal{E}, \mathcal{S}_{\mathcal{D}}) \qquad (iii)$$

$(i) \wedge (ii) \wedge (iii)$ contradicts

$$\mathcal{C}_{\mathcal{V}}((\forall x \in \mathcal{V},\ \neg(\tilde{v} \mapsto \tilde{x}) \vee (\tilde{x} = \tilde{v}') \vee \neg(\tilde{x} \mapsto \tilde{v}')), \mathcal{E}, \mathcal{S}_{\mathcal{D}})$$

So, we get (4).

Let us now consider the second term (B) of the disjunction:

$$\mathcal{C}_{\mathcal{V}}((\tilde{v} \mapsto w \mapsto \tilde{v}') \wedge \neg(\tilde{v} = \&v_1) \wedge \neg(w = \&v_1), \mathcal{E}, \mathcal{S}_{\mathcal{D}})$$

$$\Rightarrow (1) \wedge (2) \wedge (3) \wedge (4)$$

The proof for $(1) \wedge (2) \wedge (3)$ is obvious.
In order to show (4), we consider two cases:

- $\tilde{v} \mapsto \ldots \mapsto \tilde{x} \mapsto \ldots \mapsto w$
- $w \mapsto \ldots \mapsto \tilde{x} \mapsto \ldots \tilde{v}'$

We get the same contradiction as in the previous case.

The third term (C) of the disjunction is:

$$\mathcal{C}_{\mathcal{V}}((\tilde{v} = \&v_1) \wedge (\tilde{v}' = v_2), \mathcal{E}, \mathcal{S}_{\mathcal{D}})$$

$$\Rightarrow (1) \wedge (2) \wedge (3) \wedge (4)$$

We have

$$\mathcal{C}_{\mathcal{V}}((\tilde{v} = \&v_1), \mathcal{E}, \mathcal{S}_{\mathcal{D}}) \Rightarrow Val(\tilde{v}, \mathcal{E}, \mathcal{S}_{\mathcal{D}}) = Adr(v_1, \mathcal{E}, \mathcal{S}_{\mathcal{D}})$$

Lemma 6.6 implies:

$$Val(\tilde{v},\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = Val(v,\ \mathcal{E},\ S'_{\mathcal{D}'})$$

and

$$Adr(v_1,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = Adr(v_1,\ \mathcal{E},\ S'_{\mathcal{D}'})$$

We have

$$\mathcal{C}_{\mathcal{V}}((\tilde{v}' = v_2),\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \Rightarrow Val(\tilde{v}',\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = Val(v_2,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})$$

Lemma 6.6 implies:

$$Val(\tilde{v}',\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = Val(v',\ \mathcal{E},\ S'_{\mathcal{D}'})$$

and

$$Val(v_2,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) = Val(v_1,\ \mathcal{E},\ S'_{\mathcal{D}'})$$

Let α_1 be $Adr(v_1,\ \mathcal{E},\ S'_{\mathcal{D}'})$ and α_2 be $Val(v_1,\ \mathcal{E},\ S'_{\mathcal{D}'})$
We have

$$S'_{\mathcal{D}'}(\alpha_1) = \alpha_2$$

which entails (1) $\wedge$ (2) $\wedge$ (3).
Since $k = 2$, we get (4).

Correctness of `while` loop:

We assume:

1. $\{P\}$ `while`$(E)S$ $\{P \wedge \overline{!E}\}$
2. $\mathcal{C}_{\mathcal{V}}(P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}})$
3. $\mathcal{E} \vdash_{stat} <$`while`$(E)S, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}'_{\mathcal{D}'}$

and we want to prove:

$$\mathcal{C}_{\mathcal{V}}(P \wedge \overline{!E},\ \mathcal{E},\ \mathcal{S}'_{\mathcal{D}'})$$

The `while`-false case is obvious.
Let us consider the `while`-true case.

