Multi-Prover Verification of C Programs

Jean-Christophe Filliâtre and Claude Marché

PCRI — LRI (CNRS UMR 8623) — INRIA Futurs — Université Paris 11
Bât 490, Université Paris-sud, 91405 Orsay cedex, France

Abstract. Our goal is the verification of C programs at the source code level using formal proof tools. Programs are specified using annotations such as pre- and postconditions and global invariants. An original approach is presented which allows to formally prove that a function implementation satisfies its specification and is free of null pointer dereferencing and out-of-bounds array access. The method is not bound to a particular back-end theorem prover. A significant part of the ANSI C language is supported, including pointer arithmetic and possible pointer aliasing. We describe a prototype tool and give some experimental results.
Keywords: C programming language, Hoare logic, pointer programs, formal verification and proof.

1 Introduction

Using formal methods for verifying properties of programs at their source code level has gained more interest with the increased use of embedded programs, which are short programs where a high-level of confidence is required. Such embedded programs are no more written in assembly language but in C (plane command control, cars, etc.) or in JavaCard [21] (mobile phones, smart cards, etc.). This paper introduces a verification method for ANSI C source code, following the same approach as the one we used for Java [15, 16]. A prototype tool has been implemented, called Caduceus, freely available for experimentation [11]. This tool is currently under experimentation at Axalto (smart cards) and Dassault Aviation (aeronautics).

In the following, we assume the reader fairly familiar with ANSI C [13]. In our setting, the properties that may be checked are of two kinds: first, the program may be checked to be free of *threats* (null pointer dereferencing or out-of-bounds array access) and second, it may be proved satisfying functional properties given as functions's *postconditions*. These kinds of properties are not independent since they both usually need insertion of appropriate annotations (functions's *preconditions*, *global invariants*, *loop invariants*, etc.) as usual in a Hoare logic framework. In practice, these annotations are inserted in the source code as comments of a specific shape `/*@...*/`. The specification language we use in those comments is largely inspired by the Java Modeling Language (JML) [14]. It has however significant differences, mainly due to the fact that unlike JML we do not seek runtime assertion checking.

[0] This research was partly supported by the "projet GECCOO de l'ACI Sécurité Informatique", CNRS & INRIA, `http://gecco.lri.fr`

Once a C program is annotated, verification is done by running Caduceus on the sources in a way similar to a classical compiler but resulting in the generation of the so-called *verification conditions*: these are first-order predicate logic formulas whose validity implies the soundness of the program with respect to the absence of threat and to the functional properties given as annotations. At this point, a general purpose theorem prover must be used to establish those verification conditions. An original Caduceus feature is its independence with respect to the prover. It currently supports the Coq interactive proof assistant [7] and the Simplify automatic prover [1]. New provers may easily be added provided a suitable pretty-printer; support for PVS [18] and haRVey [19] are planned in a near future.

A significant part of ANSI C is supported. All control structures are supported except `goto`. Pointer programs are supported, including pointer arithmetics and possible pointer aliasing, using old ideas from Burstall [4] recently emphasized by Bornat [3] and used for a case study by Mehta and Nipkow [17]. The main unsupported feature is the cast of pointer (e.g. when pointers to `void` are used to write "polymorphic" functions).

In this paper we present both an overview of Caduceus from a user's point of view and the method used internally to generate the verification conditions. This method is very similar to the one we used for Java programs and a significant amount of work is shared: the generation of the conditions is indeed realized by a common intermediate tool called Why [9, 10] whose input language—also annotated with Hoare-like pre- and postconditions—is specifically designed for the purpose of verification conditions generation. In other words, Caduceus is a compiler from annotated C programs to the Why input language.

We seek for a method which guarantees soundness with the highest level of confidence, even if it is not fully automatic. In other words, our approach differs from static checking methods, which are always fully automated but may sometimes miss bugs.

This paper is organized as follows. Section 2 gives an overview of the use of Caduceus with a presentation of the specification language, examples and experimental results. In Section 3, we describe the internal approach used to perform verification, including the modeling of heap memory, the translation to Why and discussion about soundness. We conclude in Section 4 with a comparison to related works.

2 Overview of Caduceus and its specification language

This section gives an overview of Caduceus from a user's point of view based on small examples. A comprehensive description of the annotation syntax can be found in the Caduceus reference manual [11].

2.1 A simple character queue

We start with a very simple C program implementing a character queue within a circular array. The queue is a global structure `q` declared on Figure 1. The array `q.contents` contains the queue elements stored from index `q.first` included to index `q.last` excluded. `q.length` is the length of `q.contents`, that is the maximal capacity of

```
struct queue {
  char *contents;
  int length;
  int first, last;
  unsigned int empty, full :1;
} q;
```

Fig. 1. A character queue within a circular array

the queue. The boolean fields `q.empty` and `q.full` indicate whether the queue is empty or full respectively.

