Static Detection of Pointer Errors: an Axiomatisation and a Checking Algorithm

Pascal Fradet, Ronan Gaugne and Daniel Le Métayer
Irisa/Inria
Campus de Beaulieu,
35042 Rennes, France
[fradet,gaugne,lemetayer]@irisa.fr

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.

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 reviews related work and suggests optimisations and other applications of the analysis.

2 A Hoare Logic for Pointers

The syntax and semantics of the subset of C considered in this paper are provided in Fig. 9 and 10 in the appendix. 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) $\mathcal{D}$.

This paper is concerned with the analysis of blocks of instructions excluding procedure calls and gotos (see [13] for extensions). 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:

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

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 $*v_1 = \mathsf{undef}$ 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.

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 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 "$\bar{\ }$" which transforms a boolean C expression E

$$
\begin{aligned}
\mathcal{C}_{\mathcal{V}}(P,\, \mathcal{E},\, \text{illegal}) &= \mathit{false} \\
\mathcal{C}_{\mathcal{V}}(v_1 = v_2,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}) &= \text{Val}(v_1,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}) = \text{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 = \text{Val}(v_1,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}),\ \alpha_k = \text{Val}(v_2,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}), \\
&\quad\ \forall i\ (1 \le i < k)\, \mathcal{S}_{\mathcal{D}}(\alpha_i) = \alpha_{i+1} \\
&\quad\ \forall i\ (1 < i < k),\ \forall v \in \mathcal{V},\ \alpha_i \neq \text{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} \\
\\
\text{Val}(\text{undef},\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}) &= \bot \\
\text{Val}(\&id,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}) &= \mathcal{E}(id) \\
\text{Val}(id,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}) &= \mathcal{S}_{\mathcal{D}}(\mathcal{E}(id)) \\
\text{Val}(*id,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}) &= \mathcal{S}_{\mathcal{D}}(\text{Val}(id,\, \mathcal{E},\, \mathcal{S}_{\mathcal{D}}))
\end{aligned}
$$

Fig. 1. Correspondence relation

$$
\begin{gathered}
(v_1 = v_2) \wedge P \ \Rightarrow\ P[v_2/v_1] \qquad \&*v = v \qquad *\&v = v \qquad v_1 = v_2 \Rightarrow *v_1 = *v_2 \\
v = v \qquad v_1 = v_2 \wedge v_2 = v_3 \Rightarrow v_1 = v_3 \qquad v_1 = v_2 \Rightarrow v_2 = v_1 \\
v \mapsto *v \qquad v_1 \mapsto v_2 \Rightarrow (v_2 = *v_1) \vee (*v_1 \mapsto v_2) \quad x = \text{undef} \Rightarrow *x = \text{undef} \\
v_1 \mapsto w \wedge w \mapsto v_2 \Rightarrow v_1 \mapsto v_2 \ \text{ with } w \notin \mathcal{V} \\
\\
\frac{P_1 \Rightarrow P \quad P_2 \Rightarrow P}{P_1 \vee P_2 \Rightarrow P} \qquad P \Rightarrow P \qquad \frac{P_1 \Rightarrow P_2 \quad P_2 \Rightarrow P_3}{P_1 \Rightarrow P_3} \\
\\
P_1 \wedge P_2 \Rightarrow P_1 \qquad P_1 \wedge P_2 \Rightarrow P_2 \qquad P_1 \Rightarrow P_1 \vee P_2 \qquad P_2 \Rightarrow P_1 \vee P_2
\end{gathered}
$$

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

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 1.

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

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.

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

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

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

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

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

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

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

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

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

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

Fig. 3. Axiomatics of statements and expressions

- 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 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 $Q[\![v_2/v_1]\!]_P^{\mathcal{V}}$ can be found in Fig. 4. Roughly speaking, $Q[\![v_2/v_1]\!]_P^{\mathcal{V}}$ 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 $[\![v_2/v_1]\!]_P^{\mathcal{V}}$ 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 $id[\![v_2/v_1]\!]_P^{\mathcal{V}}$ 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.

