A Certified Multi-prover Verification Condition Generator*

Paolo Herms[1,2,3], Claude Marché[2,3], and Benjamin Monate[1]

[1] CEA, LIST, Lab. de Sûreté du Logiciel, Gif-sur-Yvette F-91191
[2] INRIA Saclay - Île-de-France, 4 rue Jacques Monod, Orsay, F-91893
[3] Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405

Abstract. Deduction-based software verification tools have reached a maturity allowing them to be used in industrial context where a very high level of assurance is required. This raises the question of the level of confidence we can grant to the tools themselves. We present a certified implementation of a verification condition generator. An originality is its genericity with respect to the logical context, which allows us to produce proof obligations for a large class of theorem provers.

1 Introduction

Among the various classes of approaches to static verification of programs, the so-called *deductive verification* approach amounts to verifying that a program satisfies a given behavioral specification by means of theorem proving. Typically, given a program and a formal specification, a verification condition generator produces a set of logical formulas, that must be shown to be valid by some theorem prover. Deductive verification tools have nowadays reached a maturity allowing them to be used in industrial context where a very high level of assurance is required [25]. This raises the question of the level of confidence we can grant to the tools themselves. This is the question we address in this paper.

One can distinguish two main kinds of deductive verification approaches. The first kind is characterized by the use of a deep embedding of the input programming language in a general purpose proof assistant. One of the earlier work of this kind is done in the SunRise system in 1995 [17] where a simple imperative language is defined in HOL, with a formal operational semantics. A set of Hoare-style deduction rules are then shown valid. A SunRise program can then be specified using HOL assertions, and proved in the HOL environment.

The second kind of approaches provide standalone verification condition generators automatically producing verification conditions, usually by means of variants of Dijkstra's weakest precondition calculus. This is the case of ESC/Java [10], B [1] ; the Why platform [14] and its Java [22] and C [13] front-ends ; and Spec# [3] and VCC [11] which are front-ends to Boogie [2]. Being independent of

* This work is partly supported by the *U3CAT* project (ANR-08-SEGI-021, `http://frama-c.com/u3cat/`) of the French national research organization (ANR).

any underlying proof assistant, these tools analyze programs where formal specifications are given in ad-hoc annotation language such as JML [7] and ACSL [4]. However, up to now these standalone tools have never been formally proved to be sound.

Our goal is to combine the best of both approaches: a guaranteed sound VC generator, able to produce VCs for multiple provers. We implement and prove sound, in the Coq proof assistant [5], a VC generator inspired by the former Why tool. To make it usable with arbitrary theorem provers as back-ends, we make it generic with respect to a *logical context*, containing arbitrary abstract data types and axiomatizations. Such a generic aspect is suitable to formalize memory models needed to design front-ends for mainstream programming language, as it is done for C by VCC above Boogie or Frama-C/Jessie above Why. The input programs of our VC generator are imperative programs written in a core language which operates on mutable variables whose values are data types of the logical context. The output logic formulas are built upon the same logical context. This certified Coq implementation is crafted so it can be extracted into a standalone executable.

Section 2 formalizes our notion of generic logical contexts. Section 3 formalizes our core language, and defines its operational semantics. Section 4 defines the weakest precondition computation WP and proves its *soundness*. Theorem 1 states that if for each function of a program, its pre-condition implies the WP of its post-condition, then all its annotations are satisfied. Section 5 aims at the extraction of a standalone executable. We introduce a variant wp of the calculus which produces concrete formulas instead of Coq ones. Theorem 2 states that wp obligations imply the WP obligations. The main result is then Theorem 3 which states the soundness of the complete VC generation process. We conclude in Section 6 by comparing with related works and discussing perspectives. Due to lack of space, we sometimes refer to our technical report [16] for technical details. The sources of the underlying Coq development are available at `http://www.lri.fr/~herms/` .

2 Logical Contexts

Our background logic is multi-sorted first-order logic with equality. Models for specifying programs can be defined by declaring types, constant, function and predicate symbols and axioms. Models may integrate predefined theories, a typical example being integer arithmetic.