It is natural to associate an invariant to such a global declaration. It avoids repeating the same properties in all functions's specifications. For instance, we may declare that `q.contents` points to a valid (i.e. allocated) array of at least `q.length` elements and that `q.first` and `q.last` are valid indexes within this array. This is done using the global declaration `invariant` as follows:

```
/*@ invariant q_invariant :
  @   \valid_range(q.contents, 0, q.length-1) &&
  @   0 <= q.first < q.length &&
  @   0 <= q.last < q.length
  @*/
```

Informally, such an invariant is added to the precondition of each function involving `q` and to the postcondition of each function possibly modifying `q`. The precise meaning of "involving" and "possibly modifying" is given below.

Annotations such as the above invariant are first-order logic predicates built from atoms and connectives such as `&&`, `||`, `!`, etc. Atoms are built from side-effects free C expressions and predicate symbols. Here, `\valid_range` is a predefined predicate expressing that a given array range can be safely dereferenced.

A function specification is given right before the function definition and is composed of three (optional) clauses: a precondition introduced by the keyword `requires`, a side-effects clause introduced by the keyword `assigns` and a postcondition introduced by the keyword `ensures`. For instance, here is a possible specification/definition of a function `push` adding an element in `q`:

```
/*@ requires !q.full
  @ assigns  q.empty, q.full, q.last, q.contents[q.last]
  @ ensures  !q.empty && q.contents[\old(q.last)] == c
  @*/
void push(char c) {
  q.contents[q.last++] = c;
  if (q.last == q.length) q.last = 0;
  q.empty = 0;
  q.full = (q.first == q.last);
}
```

The `requires` clause is a predicate that must hold whenever the function is called; here it states that `q` must not be full.

The `assigns` clause indicates the set of locations possibly modified by the function call. More precisely, it says that any memory location already allocated before the function call and not mentioned in this set is left untouched by the function call.

The `ensures` clause is a predicate that must hold whenever the function returns. Values of expressions or predicates in the prestate—the program point corresponding to the function call—can be obtained using the `\old()` modifier. For instance we write here `\old(q.last)` to denote the value of `q.last` when `push` was called.

When run on the above code, Caduceus produces several verification conditions expressing that (1) the postcondition and the `assigns` clause of `push` are established, (2) the invariant `q_invariant` is preserved by `push`, and (3) the code of `push` contains no null pointer dereference (which means here that the assignment of `q.contents[q.last++]` is valid).

To establish the validity of these conditions, a theory (i.e. a set of axioms) is provided (see Section 3.4). Using the Simplify decision procedure with this theory, the three conditions for `push` are automatically discharged. With the Coq proof assistant, proofs are easily handled (6 lines of tactics to write).

A function specification can also be attached to a function declaration, this function being defined later or not. For instance we can declare a `pop` function together with its specification:

```
/*@ requires !q.empty
  @ assigns q.empty, q.full, q.first
  @ ensures !q.full && \result == q.contents[\old(q.first)]
  @*/
char pop();
```

Notice, in the postcondition, the use of `\result` to denote the returned value.

It is important to notice that our verification process is modular, in the sense that when another function calls `push` or `pop`, only their specifications are seen. In other words the implementation of `push` may be changed or the one of `pop` added without affecting the remaining of the verification process. Notice also that the `assigns` clauses are essential for modularity of the verification, not only because they specify what is modified, but mainly because they specify that everything else is unmodified (see Section 3.4).

2.2 In-place list reversal

We continue with a now classical example: in-place list reversal [3, 4, 17]. The code is given Figure 2. It introduces a type `list` for linked list of integers and a function `reverse` that reverses a list in-place.

To write a specification for `reverse` we first need to express that the list `p0` is finite. This cannot be expressed using a first-order predicate (assuming no primitive). A solution could have been to allow the use of side-effect free functions from the code itself, like in JML for instance. This would clearly make sense for runtime assertion checking, but for formal verification we think it is better to clearly separate the code side

```
typedef struct struct_list {
  int hd;
  struct struct_list *tl;
} *list;
```

```
list reverse(list p0) {
  list r = p0, p = NULL;
  while (r != NULL) {
    list q = r;
    r = r->tl; q->tl = p; p = q;
  }
  return p;
}
```

Fig. 2. In-place list reversal

and the logic side. So instead, in our setting predicates and logical functions required to write specifications must be defined on the logic side, that is in the language of the backend theorem prover, and imported into the source code.

For this example, we import a predicate `is_list`, supposed to characterize finite lists, with the following declaration:

```
/*@ predicate is_list(list l) reads l->tl */
```

The clause `reads` indicates the memory locations involved in the predicate definition. This is indeed mandatory for the predicate to be applied to the corresponding variables on the logic side. On this example, the definition of `is_list` obviously requires the knowledge of the `tl` fields of lists cells.

We need additional predicates to give `reverse` a postcondition. Indeed we must relate the returned list to the input list. Following [17] we introduce a predicate `llist` relating a C list `p` (a pointer) to the list of all its elements:

```
/*@ predicate llist(list p, plist l) reads l->tl */
```

Here `plist` is an abstract type of pointer lists defined on the logic side. With the additional declaration of a reverse function on type `plist`

```
/*@ logic plist rev(plist pl) */
```

it is then possible to write a specification for `reverse`:

```
/*@ requires is_list(p0)
  @ ensures \forall plist l0;
  @    \old(llist(p0, l0)) => llist(\result, rev(l0))  @*/
list reverse(list p0) { ...
```

It is important to note that the function symbol `rev` involved in the specification is not at all related to the C function `reverse` being defined.