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

Fig. 4. Definition of substitution with aliasing

The following theorems establish the soundness of the inference system:

Theorem 2.

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

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

Corollary 3.

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

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

Corollary 3 is a direct consequence of Theorem 2. It shows that the logic can be used to detect illegal pointer dereferences. The proof of Theorem 2 is made by induction on the form of S. The most difficult part of the proof is the assignment case which relies on the following lemma:

Lemma 4.

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

Lemma 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 5.

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

More details about the proofs of properties stated in this paper can be found in [13].

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 an access chain.

Definition 6. *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:*

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

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

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

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.

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 Q_i is strictly increasing:

$$Q_{i-1} \models\!\!\!\Rightarrow Q_i \quad Q_i \not\models\!\!\!\Rightarrow Q_{i-1}$$

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

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

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

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

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

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

$$\frac{\begin{array}{c} P_0 = P \\ \{P_0\}\ E\ \{Q_0\} \\ \{Q_0 \wedge \overline{E}\}\ S\ \{P_1\} \end{array}\quad \begin{array}{c} i \in 1,n\ \{P_i \vee P_{i-1}\}\ E\ \{Q_i\} \\ \{Q_i \wedge \overline{E}\}\ S\ \{P_{i+1}\} \\ Q_i \not\models\!\!\Rightarrow Q_{i-1} \end{array}\quad \begin{array}{c} \{P_n \vee P_{n-1}\}\ E\ \{Q_n\} \\ Q_n \models\!\!\Rightarrow Q_{n-1} \end{array}}{\{P\}\ \text{while}(E)\ S\ \{Q_n \wedge \overline{!E}\}}$$

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

$$\frac{\{P\}\ v_1\ \{P\}\quad \{P\}\ v_2\ \{P\}}{\{P\}\ v_1{=}v_2\ \{\bigvee_{i=1}^{n} Assign_{v_2}^{v_1}(P_i)\}}\quad \text{with } P = \bigvee_{i=1}^{n} P_i$$

$$\{P\}\ \text{free}(v)\ \{\bigvee_{i=1}^{n} Assign_{\text{undef}}^{*v}(P_i)\}\quad \text{with } P = \bigvee_{i=1}^{n} P_i$$

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

$$\{P\}\ *id\ \{P\}\quad \text{if } \forall i = 1,\ldots,n\ \ P_i \models\!\!\Rightarrow \neg(*id = \text{undef})\quad \text{with } P = \bigvee_{i=1}^{n} P_i$$

$$\{P\}\ \text{z} = \text{alloc}(T)\ \{Alloc(P,z)\}$$

Fig. 5. Rules of the analyzer

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

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

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

so, by definition of substitution, $A \Rightarrow \bigwedge_{v \in Subst_i}(x\ op\ v)[\![v_2/v_1]\!]_{P_i}^{\mathcal{V}}$. When $op =$ " $\mapsto$ " or "$\neg\mapsto$ " we also have to check that $P_i \models\!\!\Rightarrow \neg(x = \&v_1)$ to be able to apply the definition of substitution (see Fig. 4). The three other cases in the definition of $Produce_{v_2}^{v_1}(x\ op\ y)$ 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 $Produce_{v_2}^{v_1}$ 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