$$\frac{\mathcal{E} \vdash_{exp} <E, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <b, \mathcal{S}''_{\mathcal{D}''}> \qquad \dfrac{\mathcal{E} \vdash_{stat} <S, \mathcal{S}''_{\mathcal{D}''}> \rightsquigarrow \mathcal{S}'''_{\mathcal{D}'''} \quad \mathcal{E} \vdash_{stat} <\texttt{while}\,(E)\ S, \mathcal{S}'''_{\mathcal{D}'''}> \rightsquigarrow \mathcal{S}'_{\mathcal{D}'}}{\mathcal{E} \vdash_{stat} <S; \texttt{while}\,(E)\ S, \mathcal{S}''_{\mathcal{D}''}> \rightsquigarrow \mathcal{S}'_{\mathcal{D}'}}}{\mathcal{E} \vdash_{stat} <\texttt{while}(E)S, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow \mathcal{S}'_{\mathcal{D}'}}$$

By Lemma 6.2

$$\mathcal{E} \vdash_{exp} <E, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <b, \mathcal{S}''_{\mathcal{D}''}>,\ b \neq 0\ \Rightarrow\ \mathcal{C}_{\mathcal{V}}(\overline{E},\ \mathcal{E},\ S''_{D''}) \qquad (1)$$

We have

$$\{P\}\ E\ \{P\} \wedge \mathcal{C}_{\mathcal{V}}(P,\ \mathcal{E},\ \mathcal{S}_{\mathcal{D}}) \wedge \mathcal{E} \vdash_{exp} <E, \mathcal{S}_{\mathcal{D}}> \rightsquigarrow <b, \mathcal{S}''_{\mathcal{D}''}> \ \Rightarrow\ \mathcal{C}_{\mathcal{V}}(P,\ \mathcal{E},\ S''_{D''}) \qquad (2)$$

$(1) \wedge (2) \Rightarrow \mathcal{C}_{\mathcal{V}}(Q \wedge \overline{E},\, \mathcal{E},\, S''_{\mathcal{D}''})$. (3)

By induction hypothesis, we get:

$$(3) \wedge\ \{Q \wedge \overline{E}\}\ S\ \{P\}\ \wedge\ \mathcal{E} \vdash_{stat} <S, \mathcal{S}''_{\mathcal{D}''}> \rightsquigarrow \mathcal{S}'''_{\mathcal{D}'''}\ \Rightarrow\ \mathcal{C}_{\mathcal{V}}(P,\, \mathcal{E},\, S'''_{\mathcal{D}'''}) \quad (4)$$

Applying the induction hypothesis again, we have:

$$(4) \wedge\ \{P\}\ \texttt{while}(E)S\ \{P \wedge \overline{!E}\}\ \wedge\ \mathcal{E} \vdash_{stat} <\texttt{while}\,(E)\ S, \mathcal{S}'''_{\mathcal{D}'''}> \rightsquigarrow \mathcal{S}'_{\mathcal{D}'}$$

$$\Rightarrow\ \mathcal{C}_{\mathcal{V}}(P \wedge \overline{!E},\, \mathcal{E},\, \mathcal{S}'_{\mathcal{D}'})$$

Sketch for the correctness of other cases

- `if`-true and `if`-false cases are treated in the same way. They are straightforward consequences of lemma 6.2.
- the `free` case is similar to the assignment case, with $Val(\mathsf{undef},\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}) = \bot$ and $\mathcal{S}_{\mathcal{D}}(a) = \bot$ if $a \notin \mathcal{D}$.
- the weakening case is a straightforward consequence of lemma 6.1.

Appendix 3 Proofs of Theorems 3.2 and 3.3

Proof of Theorem 3.2

Let us first show three useful lemmas.

Definition **6.8** *A property is said to be in anf if it is in adnf with no disjunction.*

Lemma **6.9** *If P is in anf w.r.t. Var_{Prog} and $*v_2 \notin Var_{Prog}$ then $Complete_{*v_2}(P)$ is in adnf w.r.t. $Var_{Prog} \cup \{*v_2\}$*

Proof:

Since no property between two variables of $\mathrm{Var}_{\mathrm{Prog}}$ is taken off by the *Complete* function, we have just to check properties involving $*v_2$. We consider different cases over v_2 corresponding to the different cases in the definition of the $Complete_{*v_2}$ function.