As usual with Hoare-like annotations, we need to annotate the loop to be able to prove function `reverse` correct. We insert a loop invariant and a variant to ensure the termination, respectively introduced by the keywords `invariant` and `variant`:

```
...
/*@ invariant
```

```
     \exists plist lp; \exists plist lr;
       llist(p, lp) && llist(r, lr) && disjoint(lp, lr) &&
     \forall plist l;
       \old(llist(p0, l)) => app(rev(lr), lp) == rev(l)
  @ variant length(r) for length_order */
while (r != NULL) { ...
```

The invariant involves new importations:

```
/* app(l1,l2) is the concatenation of l1 and l2 */
/*@ logic plist app(plist l1, plist l2) */
/* disjoint(l1,l2) is true whenever l1 and l2 have no
   element in common */
/*@ predicate disjoint(plist l1, plist l2) */
```

and the variant also requires importations of `length` and `length_order` (see [11] for details).

With these annotations, Caduceus generates 7 verification conditions for the C function `reverse`. The Simplify prover only established 4 of them automatically, which is not surprising because indeed we did not provide a full theory for the new predicates such as `llist`. On the other hand, we defined in Coq a model for these predicates using the predefined Coq lists and all conditions have been proved correct. The proofs amount to 70 lines of tactics. The model for linked lists (definition of `llist`, etc.) is around 180 lines long and is reusable for other programs involving linked lists.

We are currently experimenting Caduceus on the Schorr-Waite algorithm which is considered as a benchmark for pointer programs verification [3, 17]. Compared to the proof in Isabelle/HOL [17], we also check the absence of threats. Since annotations from [17] fit in our specification language, we strongly believe that Isabelle/HOL proofs will be adapted to Coq without difficulty.

3 The verification technique

Our approach is to translate annotated C programs into Why's input language. The latter does not provide any data structure, but only applicative types (primitive types such as integers, booleans, etc. and *abstract* datatypes) and mutable variables containing applicative values. This means that Why has absolutely no knowledge of arrays, structures, pointers and memory heap in general, hence our translation mechanism needs to introduce definitions of abstract datatypes to model the memory state of a given C program. Then the semantics of C statements must be translated into Why statements operating on these abstract datatypes. In this section we proceed as follows: first we present Why's input language, then we describe our modeling of memory heaps, then we give the translation rules from C statements to Why's statements, then the background theory and finally we discuss about the soundness of the approach.

3.1 The Why tool

Unlike usual approaches in Hoare logic, Why's core language is not Pascal-like but ML-like: there is no difference between expressions and statements (the latter are simply expressions of type `unit`, which is similar to `void` in C) and immutable variables

$$
\begin{array}{l}
\textbf{parameter}\ \texttt{push\_parameter}: c : \texttt{int} \rightarrow \\
\quad \{\ \texttt{acc}(\mathit{full}, q) = 0 \wedge \texttt{valid}(\mathit{alloc}, q) \wedge \\
\qquad \texttt{valid\_range}(\mathit{alloc}, \texttt{acc}(\mathit{contents}, q), 0, \texttt{acc}(\mathit{length}, q) - 1) \wedge \\
\qquad 0 \leq \texttt{acc}(\mathit{first}, q) < \texttt{acc}(\mathit{length}, q) \wedge 0 \leq \texttt{acc}(\mathit{last}, q) < \texttt{acc}(\mathit{length}, q)\ \} \\
\quad \texttt{unit}\ \textbf{reads}\ \mathit{alloc}, \mathit{contents}, \mathit{empty}, \mathit{first}, \mathit{full}, \mathit{intP}, \mathit{last}, \mathit{length}, q \\
\quad \textbf{writes}\ \mathit{empty}, \mathit{full}, \mathit{intP}, \mathit{last} \\
\quad \{\ \texttt{acc}(\mathit{empty}, q) = 0 \wedge \texttt{acc}(\mathit{intP}, \texttt{shift}(\texttt{acc}(\mathit{contents}, q), \texttt{acc}(\mathit{last}@, q@))) = c \wedge \\
\qquad \texttt{assigns}(\mathit{alloc}@, \mathit{last}@, \mathit{last}, \texttt{pointer\_loc}(q@)) \wedge \\
\qquad \texttt{assigns}(\mathit{alloc}@, \mathit{intP}@, \mathit{intP}, \\
\qquad\qquad \texttt{pointer\_loc}(\texttt{shift}(\texttt{acc}(\mathit{contents}@, q@), \texttt{acc}(\mathit{last}@, q@)))) \wedge \\
\qquad \texttt{assigns}(\mathit{alloc}@, \mathit{full}@, \mathit{full}, \texttt{pointer\_loc}(q@)) \wedge \\
\qquad \texttt{assigns}(\mathit{alloc}@, \mathit{empty}@, \mathit{empty}, \texttt{pointer\_loc}(q@)) \wedge \\
\qquad \texttt{valid\_range}(\mathit{alloc}, \texttt{acc}(\mathit{contents}, q), 0, \texttt{acc}(\mathit{length}, q) - 1) \wedge \\
\qquad 0 \leq \texttt{acc}(\mathit{first}, q) < \texttt{acc}(\mathit{length}, q) \wedge 0 \leq \texttt{acc}(\mathit{last}, q) < \texttt{acc}(\mathit{length}, q)\ \}
\end{array}
$$

