Separation Analysis for Weakest Precondition-based Verification*

Thierry Hubert[1,2,3] and Claude Marché[2,3]

[1] Dassault Aviation, Saint-Cloud, F-92214
[2] INRIA Futurs, ProVal, Parc Orsay Université, F-91893
[3] Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405

Abstract. The component-as-array model is a widely used technique for modeling heap memory in Weakest Precondition-based deductive verification of pointer programs. We propose a separation analysis which can be integrated in the core of this model. This allows to greatly simplify the verification conditions generated, and thus greatly helps in proving such pointer programs. We illustrate the improvements both in term of scaling up for codes of large size, and in term of simplification of the reasoning for establishing advanced behaviors.

1 Introduction

To perform verification of pointer programs, it is widely known that detection of pointer aliasing is essential. A separation analysis is a technique to automatically detect that two given pointers are not alias to each other. Various separation analyses have been proposed, in the context of advanced static analysis of programs and abstract interpretation.

Deductive verification is the class of verification techniques that are based on logical semantics of programs, starting from the landmark work of Floyd and Hoare [1, 2], and the concepts of logical assertions such as pre- and post-conditions, loop invariants, etc. Compared to static analysis techniques, deductive verification is potentially much more expressive, and is able to establish advanced behaviors of programs, the main drawback being that logical assertions must be given by the programmer.

The most well-known technique for analysing separation in the context of deductive verification is Separation Logic proposed by Reynolds [3]. Yet, as far as we know, no tool implementing Separation Logic has demonstrated a disruptive progress on reasoning on concrete case studies such as industrial embedded code.

For deductive verification, the technique that has shown itself the most effective in practice is the Weakest Precondition (WP) calculus of Dijkstra [4]. It is the base of effective tools such as ESC/Java [5], several tools for Java programs annotated using the Java Modeling Language [6], Spec# [7] for the C# programming language, and a tool of our own for C programs called Caduceus [8].

Dealing with pointer programs in the context of a WP calculus is usually done by providing an appropriate axiomatic modeling of the memory heap. The component-as-array model, coming from an old idea by Burstall [9], has been emphasized by Bornat [10]. Variants of this modeling are used by the tools mentioned above. Unfortunately

* This research is partly supported by "CIFRE" contract 774/2004 with Dassault Aviation company, and ANR RNTL grant "CAT"

Separation Logic is not easily compatible with those techniques based on WP and the component-as-array modeling.

In this paper we propose a new separation analysis, directly suitable for the component-as-array modeling. The main idea is that memory regions that are guaranteed to be separated can be modelled by distinct components of the heap memory model. The method we present here is general and applicable to many programming languages: we present it for C but it is clearly amenable to Java, C#, etc. In Section 2, we define the core language we consider, and recall the principles of the component-as-array modeling.

Section 3 presents our new separation analysis, and define the refined component-as-array memory model it leads to. The first ingredient is a type system for our core language, where pointer types are parameterized by regions. A very important feature is that for a function arguments that has a pointer type, its region can itself by a parameter of the function: this leads to a *polymorphic* type system à la Milner [11]. The main result is Theorem 1: the type system is shown *relatively sound*, that is the interpretation of a well-typed program has the same semantics as the same program interpreted in a classical component-as-array model.

The second ingredient is an algorithm for inferring types (Section 3.3). We do not prove its soundness on paper, but obtain soundness by the mean of the intermediate Why tool: the back-end verification condition generator of Caduceus. The Why input language [12] is a polymorphically typed language *à la* ML, and we use that feature to obtain the full soundness of our separation analysis: the Why tool verifies *a posteriori*, for each C program, that its interpretation in our refined memory model is well-typed.

In Section 4, we show applications of the technique, and experimental results obtained with Caduceus. Two important benefits are presented: first, it is shown that our separation analysis provides significant improvements for programs with a large amount of data. Second, more unexpectedly, it is shown that separation can be a very useful ingredient for reasoning on advanced behaviors of programs.