Definition 1. *A* logical signature *is composed of (1) a set utype of sort names introduced by the user ; (2) a set* sym *of constant, function and predicate symbols; (3) a set* ref *of global reference names and (4) a set* exc *of exceptions names. The set of all data types is defined by the grammar*

$$type ::= \text{Tuser } utype \mid \text{Tarrow } type\ type \mid \text{Tprop}$$

that is, the types are completed with built-in types for propositions and functions. We require every symbol, exception and reference to have an associated

```
type array
logic select : array -> int -> int
logic store :
 array -> int -> int -> array
axiom select_eq: forall a,i,x.
 select (store a i x) i = x
axiom select_neq : forall a,i,j,x.
 i <> j ->
  select (store a i x) j =
  select a j

logic sorted : array -> int -> prop
axiom sorted_def: forall a,n.
 sorted a n <->
 forall i,j. 0 <= i <= j < n ->
  select a i <= select a j

logic swap : array -> array ->
              int -> int -> prop
```

```
axiom swap_def: forall a,b,i,j.
 swap a b i j <->
 select a i = select b j /\
 select a j = select b i /\
 forall k. k <> i /\ k <> j ->
  select a k = select b k

logic permut:
 array -> array -> int -> prop
axiom permut_refl: forall a,n.
 permut a a n
axiom permut_sym: forall a,b,n.
 permut a b n -> permut b a n
axiom permut_trans:
 forall a,b,c,n.
  permut a b n /\ permut b c n ->
  permut a c n
axiom permut_swap:
 forall a,b,i,j,n.
  0 <= i < n /\ 0 <= j < n /\
  swap a b i j -> permut a b n
```

Fig. 1. Logical context for sorting

type. In our Coq implementation, *sym*, *ref*, and *exc* are of type *type* → Type (see the report for details [16]). The parameter of the latter are written as subscript in the following.

Example 1. Fig. 1 presents an appropriate model for specifying a program for sorting an array. An abstract type `array` is introduced to model arrays of integers indexed by integers. It is axiomatized with the well-known theory of arrays. We also define predicates (`sorted` t i) meaning that $t[0], \ldots, t[i-1]$ is an increasing sequence, and (`permut` t_1 t_2) meaning that t_1 is a permutation of t_2. The latter is axiomatized: it is an equivalence relation that contains all transpositions (`swap`) of two elements.

The logical signature of this example is thus given by *utype* = {`array`} and *sym* = {`select`, `store`, `sorted`, `swap`, `permut`} (*ref* and *exc* will come later). Each symbol is annotated by the appropriate *type*, e.g. `select` : $sym_{(\text{Tarrow}(\text{Tuser }\texttt{array})(\text{Tarrow Tint Tint}))}$.

2.1 Dependently Typed *de Bruijn* Indices

A design choice in our formalization is to define terms and expressions such that they are well typed by construction. This simplifies the definition of the semantics and the weakest precondition calculus on such expressions, as we don't need to handle malformed constructions at those points. To begin we need to ensure that occurrences of variables actually correspond to bound variables in their current scopes and that they are used with the correct type. Here we use so-called dependently typed *de Bruijn* indices following the preliminary approach of Herms [15] as documented in [8].

Dependent indices are like regular *de Bruijn* indices, in that I_0 refers to the innermost bound variable, $(I_S\ I_0)$ to the second innermost bound variable, etc. Additionally they carry information about their typing environment and about the type of the variable they represent. We use indices of type $\mathit{idx}_{A,E}$ to represent variables of type A under a typing environment E, that is the list of the types of the bound variables. The type of the innermost bound variable is stored at the first position in the typing environment, the type of the second innermost bound variable at the second position, etc. In Coq we can formalize this constraint about the valid parameters of *idx* by assigning its constructors the types $I_0 : \mathit{idx}_{A,A::E}$ and $I_S : \mathit{idx}_{A,E} \to \mathit{idx}_{A,B::E}$ (see [16]).

Dependent indices are thus placeholders within terms but they can also be used to reference elements within heterogeneous lists. In such a heterogeneous list each element may have a different type. The type hlist_E of heterogeneous lists then depends on the list of types E of their elements. Thanks to the constraints on the type parameters, if an index $i : \mathit{idx}_{A,E}$ references an element within a heterogeneous list $l : \mathit{hlist}_E$, we are sure to find an element of type A at i-th position of l. This allows us to define the function $\mathit{accsidx} : \mathit{idx}_{A,E} \to \mathit{hlist}_E \to A$ which recursively accesses elements within a heterogeneous list.

We will use these heterogeneous lists to give semantics to our languages. Precisely, heterogeneous lists are the representation of evaluation environments which associate a value to each variable in the current typing environment. The function *accsidx* is then used in the semantics rule for variable access.