Fig. 3. Why interpretation of `push`

are introduced by `let..in..` constructs. On top of this side-effect free core language, modifiable variables are introduced using an explicit reference type `ref` as in SML or OCaml. A feature of great importance for us is that Why rejects any variable aliasing (rejecting aliasing is required by Why to guarantee the soundness of verification condition generation), hence dealing with pointer programs requires an appropriate modeling of memory. Finally, beside usual imperative control statements (sequence, conditional and `while` loops) Why features exceptions with a try-catch mechanism. Exceptions will be used to encode `return`, `break` and `continue` C statements.

Why programs can be annotated using pre- and postconditions, loop invariants and intermediate assertions. Logical formulas are written in a (possibly polymorphic) typed first-order logic with built-in equality, booleans, infinite precision integers and real numbers with usual operations. Why performs an *effect analysis* (see below) on modifiable variables of programs, a classical weakest precondition computation [8] and generates proof obligations for various provers.

Why has a modular approach. New logic functions or predicate symbols can be introduced and freely used in the specification part. Sub-programs can also be introduced abstractly, by the mean of a full specification: type of input variables and result, pre- and postconditions, as well as its *effects*, that is the sets of global references possibly read and/or written by the subprogram.

In order to translate an annotated C program into Why, one needs to proceed the following way:

1. Find an appropriate modeling of C memory states using global Why variables which will never be aliased;
2. Translate C constructs into Why statements, with assignments over the global variables introduced above;
3. Translate annotations into Why predicates over those variables.

Because C functions can be mutually recursive, we first give a specification for all the functions as abstract Why subprograms with specifications. This abstract view is used to interpret function calls and suffices to ensure partial correctness of programs.

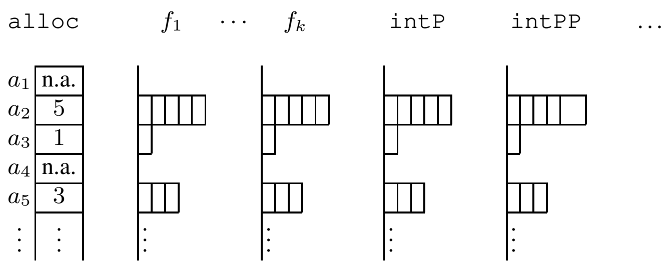


Fig. 4. Modeling of C memory heap

Figure 3 shows the Why abstract declaration of the subprogram corresponding to the `push` function of Section 2.1. It has one argument (the integer c) which is supposed to satisfy the precondition (first formula between curly braces). There is no result (output type is `unit`), it accesses nine global variables (listed after **reads**) and writes four (listed after **writes**). The postcondition (second pair of curly braces) uses the notation $x@$ to denote the value of variable x before the function application. The predicates `valid`, `valid_range` and `assigns`, and function symbols `acc`, `shift` and `pointer_loc` are easily understood as translation of specifications; they are formally introduced in the next sections.

3.2 Modeling C memory heap

The C memory is represented by a finite set of Why variables, each containing an applicative map. We adopt Burstall's approach [4], recently emphasized by Bornat [3] then Mehta and Nipkow [17]. The key idea is to have one Why variable for each structure field. Indeed, two structure fields cannot be aliased (as soon as one conforms to ANSI C). The very useful consequence is that whenever one structure field is updated, only the corresponding Why variable is modified and we have for free that any other field is left untouched. We extend this model to arrays and more generally to pointer arithmetic.

The set of variables representing the C memory is displayed on Figure 4. All these variables contain applicative maps indexed by addresses a_1, a_2, ... belonging to some abstract data type `addr`. The variable `alloc` on the left of the figure is the *allocation store* which tells for each address whether it is allocated and, when so, the size of the block it points to. All other variables map addresses to 1-dimensional arrays indexed by integers. A type `pointer` is introduced to represent C pointers. A value of type `pointer` is either the `null` pointer or a pair of an address and an integer index. For instance, address a_2 points to a block of length 5 according to `alloc` and thus the pointer $(a_2, 3)$ refers to the fourth element of each array $f_1(a_2)$, $f_2(a_2)$, ... The variable `intP` is used for arrays of integers i.e. pointers to `int`, `intPP` for pointers to pointers to `int`, etc. Similarly the variable `pointerP` is used for pointers to structures, `pointerPP` for pointers to pointers to structures, etc.

On the Why side, each of those applicative maps is of type α `memory`, an abstract polymorphic datatype equipped with the following side-effect free functions, to access a cell and to update it, respectively.

```
acc: α memory, pointer → α
upd: α memory, pointer, α → α memory
```

Depending on the memory used, α is either `int`, `float` or `pointer`. We also introduce one Why variable for each C global variable or static local variable. Finally, C local variables are represented using Why local variables since they cannot be aliased.