We compare to related work in Section 5 and conclude in Section 6.

2 Preliminaries

2.1 Core language

Our analysis is described on a core language. Data types of this language are only integers and structures. Core expressions are made of constants, variables, standard operators, function calls, field accesses, pointer arithmetic. Grammar of expressions and statements are given on Figure 1, where τ denotes types which are either the **int** type for integers or pointers to structures. Other C constructs can be translated to these, as shown in next Section 2.2. We indeed have such a intermediate language in the implementation of Caduceus, with some others constructs like **switch**, **break**, **for**, etc. Only constructs given above are important for the rest of the paper.

2.2 Normalization of C source

We present briefly how we transform C code into our core language. It allows a fairly large subset of C, main unsupported features being pointer cast, unions, and pointers to function.

e	$::= c$	(integer constants)
	$\mid$ **null**	(null pointer)
	$\mid v$	(variables)
	$\mid e \rightarrow f$	(field access)
	$\mid id(e, \ldots, e)$	(function call)
	$\mid e\ op\ e$	(integer operators +,-,*,/,%, &&, etc.)
	$\mid e \oplus e$	(addition pointer+integer)
	$\mid e\ op\ e$	(pointer subtraction $\ominus$ and comparisons)
s	$::= v = e;$	(variable assignment)
	$\mid \tau\ v = e;$	(local variable introduction)
	$\mid e \rightarrow f = e;$	(field assignment)
	$\mid$ **return** $e;$	(function return)
	$\mid$ **if**$(e)\ s$ **else** s	(conditional branching)
	$\mid$ **while**$(e)\ s$	(while loop)

Fig. 1. Grammar of our core language

The two main points are first to remove the address operator &, and second to reduce the star operator $*p$, array accesses $t[i]$, and dot fields accesses $e.f$ to arrow field access $e \rightarrow f$. The address operator is removed by an initial phase: whenever a variable x is present as argument of &, it is transformed into a pointer (or more precisely an array of size 1) to the type of x in the original C code. Then each occurrence of $\&x$ becomes x and each occurrence of x becomes $*x$. The same is done for structure fields: if the address of a structure field f is taken somewhere as in $\&e.f$, then the type of f becomes a pointer (more precisely an array of size 1) to the type of f in the original C code. Each occurrence of $\&e.f$ becomes $e.f$ and $e.f$ becomes $e \rightarrow f$ (because it is equivalent to $(*e).f$. In a second phase, any expression e is normalized to $\overline{e}$ by reducing star operators, array accesses, and remaining dot field accesses by the following rules:

$$\overline{e.f} \mapsto \overline{e} \rightarrow f \qquad \overline{*e} \mapsto \overline{e} \rightarrow F(e) \qquad \overline{e_1[e_2]} \mapsto (\overline{e_1} \oplus \overline{e_2}) \rightarrow F(e_1)$$

where $F(e)$ is a field name generated from the type of e: for the `int` type we use `intM` (for "int memory"), for type `int*` we use `intPM` ("int pointer memory"), etc.

Figure 2 shows an example of normalization of C code.

2.3 Component-as-array modeling

The key idea proposed by Burstall [9] is to have one 'array' variable for each structure field: an applicative map which can be accessed or modified only via two side-effect free functions *select* and *store*. This modeling syntactically encode the fact that two structure fields cannot be aliased. The important consequence is that whenever one field is updated, only the corresponding array variable is modified and we have for free that any other field is left untouched.