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

$$
Produce_{v_2}^{v_1}(P) = Prod_P(P)
$$

$\underline{\text{where}}$

$$
\begin{array}{lll}
Prod_P(P_1 \vee P_2) & = & Prod_P(P_1) \vee Prod_P(P_2) \\
Prod_P(P_1 \wedge P_2) & = & Prod_P(P_1) \wedge Prod_P(P_2) \\
Prod_P(x \text{ op } y) & = & \underline{\text{if}} \text{ op } \in \{\mapsto, \neg \mapsto\} \ \underline{\text{and}}\ P \models\!\!\Rightarrow x = \&v_1 \ \underline{\text{then}} \text{ True} \\
 & & \underline{\text{else}} \displaystyle\bigwedge_{v, v' \in (Subst_i, Subst_j)_{i,j \in \{0,1\}}} \\
 & & \begin{array}{lll}
\{\ (v \text{ op } v') & \text{if } x = *^i v_2 \wedge y = *^j v_2 & \text{otherwise True} \\
\ \ (x \text{ op } v) & \text{if } y = *^j v_2 \wedge x \in \mathrm{Var}_{\mathrm{Prog}} - Affected_{v_1} & \text{otherwise True} \\
\ \ (v \text{ op } y) & \text{if } x = *^i v_2 \wedge y \in \mathrm{Var}_{\mathrm{Prog}} - Affected_{v_1} & \text{otherwise True} \\
\ \ (x \text{ op } y) & \text{if } x \in \mathrm{Var}_{\mathrm{Prog}} - Affected_{v_1} \wedge y \in \mathrm{Var}_{\mathrm{Prog}} - Affected_{v_1} & \\
 & & \text{otherwise True } \}
\end{array} \\
Affected_{v_1} & = & \{x \in \mathrm{Var}_{\mathrm{Prog}} \mid \exists y \text{ suffix of } x, P \models\!\!\Rightarrow y = \&v_1\} \\
Subst_i & = & \{x \in Affected_{v_1} \mid P \models\!\!\Rightarrow x[\![v_2/v_1]\!]_P^{\mathcal{V}} = *^i v_2\} \quad i = 0 \text{ or } i = 1
\end{array}
$$

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

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

Fig. 6. Functions for the assignment rule

added to the pre-condition 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:

$$
\begin{array}{lll}
Affected_y & = & \{y, *y\} \text{ (set of variables with a suffix alias of } y) \\
Subst_0 & = & \{y\} \quad\ \text{(set of variables equated to } x \text{ by substitution)} \\
Subst_1 & = & \{*y\} \quad \text{(set of variables equated to } *x \text{ by substitution)}
\end{array}
$$

Let us assume that the variable $*x$ is not in $\mathrm{Var}_{\mathrm{Prog}}$; $Produce_x^y$ 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 $Complete_{*x}(P)$. Since $P \models\!\!\Rightarrow (x \mapsto z)$, we have $Complete_{*x}(P)= Closure(P \wedge (*x = z)) \vee Insert(P, *x, z)$. 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. $Closure(P \wedge (*x = z))$ adds all missing properties of $*x$ (identical to properties of z) and yields an *adnf*. $Insert(P, *x, z)$ adds the property $(*x \mapsto z)$. It is more involved because other variables pointing

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

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

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

with:

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

$\mathit{Replace}(P \wedge p_1, p_1, p_2) = P \wedge p_2 \wedge \neg p_1$

NF_1^{*x} normal form of the $\succ_1^{*x}$ relation defined as follows:
$P = P' \wedge (a \mapsto b) \wedge \neg(b = c) \succ_1^{*x} P \wedge \neg(a \mapsto c)$
$P = P' \wedge (*x \mapsto a) \wedge (b \mapsto a) \wedge \neg(b = *x) \wedge \neg(b = x) \succ_1^{*x} P \vee \mathrm{Replace}(P, b \mapsto a, b \mapsto *x)$
$P = P' \wedge (*x \mapsto a) \wedge \neg(b \mapsto a) \wedge \neg(b = x) \succ_1^{*x} P \wedge \neg(b \mapsto *x)$

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

with:

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

NF_2^{*x} normal form of the $\succ_2^{*x}$ relation defined as follows:
$P \succ_2^{*x} (P \wedge \neg(b \mapsto *x)) \vee (P \wedge (b \mapsto *x))$ if $b \in \{v \in \mathrm{Var}(P) \mid \nexists w \, P \models\!\!\Rightarrow (v \mapsto w)\}$
$P = P' \wedge (a \mapsto b) \wedge \neg(b = *x) \succ_2^{*x} P \wedge \neg(a \mapsto *x)$