Effects inference. Once we have determined the set of variables representing the memory state, a non-trivial step to be able to deal modularly with function calls is to statically compute the *effects* of each statement and each function. An effect is a pair of sets of Why variables respectively (possibly) accessed and modified. Such an effect appears on Figure 3 (the **reads** and **writes** clauses).

Computing statements effects is straightforward: it is only a matter of traversing statements and expressions recursively, collecting accessed and modified variables along and assuming known effects for all functions. Computing functions effects is then obtained by iterating effects inference on each function body until a fixpoint is reached, which is guaranteed to happen since the set of variables is finite and effects can only grow during this process.

3.3 Translating C source code

This section describes the translation from C source code to Why source code. The translation is realized through four mutually recursive interpretations:

- $[\![e]\!]$: the interpretation of a C expression e;
- $[\![e]\!]_b$: the interpretation of a C expression e as a Why boolean expression—though C does not have booleans, the translation is more natural when a C expression e used as a boolean is directly translated into a Why boolean;
- $[\![e]\!]_l$: the interpretation of a C *left value* e, i.e. an expression allowed on the left of an assignment;
- $[\![s]\!]$: the interpretation of a C statement s.

Left values. The interpretation of a left value e is either a Why reference x or a pair (m, p) of a memory m and a pointer expression p. This interpretation is defined as follows:

- $[\![x]\!]_l = x$
- $[\![\texttt{*}e]\!]_l = (m, [\![e]\!])$, where m depends on the type of e, e.g. `intP` for `int*`, `intPP` for `int**`, etc.
- $[\![e\texttt{->}f]\!]_l = (f, [\![e]\!])$

Expressions. We only give the most significant constructs. Translation of memory access and assignment is done via two abstract Why programs

parameter $\texttt{access} : m : \alpha\,\texttt{memory ref} \rightarrow p : \texttt{pointer} \rightarrow$
$\{\texttt{valid}(alloc, p)\}\ \alpha\ \textbf{reads}\ alloc, m\ \{result = \texttt{acc}(m, p)\}$
parameter $\texttt{update} : m : \alpha\,\texttt{memory ref} \rightarrow p : \texttt{pointer} \rightarrow v : \alpha \rightarrow$
$\{\texttt{valid}(alloc, p)\}\ \texttt{unit}\ \textbf{reads}\ alloc, m\ \textbf{writes}\ m\ \{m = \texttt{upd}(m@, p, v)\}$

where $\texttt{valid}(alloc, p)$ expresses that p can be safely dereferenced according to $alloc$. The precise meaning of `valid` is given Section 3.4.

- $[\![x]\!]$ = `!`x
- $[\![\texttt{*}e]\!]$ = `access` $m\ p$, when $[\![\texttt{*}e]\!]_l = (m, p)$
- $[\![e\texttt{->}f]\!]$ = `access` $f\ [\![e]\!]$
- $[\![x = e_2]\!]$ = x `:=` $[\![e_2]\!]$`;` `!`x
- $[\![e_1 = e_2]\!]$ = `let` $v_1 = p_1$ `in let` $v_2 = [\![e_2]\!]$ `in update` $m\ v_1\ v_2$`;` v_2, when $[\![e_1]\!]_l = (m, p_1)$
- $[\![e_1 + e_2]\!]$ = `shift` $[\![e_1]\!]\ [\![e_2]\!]$, when e_1 is a pointer
- $[\![e_1\ op\ e_2]\!]$ = $[\![e_1]\!]\ op\ [\![e_2]\!]$, with $op \in \{$`+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`$\}$
- $[\![e_1\ op\ e_2]\!]$ = `if` $[\![e_1\ op\ e_2]\!]_b$ `then 1 else 0`, with $op \in \{$`==`, `!=`, `>`, `>=`, `<`, `<=`, `&&`, `||`$\}$
- $[\![\texttt{f}(e_1, \ldots, e_n)]\!]$ = `f_parameter(`$[\![e_1]\!]$,…,$[\![e_n]\!]$`)`

Boolean expressions. The interpretation of a "boolean" expression is defined as follows:

- $[\![e_1\ op\ e_2]\!]_b$ = `let` $v_1 = [\![e_1]\!]$ `in let` $v_2 = [\![e_2]\!]$ `in` $v_1\ op\ v_2$, with $op \in \{$`==`, `!=`, `>`, `>=`, `<`, `<=`$\}$
- $[\![e_1\ \texttt{\&\&}\ e_2]\!]_b$ = `if` $[\![e_1]\!]_b$ `then` $[\![e_2]\!]_b$ `else false`
- $[\![e_1\ \texttt{||}\ e_2]\!]_b$ = `if` $[\![e_1]\!]_b$ `then true else` $[\![e_2]\!]_b$
- $[\![\texttt{!}e]\!]_b$ = `not` $[\![e]\!]_b$
- $[\![e]\!]_b$ = $[\![e]\!]$ `<> 0`, otherwise