Filliâtre and Marché proposed a variant of this technique to deal with C pointer arithmetic [8]. The C memory heap is represented by a finite set of array variables,

Original code:

```
int x;
int t[2];
struct S { int y;} s;

void f() {
  int *z = &x;
  t[1] = *z;
  s.y = t[1];
}
```

Normalized code:

```
struct intM { int intM; };
struct intM x[1];
struct intM t[2];
struct S { int y; };
struct S s[1];

void f() {
  (struct intM)* z = x;
  (t ⊕ 1)->intM = z->intM;
  s->y = (t ⊕ 1)->intM;
}
```

Fig. 2. Example of C code normalization

indexed by *pointers* viewed as pairs of an address to an allocated block and an offset into this block, thus 'array' variables are indeed 2-dimensional. For our core language, we interpret the $\oplus$ operator directly into a logical operation $shift$ in the model, so that we don't need anymore to make explicit this 2-dimensional representation.

The memory model is presented by a first-order axiomatization. Instead of an unsorted logic, we use polymorphic sorts: first it makes the presentation easier to follow, and second it will permit the soundness of the inference algorithm, in Section 3.3.

We introduce two logic sorts: `pointer` and α `memory` (denoting a 'memory array' containing values of type α), and operations

$$\begin{aligned} shift &: \texttt{pointer}, \texttt{integer} \rightarrow \texttt{pointer} \\ select &: \alpha\ \texttt{memory}, \texttt{pointer} \rightarrow \alpha \\ store &: \alpha\ \texttt{memory}, \texttt{pointer}, \alpha \rightarrow \alpha\ \texttt{memory} \end{aligned}$$

satisfying the implicitly universally quantified axioms

$$select(store(a, i, v), i) = v \tag{1}$$

$$select(store(a, i, v), j) = select(a, j) \text{ if } i \neq j \tag{2}$$

$$shift(p, 0) = p \tag{3}$$

$$shift(shift(p, i), j) = shift(p, i + j) \tag{4}$$

Our core language constructs are then interpreted into our intermediate language Why by transformation rules:

$$[e \rightarrow f] \mapsto select(f, [e]) \qquad [v = e] \mapsto v := [e]$$

$$[e_1 \oplus e_2] \mapsto shift([e_1], [e_2]) \qquad [e_1 \rightarrow f = e_2] \mapsto f := store(f, [e_1], [e_2])$$

Statements are interpreted into Why constructs, and memory accesses $e \rightarrow f$ are actually guarded with assertions to check validity of pointer dereferencing. This last aspect is not useful for the remaining, so we refer to [8, 13] for details.

```
struct S { int i; };

/*@ requires \valid(x) && \valid(y)
  @ assigns x->i, y->i
  @ ensures x->i == 1 && y->i == 2
  @*/
void f(struct S *x, struct S *y) {
  x->i = 1; y->i = 2;
}

struct S t1[1], t2[1];

//@ ensures t1[0].i == 1 && t2[0].i == 2
void g() { f(&t1[0],&t2[0]); }
```

Fig. 3. Simple case of separation analysis

3 Separation Analysis

3.1 Modeling with regions

We illustrate on simple examples how we integrate a separation analysis into the modeling presented before. Consider the C source code of Figure 3, where we use the syntax of the Caduceus JML-like specification language: annotations are given in special comments `/*@ .. */`, **requires** introduces a pre-condition, **ensures** a post-condition, and **assigns** is a clause to specify which memory locations are modified [8, 13]. The annotation `\valid(x)` means that `x` points to a safely allocated memory location.

Post-condition of function `f` cannot be established: it is indeed wrong in case of aliasing: pointers `x` and `y` may be equal. For the call to `f` in function `g` they are different, but since we use a modular reasoning (function by function, as for any technique based on WP), the whole code cannot be proved correct. A possible solution could be to add to the pre-condition of `f` the additional hypothesis `x != y`, but our goal in this paper is to avoid this extra condition. The Why interpretation of the C code of `f` is

$$i := store(i, x, 1); \qquad i := store(i, y, 2);$$

so establishing the post-condition $select(i, x) = 1$ amounts to prove the corresponding weakest precondition

$$select(store(store(i, x, 1), y, 2), x) = 1$$

which is a consequence of axioms (1) and (2) if $x \neq y$. Our goal is to interpret the code of `f` differently using two distinct variables for representing the fields of `x` and `y`:

$$i_x := store(i_x, x, 1); \qquad i_y := store(i_y, y, 2);$$

i_x and i_y are two distinct *regions* for field `i`. With that interpretation, post-conditions $select(i_x, x) = 1$ and $select(i_y, y) = 2$ follows from axiom (1) without need of $x \neq y$.

```
struct T {
  struct S t1[2];
  struct S t2[2];
};

/*@ ensures s.t1[0].i == 1 && s.t2[0].i == 2 &&
  @         s.t1[1].i == 2 && s.t2[1].i == 1
  @*/
void h(struct T s) {
  f(&s.t1[0],&s.t2[0]);
  f(&s.t2[1],&s.t1[1]);
}
```

Fig. 4. Case of parametric regions

Consider additionally the code of Figure 4. Function `f` is now called twice, in first call `x` points to array `s.t1` and `y` points to array `s.t2`, and it is reversed in the second call. To allow separation of `s.t1` and `s.t2` into different regions, we need to make these regions *parameters* of `f`, and we call them *parametric regions*. The complete interpretations of `f` and `h` are then

$$
\begin{array}{l}
\texttt{void}\ f(i_x, i_y, x, y)\ \{ \\
\quad i_x := store(i_x, x, 1); \quad i_y := store(i_y, y, 2); \\
\} \\
\texttt{void}\ h(t_1, t_2, i_1, i_2, s)\ \{ \\
\quad f(i_1, i_2, select(t_1, s), select(t_2, s)); \\
\quad f(i_2, i_1, shift(select(t_2, s), 1), shift(select(t_1, s), 1)); \\
\}
\end{array}
$$

and their post-conditions can be established by simple first-order reasoning.

So our goal is to integrate a notion of separation into the modeling of C code, by attaching regions to pointers and memory variables. The interpretation of a memory access $e \rightarrow f$ is now $select(f\_r, e)$ where r is the region of e. We now see how we compute those regions.

3.2 Regions as types

We see regions as a rich type system for pointers. For simplicity, we only consider pointers and the `int` base type. The types of expressions are given by the grammars

(types)	$\tau ::=$	`int`	
		$\mid r$ `pointer`	(pointer to region r)
		$\mid (\tau, r)$ `memory`	(memory of values of type τ in region r)
(regions)	$r ::= \rho$		(region variable)
		$\mid R$	(region constant)

Region variables are needed for the parametric regions passed to functions: regions as function parameters have a *polymorphic* type. Our type system is then just a particular case of a polymorphic type system *à la* Milner [11].

If we consider the function f of example above, its profile is

$$f(i_x : (int, \rho_1)\ \texttt{memory}, i_y : (int, \rho_2)\ \texttt{memory}, x : \rho_1\,\texttt{pointer}, y : \rho_2\,\texttt{pointer})$$

that is polymorphic in ρ_1, ρ_2: for each call to f these regions can be instantiated differently.

Region typing rules We now express separation by giving typing rules for expressions, using types with regions. The typing environment is made of two parts denoted Γ and Δ. Γ is a classical typing environment which maps variable identifiers to types: we denote $x : t \in \Gamma$ whenever Γ maps variable x to the type t. Δ is a *region environment* which maps each pair (r, f), where r is a region and f is a field identifier, to the type t of $p \to f$ when p is a pointer in region r. We denote that as $(r, f) : t \in \Delta$.

We are now able to give typing rules for expressions. For constants we have

$$\frac{}{\Gamma, \Delta \vdash n : \texttt{int}} \qquad \frac{}{\Gamma, \Delta \vdash \textbf{null} : r\,\texttt{pointer}}$$

for any region r (region polymorphism for **null**).