1. If $P \models v_2 = \mathsf{undef}$ then $*v_2$ verifies the same properties than v_2 which are added by *Closure*:
   - $(v_2 = \mathsf{undef}) \wedge (v_2 = v_2) \wedge (*v_2 = \mathsf{undef}) \succ (v_2 = *v_2)$.
   - $(v_2\ \mathrm{op}\ v) \wedge (v_2 = *v_2) \wedge (v = v) \succ (*v_2\ \mathrm{op}\ v)$.
   - $(v\ \mathrm{op}\ v_2) \wedge (v = v) \wedge (v_2 = *v_2) \succ (v\ \mathrm{op}\ v_2)$.

2. If $P \models\!\Rightarrow (v_2 \mapsto y) \wedge (\&y = v_2)$, $*v_2$ verifies the same properties than y which are added by *Closure*.

3. Otherwise, if $P \models\!\Rightarrow (v_2 \mapsto y)$, $Closure(P \wedge (*v_2 = y))$ is an *anf* and we have to show that $Insert(P, *v_2, y)$ is an *adnf*.
We just check that all properties on $*v_2$ are added since other properties remain unchanged or are translated into their negation (by *Replace*).

   - Properties of the form $(\neg)(*v_2 = v)$: *Mk-node* returns $(*v_2 = *v_2)$ and $\neg(*v_2 = v), \forall v \neq *v_2$.
   - Properties of the form $(\neg)(*v_2 \mapsto v)$: for all v such that $v \neq y$, the property $\neg(*v_2 \mapsto v)$ is added by first rule of $\mathrm{NF}_1^{*v_2}$. *Closure* and *Replace* add the property $*v_2 \mapsto v$ for others v.
   - Properties of the form $(\neg)(v \mapsto *v_2)$: *Replace* adds $v \mapsto *v_2$ for all v such that $v = v_2$.
     For all v pointing to a variable different from $*v_2$, *Mk-node* and first rule of $\mathrm{NF}_1^{*v_2}$ add the property $\neg(v \mapsto *v_2)$.
     The third rule of $\mathrm{NF}_1^{*v_2}$ adds the property $\neg(v \mapsto *v_2)$ for other variables v (those pointing to no variable).

4. Otherwise, v_2 is such that $\forall v \in P$, $P \models\!\Rightarrow \neg(v_2 \mapsto v)$. We have to prove that $Add(P, *v_2)$ is in *adnf* w.r.t. $\mathrm{Var}_{\mathrm{Prog}} \cup \{*v_2\}$.

   - Properties of the form $(\neg)(*v_2 = v)$ are added by *Mk-node*.
   - Properties of the form $(\neg)(*v_2 \mapsto v)$: $\neg(*v_2 \mapsto v)$ is added by $End(P, *v_2)$
   - Properties of the form $(\neg)(v \mapsto *v_2)$: for all v equal to v_2, the property $v \mapsto *v_2$ is added by *Closure*.
     If v is such that $\forall w \in P$, $P \models\!\Rightarrow \neg(v \mapsto w) \wedge \neg(v = v_2)$, then the first rule of $\mathrm{NF}_2^{*v_2}$ creates a disjunction containing the desired property (or its negation).
     If v points to a variable different from $*v_2$ then the second rule of $\mathrm{NF}_2^{*v_2}$ adds the property $\neg(v \mapsto *v_2)$.

LEMMA **6.10** *If P is in anf then $Assign_{v_2}^{v_1}(P)$ is in adnf w.r.t. Var_{Prog}.*

Proof:

We prove for each property $(v \text{ op } v')$, op = '=' or '$\mapsto$', that this property or its negation appears in $Assign_{v_2}^{v_1}(P)$.

1. Let us first suppose $*v_2 \in \mathrm{Var}_{\mathrm{Prog}}$.
We have to prove P is in *anf* $\Rightarrow Closure(Produce_{v_2}^{v_1} P)$ is in *adnf* w.r.t. $\mathrm{Var}_{\mathrm{Prog}}$.