Example 2. The heterogeneous list $l = [5; \mathit{true}; \mathit{succ}]$ has type $\mathit{hlist}\ [\mathbb{Z}; \mathit{bool}; \mathbb{Z} \to \mathbb{Z}]$. *De Bruijn* indices $I_0 : \mathit{idx}_{\mathbb{Z},[\mathbb{Z};\mathit{bool};\mathbb{Z}\to\mathbb{Z}]}$ and $I_S\ (I_S\ I_0) : \mathit{idx}_{(\mathbb{Z}\to\mathbb{Z}),[\mathbb{Z};\mathit{bool};\mathbb{Z}\to\mathbb{Z}]}$, are well-typed and can be used to access their values, e.g. $\mathit{accsidx}\ I_0\ l = 5 : \mathbb{Z}$ and $\mathit{accsidx}\ (I_S\ I_0)\ l = \mathit{true} : \mathit{bool}$.

2.2 Terms and Propositions

Terms and propositions follow the usual classical first-order logic. For the need of programs, we add the declaration of local names using `let` binders, the access to a reference r (with concrete syntax `!r`) and the dereferencing of such a reference in a former state labeled by l (concrete syntax `r@l`). Labels are represented by bounded integers and new labels can be declared at the expression level.

The formal syntax of terms and propositions is given in Fig. 2. Terms $t_{L,E,A}$ and propositions $p_{L,E}$ depend on the parameters E and L, denoting respectively the typing environment and the highest index of a valid label. Terms additionally depend on the parameter A, the type of the value they denote. Variables are represented by our dependent indices $\mathit{idx}_{A,E}$. The constructor Tlet expresses let-blocks at the term level. As usual with *de Bruijn* indices, no variable name is given and the body of the block is typed in a typing environment that is enriched

$$
\begin{array}{rcll}
t_{L,E,A} & ::= & \text{Tconst } \mathit{sym}_A & \\
& \mid & \text{Tvar } \mathit{idx}_{A,E} & \\
& \mid & \text{Tapp } t_{L,E,(\text{Tarrow} B A)} \ t_{L,E,B} & \\
& \mid & \text{Tlet } t_{L,E,B} \ t_{L,B::E,A} & \\
& \mid & \text{Tderef } \mathit{ref}_A & (^*\ \texttt{!r}\ ^*) \\
& \mid & \text{Tat } \mathit{label}_L \ \mathit{ref}_A & (^*\ \texttt{r@l}\ ^*)
\end{array}
\qquad
\begin{array}{rcl}
p_{L,E} & ::= & \text{Peq } t_{L,E,A} \ t_{L,E,A} \\
& \mid & \text{Pand } p_{L,E} \ p_{L,E} \\
& \mid & \text{Pimply } p_{L,E} \ p_{L,E} \\
& \mid & \text{Pforall } p_{L,A::E} \\
& \mid & \text{Plet } t_{L,E,A} \ p_{L,A::E} \\
& \mid & \text{Pfalse} \\
& \mid & \text{Pterm } t_{L,E,\text{Tprop}}
\end{array}
$$

Fig. 2. Inductive definitions of terms and propositions

by the type of the term to be remembered. The symbol application is formalized in a curryfied style. For the propositions we define only the ones needed within the WP calculus. The constructor Pterm allows to construct user-defined atomic propositions from terms. As Tlet at the term-level, Plet expresses let-blocks at the level of props and binds a new *de Bruijn* variable. Similarly Pforall binds a new de Bruijn variable but generalizing it instead of assigning a value to it. The Pforall and the Plet bind a new *de Bruijn* variable.

2.3 Logical Contexts, Semantics

The semantics of our generic language depends on an *interpretation* given to types and symbols. From such an interpretation, any term or proposition can be given a value, in a given *environment* for variables and given *state* for references.

Given a logical signature, an *interpretation* is a pair of a function *denutype* giving an interpretation of the user types, and a function *densym* giving an interpretation of the introduced function and predicate symbols. Given *denutype* we define *dentype* to interpret all types. An *evaluation environment* Γ of type env_E is a heterogeneous list as described above. A *memory state* S of type $state_L$ is a vector of size L of mappings from references ref_A to values of type $dentype A$. The first element denotes the current state whereas the $(l+1)$-nth element denotes the state labeled by l. This is the reason for the L-parameter of terms and propositions. A term of type $t_{L,E}$ can be safely evaluated in a state of type $state_L$. As a special case, a $state_0$ is only composed of the current state and $t_{0,E}$ cannot contain any labeled dereferenciation at all. The semantics of terms is defined by structural recursion (Fig. 3), where we use the syntactic sugar $Here\ S = S[0]$ and $At\ S\ l = S[l+1]$ by analogy to the syntax. Note how the rules for Tlet and Pforall push the newly bound variable into Γ. Note also how correct typing is ensured by construction.