Type of a variable follows the environment Γ and type of a field access follows the environment Δ:

$$\frac{}{\Gamma, \Delta \vdash x : t} \text{ if } x : t \in \Gamma \qquad \frac{\Gamma, \Delta \vdash l : r\,\texttt{pointer}}{\Gamma, \Delta \vdash l \to f : t} \text{ if } (r, f) : t \in \Delta$$

Function calls use polymorphic typing:

$$\frac{\Gamma, \Delta \vdash e_1 : t_1 \quad \cdots \quad \Gamma, \Delta \vdash e_n : t_n}{\Gamma, \Delta \vdash id(e_1, \ldots, e_n) : t}$$

if $id : (\tau_1, \cdots, \tau_n) \to \tau \in \Gamma$ and there is a region substitution σ such that $t = \tau\sigma$ and for each i, $t_i = \tau_i\ \sigma$.

Pointer shift keeps the same region:

$$\frac{\Gamma, \Delta \vdash e_1 : r\,\texttt{pointer} \quad \Gamma, \Delta \vdash e_2 : \texttt{int}}{\Gamma, \Delta \vdash e_1 \oplus e_2 : r\,\texttt{pointer}}$$

Difference and comparison of pointers only allows pointers in the same region:

$$\frac{\Gamma, \Delta \vdash e_1 : r\,\texttt{pointer} \quad \Gamma, \Delta \vdash e_2 : r\,\texttt{pointer}}{\Gamma, \Delta \vdash e_1\ op\ e_2 : int}$$

where $op \in \{\ominus, ==, <=, =>, <, >, !=\}$.

The typing rules for statements are then the following. For variable assignment:

$$\frac{\Gamma, \Delta \vdash e : t}{\Gamma, \Delta \vdash v = e : t} \text{ if } x : t \in \Gamma$$

For field assignment:

$$\frac{\Gamma, \Delta \vdash e_1 : r\,\texttt{pointer} \quad \Gamma, \Delta \vdash e_2 : t}{\Gamma, \Delta \vdash e_1 \to f = e_2 : t} \text{ if } (r, f) : t \in \Delta$$

Typing of other statements is done in a natural way.

For the typing of local or global declarations, we assume for the moment an oracle which gives the regions involved in the construction of Γ. For declarations of structures, this means that the Δ environment is also given. In other words, typing of functions is made in a given Δ.

The first result we give is a soundness property of the typing rules: this soundness is relative in the sense that it shows that the interpretations of programs is the same with or without the separation of memory variables.

Theorem 1 (Relative soundness). *If a program is well typed in a given environment Γ, Δ, then its logical interpretation with region memory variables has the same semantics as its interpretation with the classical component-as-array model.*

Proof sketch: we provide a bisimulation of execution steps of interpreted programs, with or without separation. A state of the program with regions can be seen as a partition of the state of the program without regions. Each operation on the program with regions can be simulated on the state of the program without region and vice-versa: this works because all operations respect the partition, because the programs is well-typed in term of regions.

3.3 Inference of regions

The remaining step is now to provide an inference system, to construct an environment Γ, Δ which makes a given program well-typed, if possible. Since our type system is a particular case of a polymorphic type system à la Milner, we can derive an inference method from known type inference algorithms such as the W algorithm [14]. The only specific feature is the handling of the Δ part of the environment.

In a first step, we assign a fresh region constant to each global pointer variable. Parameters of functions, and local variables, which are pointers, are given a fresh region variable. This provide an initial Γ. We build at the same time an initial Δ which makes everything separated *a priori*.

In a second step, functions are analyzed, in the order given by the call graph, to determine their polymorphic type. For each function, the code is traversed, and typing rules given above lead to equality constraints between regions, that is we perform unification of regions. Each time a function is analysed, we determine which of the region variables remain not instantiated, and we make the function type polymorphic by quantifying over them.

Unification of regions is standard, expect for the handling of Δ: each time two regions r_1 and r_2 are made identical, we need to perform a merge operation on Δ: for each field f such that Δ maps (r_1, f) to t_1 and (r_2, f) to t_2, we need to merge the mappings, and consequently unify the type t_1 and t_2, which may recursively lead to unification of other regions. During this unification phase, Γ is modified by side-effect.