   - $P \models\!\Rightarrow \neg(v = \&v_1)$

- if v and v' are not in $Affected_{v_1}$, $(v \text{ op } v')$ remains unchanged.
- if v (resp. v') is in $Affected_{v_1}$, then v (resp. v') is in $Subst_i$, and we get $(v \text{ op } v')$ or $\neg(v \text{ op } v')$ from $(\neg)(*^i v_2 \text{ op } v')$ (resp. $(\neg)(v \text{ op } *^i v_2)$)(which is certainly in P, since P is in *anf*).

- $P \models\!\!\Rightarrow v = \&v_1$
  If op $= \mapsto$ then $Prod(x \text{ op } y) = (\&v_1 \mapsto v_1) \wedge \bigwedge_{\substack{v \notin Subst_0 \\ \text{and } P \models\!\!\Rightarrow \neg(v = v_2)}} \neg(\&v_1 \mapsto v)$. If $v' \notin Subst_0$ and $P \models\!\!\Rightarrow \neg(v' = v_2)$ then *Closure* adds the property $\neg(v \mapsto v')$. If $v' \in Subst_0$ then $Prod_{v_2=v_2}$ returns the property $v' = v_1$ and *Closure* adds $(v \mapsto v')$. If $P \models\!\!\Rightarrow (v' = v_2)$ then $Prod_{v'=v_2}$ returns the property $v' = v_1$ and *Closure* adds $(v \mapsto v')$.
  If op $= \neg \mapsto$ then $Prod(x \text{ op } y) =$ True. So we have to verify that $(x \text{ op } y)$ or is negation is added in $Assign_{v_2}^{v_1}(P)$. If $v' \in Subst_0$ or $P \models\!\!\Rightarrow (v' = v_2)$ then the same reasoning as above applies. Otherwise, as $P \models\!\!\Rightarrow v = \&v_1$, we have the property $(v \mapsto v_1)$ occuring in P. $Prod_{v \mapsto v_1} = (\&v_1 \mapsto v_1) \wedge \bigwedge_{\substack{v \notin Subst_0 \\ \text{and } P \models\!\!\Rightarrow \neg(v = v_2)}} \neg(\&v_1 \mapsto v)$, so we get the property $\neg(\&v_1 \mapsto v')$ and *Closure* adds the property $\neg(v \mapsto v')$.

2/ For the case $*v_2 \notin \text{Var}_{\text{Prog}}$, lemma 6.9 implies that $Complete_{*v_2}(P)$ is in *adnf* w.r.t. $\text{Var}_{\text{Prog}} \cup \{*v_2\}$. We want to prove that $Closure(Produce_{v_2}^{v_1} Complete_{*v_2}(P))$ is in *adnf* w.r.t. Var_{Prog}. The reasoning is similar than in the case $*v_2 \in \text{Var}_{\text{Prog}}$except that we have to verify $*v_2$ has disappeared in the resulting property. This is a direct consequence of the definition of the *Prod* function where all occurrences of the variable $*v_2$ are replaced by the variables of the set $Subst_1$.

LEMMA **6.11** *P is in adnf $\Rightarrow$ $Alloc(P, v)$ is in adnf.*

We have to prove as for the previous lemma that each property $(v \text{ op } v')$, op = '=' or '$\mapsto$', or its negation appears in the resulting property. There is no difficulty here.

Proof of Theorem 3.2

Lemma 6.10 shows the property for assignment and `free` statements. Lemma 6.11 shows the property for the `alloc` case. The proof of the other rules are straightforward, assuming a function that distributes all conjunctions over disjunctions in order to keep a disjunctive normal form at each step of the inference process. This function is simple and we have omitted it for the sake of clarity.