A *logical context* is a pair of a logical signature and a set of axioms over it. The programs that will be written in the next section will assume a given logical context. The goal is to prove them valid with respect to *any* interpretation which makes the axioms of that context valid: this will allow us to use various provers to discharge them.

$$\begin{aligned}
[\![\text{Tconst } s]\!]_{\Gamma,S} &::= \mathit{densym}\ s \\
[\![\text{Tvar } v]\!]_{\Gamma,S} &::= \mathit{accsidx}\ \Gamma\ v \\
[\![\text{Tderef } r]\!]_{\Gamma,S} &::= \mathit{Here}\ S\ r \\
[\![\text{Tapp } t_1\ t_2]\!]_{\Gamma,S} &::= ([\![t_1]\!]_{\Gamma,S}\ [\![t_2]\!]_{\Gamma,S}) \\
[\![\text{Tlet } t_1\ t_2]\!]_{\Gamma,S} &::= [\![t_2]\!]_{[\![t_1]\!]_{\Gamma,S}::\Gamma,S} \\
[\![\text{Tat } l\ r]\!]_{\Gamma,S} &::= \mathit{At}\ S\ l\ r
\end{aligned}
\qquad
\begin{aligned}
[\![\text{Peq } t_1\ t_2]\!]_{\Gamma,S} &::= [\![t_1]\!]_{\Gamma,S} = [\![t_2]\!]_{\Gamma,S} \\
[\![\text{Pand } p_1\ p_2]\!]_{\Gamma,S} &::= [\![p_1]\!]_{\Gamma,S} \wedge [\![p_2]\!]_{\Gamma,S} \\
[\![\text{Pimply } p_1\ p_2]\!]_{\Gamma,S} &::= [\![p_1]\!]_{\Gamma,S} \rightarrow [\![p_2]\!]_{\Gamma,S} \\
[\![\text{Pforall } p]\!]_{\Gamma,S} &::= \forall b : B,\ [\![p]\!]_{b::\Gamma,S} \\
[\![\text{Plet } t\ p]\!]_{\Gamma,S} &::= [\![p]\!]_{[\![t]\!]_{\Gamma,S}::\Gamma,S} \\
[\![\text{Pfalse}]\!]_{\Gamma,S} &::= \bot \\
[\![\text{Pterm } t]\!]_{\Gamma,S} &::= [\![t]\!]_{\Gamma,S}
\end{aligned}$$

Fig. 3. Denotational semantics of terms and propositions

3 The Core Programming Language

3.1 Informal Description

Our core language follows most of the design choices of the input language of Why. Indeed we reduce to an even more basic set of constructs, nevertheless remaining expressive enough to encode higher-level sequential algorithms. We follow an ML-style syntax; in particular there is no distinction between expressions and instructions. A program in this language is defined by a logical context and a finite set of function definitions, denoted f below, which can modify the global references of the context and can be mutually recursive.

Following again the Why design, our core language contains an exception mechanism, providing powerful control flow structures. As we will see these can be handled by weakest pre-condition calculus without major difficulty. Loops are infinite ones, with a given invariant. The only way to exit them is by using exceptions. We use $e_1; e_2$ as a shortcut for `let` $v = e_1$ `in` e_2 when the variable v is unused.

A definition of a function follows the structure

$$\texttt{let}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = \{\ p\ \}\ e\ \{\ q\ \}$$

where the predicates p and q are the pre- and the post-condition. The types are those declared in the logical context. In the post-condition, the reserved name *result* is locally bound and denotes the result of the function of type τ and label *Old* is bound to denote the pre-state. Note that exceptions are not supposed to escape function bodies. We could easily support such a feature by adding a family of post-conditions indexed by exceptions as in Why [12].

Example 3. In Fig. 4 is a program that sorts the global array t by the classical selection sort algorithm. Note the use of the exception `Break` to exit from the infinite loops. Note also the use of labels in annotations, allowing to specify assertions, loop invariants and post-conditions that link up various states of execution.

3.2 Formal Syntax of Expressions

Like terms of the logic, expressions of programs are formalized by an inductive type $e_{L,E,A}$ depending on the parameters A, E and L, denoting respectively

```
ref t : array

let swap(i:int, j:int) : unit =
 { true }
 let tmp = select !t i in
 t := store !t i (select !t j);
 t := store !t j tmp
 { swap !t t@Old i j }

ref mi, mv, i, j : int
exc Break : unit

let selection_sort(n:int) : unit =
 { n >= 1 }
 i := 0;
 try loop
  { invariant 0 <= !i < n /\
     sorted !t i /\
     permut !t t@Old n /\
     forall k1,k2.
      0 <= k1 < i <= k2 < n ->
       select !t k1 <= select !t k2 }
  if !i >= n-1
   then raise (Break ()) else ();
```

```
  (* look for minimum value
     among t[i..n-1] *)
  mv := select !t !i; mi := !i;
  j := !i+1;
  try loop
   { invariant !i < !j /\
       !i <= !mi < n /\
       !mv = select !t !mi /\
      forall k. !i <= k < !j ->
       select !t k >= !mv }
   if !j >= n
    then raise (Break ()) else ();
   if select !t !j < !mv
    then (mi := !j ;
          mv := select !t !j)
    else ();
   j := !j + 1
  catch Break(v) ();
 label Lab:
  swap(!i,!mi);
  assert { permut !t t@Lab n } in
  i := !i + 1
 catch Break(v) ();
{ sorted !t n /\ permut !t t@Old n }
```