At the end of this process, we end up with a Γ and a Δ in which the program is well-typed. As said in introduction, we do not provide a proof of this fact. This is a classical inference algorithm tough, only the unification of regions is original, and we are pretty sure it is indeed correct, but anyway in practice the soundness is obtained in

a secure way: we generate a typed Why program, and if the inferred types are wrong then Why typing will fail.

Final remark: the W algorithm is known to compute the *principal type*. In terms of regions, this means that it computes the separation into the largest possible number of regions, allowed by the typing rules given.

4 Applications

Our separation analysis is implemented in the Caduceus tool, as a user option. Selection of this option asks to perform the inference of regions, and then the generation of the model and the Why interpretation of C code is modified accordingly. We show here a few experiments and applications.

First of all, we tried using separation analysis on the set of small C programs that are used as a non-regression test in Caduceus. Results shows an improvement, but only a small one, which is actually explainable because those C codes are small. But more importantly, this means that this does not bring overhead on examples where separation is not the concern. Remark also that the separation analysis itself is quick (we believe it is linear in the size of the code), so probably separation analysis could be turned on by default in the future.

4.1 Regions and logical annotations

This example is inspired from a piece of Java code by P. Müller [16]. It computes the set of positive elements of an array, and puts them in a new array.

```
int *m(int t[], int length) {
  int count = 0; int i; int *u;
  for (i=0; i < length; i++) if (t[i] > 0) count++;
  u = (int*)calloc(count,sizeof(int));
  count = 0;
  for (i=0; i < length; i++) if (t[i] > 0) u[count++] = t[i];
  return u;
}
```

Verification that the assignment of `u[count]` is inside the array bounds is tricky: it involves a "semantic" reasoning, noticing that the second loop counts exactly the same number of elements as the first, so the index `count` must be smaller than the value of `count` used for allocating the array `u`. To make this reasoning explicit, it is natural to annotate the loops with an invariant, the same one for both loops:

```
//@ invariant
//@   count == \num_of(int j; 0 <= j && j < i ; t[j] > 0)
for (i=0 ; i < length; i++) ...
```

where `\num_of` is a JML-like construct [17] giving the number of elements satisfying the predicate given as argument. Indeed, original example by Müller was precisely a

challenge for static verification tools because none of them supports the `\num_of` construct. Anyway, it is possible on a given example to 'expand' the use of `\num_of`, and we did that for our C code: we introduce a *logic function* [8]:

```
logic int num_of_pos(int i,int j,int a[]) reads t[..]
```

whose intended meaning is to give the number of positive elements in array `a` between indexes `i` and `j`, included. This meaning is formalized by introducing a few *axioms*:

```
axiom num_of_pos_empty :
  \forall int i, int j, int a[];
      i > j => num_of_pos(i,j,a) == 0
axiom num_of_pos_true_case :
  \forall int i, int j, int k, int a[];
      i <= j && a[j] > 0 =>
        num_of_pos(i,j,a) == num_of_pos(i,j-1,a) + 1
axiom num_of_pos_false_case :
  \forall int i, int j, int k, int a[];
      i <= j && ! (a[j] > 0) =>
        num_of_pos(i,j,a) == num_of_pos(i,j-1,a)
```

(see `http://www.lri.fr/~marche/MullerChallenge.pdf` for the remaining annotations).

The key point now is that the verification of safety cannot be done, because in the second loop, we know that `count` is less than the number of positive elements in `t`, but there is a reasoning to perform to establish than this number of elements *did not change between the two loops*. With a single heap variable `intM` in the model to represent integer arrays, this is far from simple. Indeed, the logic function `num_of_pos` is axiomatized with some inductive scheme, and one should prove that it implies that `num_of_pos(i,j,a)` only depends on the values of `a[i..j]`, which is hard.