Proof of Theorem 3.3

LEMMA **6.12**

$$P \Rightarrow Closure(P)$$

Proof:
Closure is defined as the normal form of the relation

$$P = P' \wedge (a \text{ op } b) \wedge (a = a') \wedge (b = b') \succ P \wedge (a' \text{ op } b').$$

To prove the wanted property, it is sufficient to prove

$$(a \text{ op } b) \wedge (a = a') \wedge (b = b') \Rightarrow (a' \text{ op } b').$$

which is a direct consequence of

$$(v_1 = v_2) \wedge P \;\; \Rightarrow \;\; P[v_2/v_1].$$

(Fig.2)

LEMMA **6.13**

$$P \Rightarrow \mathit{Complete}_{*x}(P)$$

Proof:

- If $P \models\!\Rightarrow (x = \mathsf{undef})$, we get the wanted property from $x = \mathsf{undef} \Rightarrow *x = \mathsf{undef}$ and lemma 6.12:
  $P = P' \wedge (x = \mathsf{undef}) \Rightarrow P \wedge (*x = \mathsf{undef}) \Rightarrow \mathit{Closure}(P \wedge (*x = \mathsf{undef}))$
  which is equal to $\mathit{Complete}_{*v_2}(P)$

- If $P \models\!\Rightarrow (x \mapsto y) \wedge (x = \&y)$, we get the wanted property from
  $(x = \&y) \Rightarrow (*x = *\&y)$ and $(*\&y = y)$ and lemma 6.12.

- If $P \models\!\Rightarrow (x \mapsto y)$, we get the wanted property from $(x \mapsto y) \Rightarrow (*x = y) \vee (*x \mapsto y)$, lemma 6.12 and the property $P \Rightarrow \mathit{Insert}(P, *x, y)$ which is not proved here (the proof is similar to the proof of lemma 6.12).

- otherwise, we get the wanted property from $P \models\!\Rightarrow (x \mapsto *x)$ and the property $P \Rightarrow \mathit{Add}(P, *x)$ (whose proof is similar to the proof of lemma 6.12).

Proof of Lemma 3.4

$$P \Rightarrow \text{Assign}^{v_1}_{v_2}(P)[\![v_2/v_1]\!]^{\mathcal{V}}_P$$

First case:
Let us suppose $*v_2 \in \mathrm{Var}_{\mathrm{Prog}}$. We have to prove

$$P \Rightarrow \mathrm{Produce}_{v_2}^{v_1}(P)[\![v_2/v_1]\!]_P^{\mathcal{V}}$$

In order to prove this property, we prove

$$\text{If } P \Rightarrow P' \text{ then } P \Rightarrow \mathit{Prod}_P P'[\![v_2/v_1]\!]_P^{\mathcal{V}}.$$

Then we get for all i

$$P_i \Rightarrow \bigvee \mathit{Prod}_{P_i} P_i[\![v_2/v_1]\!]_P^{\mathcal{V}}.$$

Since the property is true for all i, we get

$$\bigvee(P_i) \Rightarrow \bigvee \mathit{Prod}_{P_i} P_i[\![v_2/v_1]\!]_P^{\mathcal{V}}$$

which is equivalent to

$$P \Rightarrow \bigvee \mathit{Prod}_{P_i} P_i[\![v_2/v_1]\!]_P^{\mathcal{V}}.$$

By induction on P':

- Trivial for P' =True and by induction hypothesis for $P' = P_1 \wedge P_2$.
- $P' = x$ op y

  - If op = '=' or '$\neg$=', then

$$\mathit{Prod}_P(x \text{ op } y) = \bigwedge_{v,v' \in (\mathit{Subst}_i, \mathit{Subst}_j)_{i,j \in \{0,1\}}} \left\{ \begin{array}{lll} (v \text{ op } v') & \text{if } x = *^i v_2 \wedge y = *^j v_2 & \text{otherwise True} \\ (x \text{ op } v) & \text{if } y = *^j v_2 \wedge x \in \mathrm{Var}_{\mathrm{Prog}} - \mathit{Affected}_{v_1} & \text{otherwise True} \\ (v \text{ op } y) & \text{if } x = *^i v_2 \wedge y \in \mathrm{Var}_{\mathrm{Prog}} - \mathit{Affected}_{v_1} & \text{otherwise True} \\ (x \text{ op } y) & \text{if } x \in \mathrm{Var}_{\mathrm{Prog}} - \mathit{Affected}_{v_1} \wedge y \in \mathrm{Var}_{\mathrm{Prog}} - \mathit{Affected}_{v_1} & \\ & & \text{otherwise True } \} \end{array} \right.$$