Fig. 7. Normalisation functions for the assignment rule

$$\begin{aligned}
\mathit{Alloc}(P, z) = \mathit{Closure}(\, & P \wedge (z \mapsto *z) \wedge (*z \mapsto \mathrm{undef}) \wedge \bigwedge_{v \in P} (\neg(v \mapsto z)) \\
& \wedge \bigwedge_{v \in P - \{z\}} (\neg(z = v) \wedge \neg(v \mapsto *z)) \\
& \wedge \bigwedge_{v \in P - \{*z\}} (\neg(*z = v) \wedge \neg(z \mapsto v)) \\
& \wedge \bigwedge_{\substack{v \in P \\ P \models\Rightarrow \neg(v = \mathrm{undef})}} (\neg(*z \mapsto v))\ )
\end{aligned}$$

Fig. 8. Functions for the "alloc" rule

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:

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

The following theorems establish the correctness of the analyser.

Theorem 7.

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 8.

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 7 shows that the atomic disjunctive normal form representation is invariant which is crucial to prove the soundness of the algorithm. The proof of theorem 8 is made by induction on the structure of proof of the premise [13]. The difficult part is the rule for assignment which follows from the lemma:

Lemma 9. $P \Rightarrow Assign^{v_1}_{v_2}(P)[\![v_2/v_1]\!]^{\mathcal{V}}_{P}$

4 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 [5, 12, 15, 23] 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 [10, 11]. The alias pairs (x, y) of [10] correspond to $\&x = \&y$ here and the x points-to y relationship of [11] is equivalent to $x = \&y$. One of the most precise published alias analysis is the framework described in [10]. Our analysis is not directly comparable to this one in terms of precision: on one hand, the symbolic access paths used in [10] 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.

– Axiomatisation of pointer and alias relations has been studied for Pascal (see e.g. [6, 7, 21]). 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 [20] 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 [22]. 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 [9] 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) [8]. 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

[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.

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 values 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].
- 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 [4]. 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.

Due to space limitations, we considered only a kernel programming language in this paper. The interested reader can find in [13] the treatment of a number of extensions (procedures, goto, pointer variable declarations) and the presentation of a reasonably complex program involving the construction and destruction of a circular list.

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. 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.
5. 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.

6. A. Bijlsma, *Calculating with pointers*, in Science of Computer Programming 12 (1989) 191-205, North-Holland.
7. R. Cartwright and D. Oppen, *The logic of aliasing*, in Acta Informatica 15, 365-384, 1981 ACM TOPLAS, Vol. 7, 1985, pp. 299-310.
8. 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.
9. 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.
10. 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.
11. 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.
12. D. Evans, *Using specifications to check source code*, in Technical Report, MIT Lab for computer science, Jun. 1994.
13. P. Fradet, R. Gaugne and D. Le Métayer, *An inference algorithm for the static verification of pointer manipulation*, IRISA Research Report 980, 1996.
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. 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.
21. 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.
22. 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.
23. 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.

Appendix

pgm	::= stmt	
stmt	::= if (exp) stmt else stmt	If-else
	| while (exp) stmt	While loop
	| stmt ; stmt	Sequence
	| lexp = exp	Assignment
	| free (lexp)	Runtime deallocation
exp	::= *id*	Variable ($id \in$ Id)
	| $*id$	Pointer dereference
	| $\&id$	Address operator
	| alloc (type)	Runtime allocation
lexp	::= *id*	
	| $*id$	

Fig. 9. Abstract syntax of a subset of C

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

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

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

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

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

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

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

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

Definition of $\vdash_{\text{exp}}$

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

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

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

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

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

Definition of $\vdash_{\text{lexp}}$

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

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

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

Fig. 10. Dynamic semantics for statements and expressions