Statements. The translation of a C statement is defined as follows:

- $[\![e]\!]$ = `let _ =` $[\![e]\!]$ `in void`
- $[\![e_1\texttt{;}\ e_2]\!]$ = $[\![e_1]\!]$`;` $[\![e_2]\!]$
- $[\![\{\ \beta_1\ \texttt{x}_1\texttt{=}\ e_1\texttt{;}\ \ldots\ \beta_n\ \texttt{x}_n\texttt{=}\ e_n\texttt{;}\ s\ \}]\!]$
  = `let` $\texttt{x}_1$ `= ref` $[\![e_1]\!]$ `in` … `let` $\texttt{x}_n$ `= ref` $[\![e_n]\!]$ `in` $[\![s]\!]$

  The translation above assumes that each variable $\texttt{x}_i$ has an initializer e_i. If a local variable has no initializer, it is indeed initialized using a call to an unspecified function, thus simulating a random initialization.
- $[\![\texttt{for}(e_1\texttt{;}\ e_2\texttt{;}\ s_3)\ s]\!]$ =

  $[\![e_1]\!]$`;`
  `try`
  `while` $[\![e_2]\!]$ `do try` $[\![s\texttt{;}\ s_3]\!]$ `with Continue -> void end done`
  `with Break -> void end`

  A similar translation is used for `while` and `do` loops.
- $[\![\texttt{return}\ e]\!]_s$ = `raise (Return` $[\![e]\!]$`)`

Finally, a C function definition is interpreted as a Why function definition where the exception `Return` is caught appropriately:

$[\![\texttt{f}(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ \{\ s\ \}]\!]$ =
`let f (`x_1:$[\![\tau_1]\!]$`)` … `(`x_n:$[\![\tau_n]\!]$`) = try` $[\![s]\!]$ `with Return v -> v end`

Addresses. In order to interpret the C prefix operator `&` which returns the address of a left value, we first manage to have structures always referred to via pointers, artificially inserting pointers if needed. Therefore the expression $e\texttt{.}f$ is everywhere interpreted as $e\texttt{->}f$. Then we introduce a new interpretation $[\![e]\!]_a$ which stands for the memory location denoted by e. It is defined as follows:

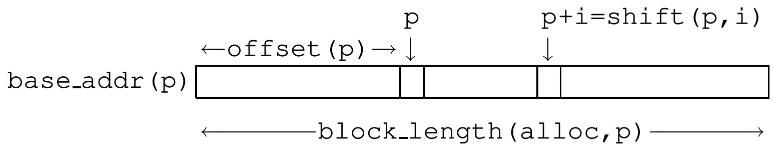
– $[\![x]\!]_a = \,!\,x$, when x is a pointer
– $[\![\star e]\!]_a = [\![e]\!]$
– $[\![e\texttt{.}f]\!]_a = [\![e\texttt{->}f]\!]_a = [\![e]\!]$, when field f is of numerical type
– $[\![e\texttt{.}f]\!]_a = [\![e\texttt{->}f]\!]_a = \texttt{access}\ \ f\ [\![e]\!]$, when field f is a pointer

This interpretation is the precise meaning of `&`, that is $[\![\texttt{\&}e]\!] = [\![e]\!]_a$.

3.4 Background theory

To be able to prove the obligations generated from a given program, it remains to provide a suitable theory, i.e. a set of axioms giving a meaning to all logical predicates and function symbols introduced so far. We give here only the significant part of the theory.

Validity of pointer dereferencing. First, a theory is introduced in order to establish obligations required by pointer dereferencing, i.e. the $\texttt{valid}(alloc, p)$ precondition to the Why functions `access` and `update` (Section 3.3). A pointer `p` is a pair of a base address, `base_addr(p)`, and an offset inside the block, `offset(p)`. Pointer arithmetic is only a matter of changing offsets inside a given block. If `i` is an integer then the pointer `p+i` has the same base address as `p` and offset `offset(p)+i`. This situation is illustrated below:



A memory block is constituted of zero, one or several elements; the length of the block `p` points at is `block_length(alloc,p)`, where `alloc` is the allocation store. Consequently, `valid` is defined as

$$\texttt{valid}(a, p) \equiv p \neq \texttt{null} \ \wedge\ 0 \leq \texttt{offset}(p) < \texttt{block\_length}(a, p)$$

Additional predicates are introduced for convenience, to express that access to `p+i` and to the whole range `p[i...j]` are valid:

$$\texttt{valid\_index}(a, p, i) \equiv p \neq \texttt{null} \ \wedge\ 0 \leq \texttt{offset}(p) + i < \texttt{block\_length}(a, p)$$

$$\begin{aligned}\texttt{valid\_range}(a, p, i, j) \equiv\ & p \neq \texttt{null} \ \wedge\ 0 \leq \texttt{offset}(p) + i \ \wedge\ i \leq j \ \wedge \\ & \texttt{offset}(p) + j < \texttt{block\_length}(a, p)\end{aligned}$$

Note that two different blocks are truly *separated* i.e. there is no way to access and/or modify any element of a block using a pointer related to another block. There is no way to express that two blocks allocated with two different calls to `malloc` are contiguous in memory. As specified in the ANSI C standard, pointer subtraction and pointer comparison are restricted to pointers with a same base address. When it is so, pointer subtraction (resp. comparison) corresponds to the subtraction (resp. comparison) of offsets.