    Since $(v, v') \in (\mathit{Subst}_i, \mathit{Subst}_j)$, they are such that $v[\![v_2/v_1]\!]_P^{\mathcal{V}} = *^i v_2$ and $v'[\![v_2/v_1]\!]_P^{\mathcal{V}} = *^j v_2$.
    So, we get

$$\mathit{Prod}_P(x \text{ op } y)[\![v_2/v_1]\!]_P^{\mathcal{V}} = (x \text{ op } y) \text{ or True.}$$

    We get the wanted property:

$$P' = x \text{ op } y \Rightarrow x \text{ op } y \text{ or } P' \Rightarrow \text{ True.}$$

– If op = '$\mapsto$' or '$\neg \mapsto$', two cases arise:

* $P \models\!\Rightarrow x = \&v_1$

$$Prod_P(x \text{ op } y) \quad = \quad \underline{\text{if}}\ \text{op}\ = \mapsto\ \underline{\text{then}}\ (\&v_1 \mapsto v_1) \wedge \bigwedge_{\substack{v \notin \text{Subst}_0 \\ \text{and } P \models\!\Rightarrow \neg(v = v_2)}} \neg(\&v_1 \mapsto v)$$
$$\underline{\text{else if}}\ \text{op}\ = \neg \mapsto\ \underline{\text{then}}\ \text{True}$$

The case op = '$\neg \mapsto$' is immediate since the result is True.
If op = '$\mapsto$', then from fig.4,

$$(\&v_1 \mapsto v_1)[\![v_2/v_1]\!]^{\mathcal{V}}_P = (\&v_1[\![v_2/v_1]\!]^{\mathcal{V}}_P = \&v_1) \wedge (v_1[\![v_2/v_1]\!]^{\mathcal{V}}_P = v_2)$$

which is equal to True since

$$\&v_1[\![v_2/v_1]\!]^{\mathcal{V}}_P = \&v_1$$
$$v_1[\![v_2/v_1]\!]^{\mathcal{V}}_P = v_2$$

For the second term of the conjunction, we get

$$\bigwedge_{\substack{v \notin \text{Subst}_0 \\ \text{and } P \models\!\Rightarrow \neg(v = v_2)}} \neg(\&v_1 \mapsto v)[\![v_2/v_1]\!]^{\mathcal{V}}_P = \bigwedge_{\substack{v \notin \text{Subst}_0 \\ \text{and } P \models\!\Rightarrow \neg(v = v_2)}} \neg(v[\![v_2/v_1]\!]^{\mathcal{V}}_P = v_2)$$

If

$$v \notin \text{Subst}_0 \text{ and } P \models\!\Rightarrow \neg(v = v_2)$$

then

$$P \Rightarrow \neg(v[\![v_2/v_1]\!]^{\mathcal{V}}_P = v_2).$$

* $P \models\!\Rightarrow \neg(x = \&v_1)$
For the case op = '$\mapsto$', we want to show

$$P \Rightarrow Prod_P(x \mapsto y)[\![v_2/v_1]\!]^{\mathcal{V}}_P$$

with

$$P \models\!\Rightarrow (x \mapsto y).$$

From fig.6, we get

$$Prod_P(x \mapsto y) \quad = \bigwedge_{v,v' \in (Subst_i, Subst_j)_{i,j \in \{0,1\}}}$$

$$\{\ \begin{array}{lll} (v \mapsto v') & \text{if } x = *^i v_2 \wedge y = *^j v_2 & \text{otherwise True} \\ (x \mapsto v) & \text{if } y = *^j v_2 \wedge x \in \text{Var}_{\text{Prog}} - Affected_{v_1} & \text{otherwise True} \\ (v \mapsto y) & \text{if } x = *^i v_2 \wedge y \in \text{Var}_{\text{Prog}} - Affected_{v_1} & \text{otherwise True} \\ (x \mapsto y) & \text{if } x \in \text{Var}_{\text{Prog}} - Affected_{v_1} \wedge y \in \text{Var}_{\text{Prog}} - Affected_{v_1} & \\ & & \text{otherwise True}\ \} \end{array}$$