On the other hand, with our modeling involving memory separation, it is statically detected that `t` and `u` are separated, and thus can be modeled with two separate heap variables `intM_t` and `intM_u`. Then, the logic function `num_of_pos` becomes *parametric* in the memory variable involved for the array argument `a`, and it becomes syntactically true that `num_of_pos(0,j-1,t)` is the same in both loop of our piece of C code. Notice also that in the Why interpretation of the axioms above, an extra quantification over the memory variable involved is added.

On that example, each verification condition is then discharged automatically by the Simplify prover.

4.2 An industrial case study

In collaboration with Dassault Aviation company, we experimented Caduceus and its separation analysis on a real embedded code for avionics. The first experiment was made on a core of this code, which is approximately 3000 lines long. The characteristics is that it contains a large number of data structures, and usually these structures contains nested arrays of other structures.

Without separation analysis, this code gives rise to 1151 VCs, 965 being discharged by Simplify, that is 83.8%. With separation, we get 1982 VCs, 1972 discharged by Simplify, that is 99.4 %. There are a total of 376 regions inferred for global variables, and there are 242 polymorphic regions added as parameter to functions. The significantly higher number of VCs with separation analysis can be explain by the high number of regions: the interpretation of **assigns** clauses produces as many propositions as the number of memory variables involved, which is larger when separation is turned on. So this may make the number of VCs larger, but each of them are simpler. Notice finally that the 10 remaining VCs have been discharged by the interactive proof assistant Coq.

We believe this is a very positive experimental result, which shows that our separation analysis is a major improvement in practice. We are currently experimenting on the whole code (70000 lines long) with good results too. We are also trying to prove a complex behavioral property, involving logical annotations and ghost variables, for which we hope that the separation analysis will greatly simplify the reasoning, as on the previous example.

5 Related work

Talpin and Jouvelot proposed in 1994 [18] a calculus for analyzing effects of programs based of a polymorphic type system. The principle is the same as ours, but their work is limited to reference variables: no deep sharing in data structures is possible.

In the context of static analysis, points-to analysis is a very advanced technique for computing information on pointers. This has been initiated by Andersen in 1994 [19] and extended further in 1996 by Steensgaard [20] and in 2000 by Das [21]. We used their idea of designing a type system for analysis separation, but it is clear that our analysis is much less precise than theirs. Our method is tailored to the generation of a refined component-as-array model for deductive verification.

The Cyclone system [22] proposes a new programming language analogous to C, but in which the programmer can specifying regions manually. Similarly to us, they allow parametric regions in function call. But first our setting applies to the real C language, and second are regions are *automatically inferred* instead of being given by the user.

Compared to Separation Logic, our separation analysis is clearly less precise. It seems that Separation Logic is very powerful in some cases: it allows for example to specify that a linked list cannot be circular, or that a graph is a tree, and to reason with that. This is something that our analysis cannot do: as soon as one traverses a linked lists, only one region is inferred for the whole list. Our analysis is clearly more adapted to deal with programs with a high size of data, but it also brings improvement when reasoning on small programs, as shown in Section 4.1. Combining all the power of Separation Logic and our Separation analysis remains a future task.

In 2006, Nanevski, Morrisett and Birkedal [23] showed the importance of parametricity of regions in their *Hoare type theory* framework. We have shown the same in the classical context of deductive verification based on weakest precondition calculus, which is directly applicable to several existing practical tools.

Generally speaking, we think our approach allowed us to address much more complex applications than previous works, such as the industrial case study of Section 4.2

6 Conclusion

We proposed a separation analysis that is potentially useful for any tool for deductive verification based on weakest precondition calculus and a component-as-array model. Our experimentations on C programs are very positive, as shown by advanced applications.