Access and update of memories. The next set of axioms is related to the combination of accesses and updates within memories. As expected, we have the following two axioms defining the theory of arrays [12]:

axiom $\mathtt{acc\_upd} : \forall m : \alpha\,\mathtt{memory}, \forall p : \mathtt{pointer}, \forall a : \alpha,$
$\quad\mathtt{acc}(\mathtt{upd}(m, p, a), p) = a$
axiom $\mathtt{acc\_upd\_neq} : \forall m : \alpha\,\mathtt{memory}, \forall p_1, p_2 : \mathtt{pointer}, \forall a : \alpha,$
$\quad p_1 \neq p_2 \rightarrow \mathtt{acc}(\mathtt{upd}(m, p_1, a), p_2) = \mathtt{acc}(m, p_2)$

These two axioms allow to handle pointer aliasing in a natural way: pointers are aliased iff they are equal. In other words, the second axiom simply says that the memory location $\mathtt{acc}(m, p_2)$ is unchanged by a memory update at location p_1 whenever $p_1 \neq p_2$.

Dealing with `assigns` clauses. `assigns` clauses are essential for reasoning modularly when several functions are involved. They are modeled using a Why predicate `assigns`. The postcondition of the translated function `f_parameter` of some C function `f` has one `assigns` predicate for each of the Why variables it modifies, as computed by the analysis of effects described in Section 3.2. The `assigns` clause is automatically split into various `assigns` predicates for each modified variable. See Figure 3 for example.

Formally, an `assigns` clause with location loc specifies that in the post-state of the considered function, every memory location which is already allocated in the pre-state and is not included in loc is unchanged. Thus `assigns` is defined as

$$\begin{array}{l}\mathtt{assigns}(a : \mathtt{alloc\_table}, m_1, m_2 : \alpha\,\mathtt{memory}, loc : \mathtt{assign\_loc}) := \\ \quad \forall\, p : \mathtt{pointer}, \\ \qquad \neg\mathtt{fresh}(a, p) \wedge \mathtt{unchanged}(p, loc) \rightarrow \mathtt{acc}(m_1, p) = \mathtt{acc}(m_2, p)\end{array}$$

where `assign_loc` is a type representing a set of modifiable locations. It remains to give axioms for the `unchanged` function, depending on the form of the locations set loc:

- The clause `assigns \nothing` specifies that nothing is modified. It is interpreted with a new constant `nothing_loc` of type `assign_loc` with the axiom:

$$\forall p : \mathtt{pointer},\ \mathtt{unchanged}(p, \mathtt{nothing\_loc})$$

- The clause `assigns p->f` specifies that the field `f` of `p` is modified. It is interpreted with a new function `pointer_loc` of type `pointer` $\rightarrow$ `assign_loc` with the axiom:

$$\forall p', p : \mathtt{pointer},\ \mathtt{unchanged}(p', \mathtt{pointer\_loc}(p)) \leftrightarrow p' \neq p$$

– The clauses `assigns t[i..j]` and `assigns t[*]` are interpreted using functions `range_loc` and `all_loc` with axioms

$$\forall p : \texttt{pointer},\ n : \texttt{int},\ t : \texttt{pointer},\ i : \texttt{int},\ j : \texttt{int},$$
$$\texttt{unchanged}(p, n, \texttt{range\_loc}(t, i, j)) \leftrightarrow p \neq t \vee n < i \vee n > j$$
$$\forall p : \texttt{pointer}\ n : \texttt{int}\ t : \texttt{pointer},$$
$$\texttt{unchanged}(p, n, \texttt{all\_loc}(t)) \leftrightarrow p \neq t$$

When a clause `assigns` l_1, l_2 specifies several (say two) locations l_1 and l_2, then two cases may occur: either l_1 and l_2 refer to locations represented by different variables and then a conjunction of two `assigns` assertions is built, or they refer to the same variable and then the clause is interpreted using the function `union_loc` satisfying

$$\forall p : \texttt{pointer},\ l_1, l_2 : \texttt{assign\_loc},$$
$$\texttt{unchanged}(p, \texttt{union\_loc}(l_1, l_2)) \leftrightarrow \texttt{unchanged}(p, l_1) \wedge \texttt{unchanged}(p, l_2)$$

Finally if a variable is detected as written by the effects analysis but there is no `assigns` location referring to it, we add an assertion $\texttt{assigns}(\ldots, \texttt{nothing\_loc})$ for it.