Let us prove

$$P \Rightarrow (v \mapsto v')[\![v_2/v_1]\!]_P^{\mathcal{V}}$$

We have:

1. $v \in subst_i$
2. $v' \in subst_j$
3. $x = *^i v_2$
4. $y = *^j v_2$

From fig.4, we get

$$\begin{array}{rl}(v \mapsto v')[\![v_2/v_1]\!]_P^{\mathcal{V}} = & [(\tilde{v} \mapsto \tilde{v}') \wedge \neg(\tilde{v} = \&v_1) \wedge \forall x \in \mathcal{V},\ (\neg(\tilde{v} \mapsto \tilde{x}) \vee (\tilde{x} = \tilde{v}') \vee \neg(\tilde{x} \mapsto \tilde{v}'))] \\ & \vee[((\tilde{v} \mapsto w \mapsto \tilde{v}') \wedge \neg(\tilde{v} = \&v_1) \wedge \neg(w = \&v_1)) \\ & \wedge\forall x \in \mathcal{V},\ (\neg(\tilde{v} \mapsto \tilde{x}) \vee (\tilde{x} = \tilde{v}') \vee \neg(\tilde{x} \mapsto \tilde{v}'))] \\ & \vee[(\tilde{v} = \&v_1) \wedge (\tilde{v}' = v_2)]\end{array}$$

Let us prove

$$P \Rightarrow (\tilde{v} \mapsto \tilde{v}') \wedge \neg(\tilde{v} = \&v_1) \wedge \forall x \in \mathcal{V},\ (\neg(\tilde{v} \mapsto \tilde{x}) \vee (\tilde{x} = \tilde{v}') \vee \neg(\tilde{x} \mapsto \tilde{v}'))$$

There is no problem for

$$P \Rightarrow (\tilde{v} \mapsto \tilde{v}') \wedge \neg(\tilde{v} = \&v_1).$$

To prove

$$P \Rightarrow \forall x \in \mathcal{V},\ (\neg(\tilde{v} \mapsto \tilde{x}) \vee (\tilde{x} = \tilde{v}') \vee \neg(\tilde{x} \mapsto \tilde{v}'))$$

it is enough to prove that if it exists $x \in \mathcal{V}$ such that

$$P \Rightarrow (\tilde{v} \mapsto \tilde{x}) \wedge \neg(\tilde{x} = \tilde{v}') \wedge (\tilde{x} \mapsto \tilde{v}')$$

then P is not consistent, which is the case since we have also

$$P \Rightarrow (*^i v_2 \mapsto *^j v_2)$$

and

$$\tilde{v} = *^i v_2$$

and

$$\tilde{v}' = *^j v_2$$

The proofs of

$$P \Rightarrow (x \mapsto v')[\![v_2/v_1]\!]_P^{\mathcal{V}}$$
$$P \Rightarrow (v \mapsto y)[\![v_2/v_1]\!]_P^{\mathcal{V}}$$
$$P \Rightarrow (x \mapsto y)[\![v_2/v_1]\!]_P^{\mathcal{V}}$$

are done is the same way.
For the case op = '$\neg \mapsto$', the process is similar.

Second case: $*v_2 \notin \mathrm{Var}_{\mathrm{Prog}}$.

$$Complete_{*v_2}(P) = \bigvee P_i$$

As in the previous case, we show the property

$$P_i \Rightarrow Prod_{P_i} P_i [\![v_2/v_1]\!]_P^{\mathcal{V}}$$

for all P_i.
The proof of this property is similar to the one presented in the previous case.
We then get

$$\bigvee P_i \Rightarrow \bigvee Prod_{P_i} P_i [\![v_2/v_1]\!]_P^{\mathcal{V}}$$

By definition of Prod_P,

$$\bigvee Prod_{P_i} P_i [\![v_2/v_1]\!]_P^{\mathcal{V}} = Produce_{v_2}^{v_1}(\bigvee P_i)[\![v_2/v_1]\!]_P^{\mathcal{V}}$$

Since

$$Complete_{*v_2}(P) = \bigvee P_i$$

we get by Lemma 6.13,

$$P \Rightarrow Produce_{v_2}^{v_1}(Complete_{*v_2}(P))$$

Then Lemma 6.12 concludes the proof of Lemma 3.4.