There are some drawbacks that we plan to address in the future. First, the separation analysis must be done on the whole program, so it is not modular. This is not a major problem since the separation analysis is quick, and because after separation analysis is performed, the remaining of the verification task can still be done modularly, function by functions. But in case we do not have the complete program available, such as if we what to verify libraries, this is a problem. We plan to add new constructs in the specification language to allow user specification of separation: for example, for a library function such as `memcpy`, one may want to specify that the source and target are separated.

Our separation analysis is tailored to its later use for the component-as-array model. In that model, a given C array will be entirely in the same region. However, by advanced static analysis, it is possible to discover that an given array may be splited into several regions, for example in the following code:

```
struct S  int x ;
struct S t[10];
void f(S *p, int n) { p->x = n; }
void main()  f(t+0,2); f(t+1,3);
```

we do not get for free that `t[0]->x == 2`. We are planning to incorporate more advanced static analysis techniques in our setting, in the context of the CAT project (`http://www.rntl.org/projet/resume2005/cat.htm`).

Acknowledgments We gratefully thank J.-C. Filliâtre, Ch. Paulin, Y. Moy for their comments about this work and this paper.

References

1. Floyd, R.W.: Assigning meanings to programs. In Schwartz, J.T., ed.: Mathematical Aspects of Computer Science. Volume 19 of Proceedings of Symposia in Applied Mathematics., Providence, Rhode Island, American Mathematical Society (1967) 19–32
2. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (1969) 576–580 and 583
3. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 17h Annual IEEE Symposium on Logic in Computer Science, IEEE Comp. Soc. Press (2002)
4. Dijkstra, E.W.: A discipline of programming. Series in Automatic Computation. Prentice Hall Int. (1976)

5. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Technical Report 159, Compaq Systems Research Center (1998) See also `http://research.compaq.com/SRC/esc/`.
6. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (2004)
7. Leino, K.R.M.: Efficient weakest preconditions. Technical Report MSR-TR-2004-34, Microsoft Research (2004)
8. Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In Davies, J., Schulte, W., Barnett, M., eds.: Sixth International Conference on Formal Engineering Methods. Volume 3308 of Lecture Notes in Computer Science., Seattle, WA, USA, Springer-Verlag (2004) 15–29
9. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. Machine Intelligence **7** (1972) 23–50
10. Bornat, R.: Proving pointer programs in Hoare logic. In: Mathematics of Program Construction. (2000) 102–126
11. Milner, R.: A theory of type polymorphismn programming. Journal of Computer and System Sciences **17** (1978)
12. Filliâtre, J.C.: Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud (2003) `http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz`.
13. Marché, C., Paulin-Mohring, C.: Reasoning about Java programs with aliasing and frame conditions. In Hurd, J., Melham, T., eds.: 18th International Conference on Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, Springer-Verlag (2005)
14. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (1982) 207–212
15. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3) (2005) 365–473
16. Müller, P.: Specification and verification challenges. Exploratory Workshop: Challenges in Java Program Verification, Nijmegen, The Netherlands (2006) `http://www.cs.ru.nl/~woj/esfws06/slides/Peter.pdf`.
17. Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., Jacobs, B.: JML: notations and tools supporting detailed design in Java. In: OOPSLA 2000 Companion, Minneapolis, Minnesota. (2000) 105–106
18. Talpin, J.P., Jouvelot, P.: Polymorphic type, region and effect inference. Journal of Functional Programming **2**(3) (1992) 245–271
19. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen (1994)
20. Steensgaard, B.: Points-to analysis in almost linear time. In: Symposium on Principles of Programming Languages. (1996) 32–41
21. Das, M.: Unification-based pointer analysis with directional assignments. In: PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, New York, NY, USA, ACM Press (2000) 35–46
22. Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-based memory management in cyclone. In: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, New York, NY, USA, ACM Press (2002) 282–293
23. Nanevski, A., Morrisett, G., Birkedal, L.: Polymorphism and separation in Hoare type theory. In Reppy, J.H., Lawall, J.L., eds.: 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, ACM (2006) 62–73