As a toy example, let us imagine a test function on our queue data structure, given Figure 5. Proof of the postcondition needs the fact that `q1` is not modified by the call to `pop()`. This is ensured by the precondition `q1 != &q` that forbids aliasing of `q1` and `q`, and the `assigns` clause for `pop`. This is automatically proved by Simplify.

```
/*@ requires \valid(q1) && q1 != &q && !q.empty
  @ ensures \result == \old(q1->empty) */
int test(struct queue *q1) {
    pop();
    return q1->empty;
}
```

Fig. 5. A toy example requiring the use of `assigns` clauses

3.5 Soundness of the approach

A main goal of our work is to guarantee the soundness with the highest level of confidence: contrary to some static checking methods, we ensure that if all proof obligations for a given function are proved valid, then the function is a correct implementation of its specification. However, this is only true for a restricted fragment of C. Indeed, possible source of bugs that are not yet covered by our tool include:

– Non-terminating recursive functions;
– Initialization of global variables not satisfying invariants;
– Arithmetic overflow;
– Use of `&` operator on stack-allocated variables.

The soundness of our method relies on both Why's own soundness and correctness of the translation from C to Why. Why's verification condition generator has been proved [9]. Moreover, each run of the Why tool produces a *validation*, which is an automatically checkable certificate. On the other hand, soundness of the translation from

C is not formally proved, but we believe that it is clearly and simply enough stated in this paper to be trusted.

Notice that there exist another possible source of error: the axioms added by the user and the background theory may be inconsistent. To ensure consistency, it is always possible to realize these axioms (in Coq for example).

4 Conclusion

We proposed a method for verifying C programs with pointers and a prototype implementation. Our approach is based on Burstall's model for structures, which we extended to arrays and pointers. We could successfully apply our tool to non-trivial case studies [17], moreover checking the absence of null pointer dereferencing. We took advantage of the modular architecture of Why allowing verification condition generation for various provers.

Similar verification tools for C program exist, such as Astrée [2], CAVEAT [5] and SDV [6]. However, these tools are bounded to a specific prover, hence mostly do not allow to conduct a formal proof within an interactive proof assistant. Furthermore, our specification language has original features, such as `assigns` clauses, which allow modular reasoning with pointer aliasing. Compared to separation logic [20] which seeks the same goal, a clear advantage of our approach is that we still use a standard first-logic, so we are able to use existing provers. On the other hand, our specification language is probably not as powerful as separation logic.

In a near future, we plan to add support for bounded integer arithmetic and floating-point arithmetic, to disallow possible overflow. This is not hard at all because it is only a matter of interpreting arithmetic operators into new abstract Why functions with appropriate preconditions. Other plans include support for initialization code as mentioned in previous section, C `union` construct, library functions such as `malloc` and `strcpy`, which all require appropriate extensions to our specification language.

Other interesting issues include assistance to the construction of specifications. Suitable preconditions could be generated or at least suggested automatically, either by analysis of counter-examples provided by automatic provers, or by exploitation of Why weakest precondition calculus. There is also an interesting issue about the cooperation of output provers: it would be nice if automatic provers could generate traces of their proofs, which could be double-checked by a proof assistant: in this way, only proof obligations that cannot be solved automatically would need to be proved manually. To obtain such a trace, the use of the haRVey [19] tool is currently under investigation.

Current experiments with Caduceus on embedded programs for smart cards and avionics show some scaling up issues in our approach. We are now investigating techniques for integrating static analysis into the tool, both for solving those issues, and more generally to assist the user in adding annotations in its code.

Acknowledgments. We thank June Andronick from Axalto, Dillon Pariente and Emmanuel Ledinot from Dassault Aviation for their very useful feedback about the specification language and the Caduceus prototype.

References

1. The Simplify decision procedure (part of ESC/Java). `http://research.compaq.com/SRC/esc/simplify/`.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE static analyzer. `http://www.astree.ens.fr/`.
3. R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.
4. R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 72.
5. CAVEAT project. `http://www-drt.cea.fr/Pages/List/lse/LSL/Caveat/index.html`.
6. B. Cook. Static driver verifier. `http://www.microsoft.com/whdc/devtools/tools/sdv.mspx`.
7. The Coq Proof Assistant. `http://coq.inria.fr/`.
8. P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 841–993. North-Holland, 1990.
9. J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.
10. J.-C. Fillitre. The why verification tool, 2002.
11. Jean-Christophe Filliâtre and Claude Marché. The Caduceus tool for the verification of C programs. `http://why.lri.fr/caduceus/`.
12. D. Kaplan. Some Completeness Results in the Mathematical Theory of Computation. *J. ACM*, 15(1):124–34, Jan 1968.
13. B. Kernighan and D. Ritchie. *The C Programming Language (2nd Ed.)*. Prentice-Hall, 1988.
14. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106, 2000.
15. C. Marché, C. Paulin, and X. Urbain. The KRAKATOA proof tool, 2002. `http://krakatoa.lri.fr/`.
16. C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.
17. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *19th Conference on Automated Deduction*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
18. The PVS system. `http://pvs.csl.sri.com/`.
19. S. Ranise and D. Deharbe. Light-weight theorem proving for debugging and verifying units of code. In *Proc. SEFM'03*, Canberra, Australia, Sept. 2003. IEEE Computer Society Press. `http://www.loria.fr/equipes/cassis/softwares/haRVey/`.
20. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.
21. Sun Microsystems. The JavaCard^{TM} application programming interface (API). `http://java.sun.com/products/javacard/`.