Proof of Theorem 3.3

The proof is by induction on the derivation tree obtained from the inference system of fig.5

- The assignment case is a direct consequence of Lemma 3.4.

- Let us inspect the `while` case. Let us note $\vdash_3$ "proven by inference system of Fig.3" and $\vdash_5$ "proven by inference system of Fig.5". We suppose

  $$\vdash_5 \{P\}\ \texttt{while}(E)\ S\ \{P_n \wedge \overline{!E}\}.$$

  which is given by the rule

  $$\frac{\begin{array}{ccc} P_0 = P & P_i = P_{i-1} \vee Q_i & \\ \{P_0\}\ E\ \{P_0\} & i \in 1,n\ \{P_i\}\ E\ \{P_i\} & \{P_n\}\ E\ \{P_n\} \\ \{P_0 \wedge \overline{E}\}\ S\ \{Q_1\} & \{P_i \wedge \overline{E}\}\ S\ \{Q_{i+1}\} & P_n \models\!\Rightarrow P_{n-1} \\ & P_i \not\Rightarrow P_{i-1} & \end{array}}{\{P\}\ \texttt{while}(E)\ S\ \{P_n \wedge \overline{!E}\}}$$

  and we want to prove

  $$\vdash_3 \{P\}\ \texttt{while}(E)\ S\ \{P_n \wedge \overline{!E}\}$$

It is enough to show

$$\vdash_3 \{P_n\}\ \texttt{while}(E)\ S\ \{P_n \wedge \overline{!E}\}$$

since the sequence P_i is increasing.
Since the rule for the `while` case in fig.3 is

$$\frac{\{P\}\ E\ \{P\} \quad \{P \wedge \overline{E}\}\ S\ \{P\}}{\{P\}\ \texttt{while}(E)\ S\ \{P \wedge \overline{!E}\}}\ ,$$

we have to prove:

1. $\vdash_3 \{P_n\}\ E\ \{P_n\}$
2. $\vdash_3 \{P_n \wedge \overline{!E}\}\ S\ \{P_n\}$

1. we have

   $$\vdash_5 \{P_n\}\ E\ \{P_n\}$$

   So the induction hypothesis allows us to conclude.
2. we have:

   $$\begin{array}{ll} \vdash_5 \{P_{n-1} \wedge \overline{!E}\}\ S\ \{Q_n\} & (A) \\ P_n = P_{n-1} \vee Q_n & (B) \\ P_n \models\!\Rightarrow P_{n-1} & (C) \end{array}$$

   The induction hypothesis on (A) implies

   $$\vdash_3 \{P_{n-1} \wedge \overline{!E}\}\ S\ \{Q_n\} \quad (D)$$

   Applying the weakening rule on (D) and (C) we get:

   $$\vdash_3 \{P_n \wedge \overline{!E}\}\ S\ \{Q_n\} \quad (E)$$

   and applying the weakening rule again on (E) and

   $$Q_n \models\!\Rightarrow P_{n-1} \vee Q_n$$

   we get the desired property.

- The if-then-else case is proved with the weakening rule $Q_1 \models\!\Rightarrow Q_1 \vee Q_2$ and $Q_2 \models\!\Rightarrow Q_1 \vee Q_2$.
- The `free` case is similar to the assignment case.
- The `alloc` case is done by proving

  $$P \wedge \bigwedge_{v \in \mathrm{Var}} \neg(z = v) \wedge \neg(*z = v) \wedge (*z \mapsto \mathsf{undef}) \Rightarrow \mathit{Alloc}(P)$$

  which is a straightforward consequence of the definition of the *Alloc* function.

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)
ISSN 0249-6399