Fig. 4. Selection sort in our core language

$e_{L,E,A}$ ::= Eterm $t_{L,E,A}$	pure term t
$\mid$ Elet $e_{L,E,B}$ $e_{L,B::E,A}$	`let` $v = e_1$ `in` e_2
$\mid$ Eassign ref_A $t_{L,E,A}$	r `:=` t
$\mid$ Eassert $p_{L,E}$ $e_{L,E,A}$	`assert {` p `} in` e
$\mid$ Eraise exc_A $t_{L,E,A}$	`raise (`ex t`)`
$\mid$ Eif $p_{L,E}$ $e_{L,E,A}$ $e_{L,E,A}$	`if` p `then` e_1 `else` e_2
$\mid$ Eloop $p_{L,E}$ $e_{L,E,B}$	`loop { invariant` p `}` e
$\mid$ Etry $e_{L,E,A}$ exc_B $e_{L,B::E,A}$	`try` e_1 `catch` $ex(v)$ e_2
$\mid$ Elab $e_{L+1,E,A}$	`label` l`:` e
$\mid$ Ecall $f_{A,P}$ $(t_{L,E,P_1},...,t_{L,E,P_n})$	call to f

Fig. 5. Inductive definition of expressions

the evaluation type, the typing environment and the highest index of a valid label. Abstract syntax of expressions including comprehensive type annotations is given in Fig 5. Notice that variables v and label l are left implicit in the inductive definition thanks to *de Bruijn* representation. Additionally expressions depend on a parameter F meaning the list of *signatures* of the functions in the program the expression can appear in. A signature is a pair of the return type of the function and the list of the function's parameters. F appears within expressions in function calls where we use dependent indexes to refer to functions, $f_{A,P} := idx_{\langle A,P\rangle,F}$. A function identifier is therefore an index pointing to an

$$\frac{}{\mathit{\Gamma}, S, \text{Eterm } t \Rightarrow S, [\![t]\!]_{\mathit{\Gamma},S}} \qquad \frac{\mathit{\Gamma}, S, e_1 \Rightarrow S', v \qquad v :: \mathit{\Gamma}, S', e_2 \Rightarrow S'', o}{\mathit{\Gamma}, S, \text{Elet } e_1\ e_2 \Rightarrow S'', o}$$

$$\frac{}{\mathit{\Gamma}, S, \text{Eassign } r\ t \Rightarrow S\left[r/[\![t]\!]_{\mathit{\Gamma},S}\right], [\![t]\!]_{\mathit{\Gamma},S}} \qquad \frac{\mathit{\Gamma}, S, e_1 \Rightarrow S', ex\,(v)}{\mathit{\Gamma}, S, \text{Elet } e_1\ e_2 \Rightarrow S', ex\,(v)}$$

$$\frac{\mathit{\Gamma}, S\!\uparrow, e \Rightarrow S', o}{\mathit{\Gamma}, S, \text{Elabel } e \Rightarrow S'\!\downarrow, o} \qquad \frac{[\![p]\!]_{\mathit{\Gamma},S} \qquad \mathit{\Gamma}, S, e \Rightarrow S', o}{\mathit{\Gamma}, S, \text{Eassert } p\ e \Rightarrow S', o}$$

$$\frac{[\![p]\!]_{\mathit{\Gamma},S} \qquad \mathit{\Gamma}, S, e_1 \Rightarrow S', o}{\mathit{\Gamma}, S, \text{Eif } p\ e_1\ e_2 \Rightarrow S', o} \qquad \frac{\neg[\![p]\!]_{\mathit{\Gamma},S} \qquad \mathit{\Gamma}, S, e_2 \Rightarrow S', o}{\mathit{\Gamma}, S, \text{Eif } p\ e_1\ e_2 \Rightarrow S', o}$$

$$\frac{[\![p]\!]_{\mathit{\Gamma},S} \qquad S, e \Rightarrow S', v \qquad S', \text{Eloop } p\ e \Rightarrow S'', o}{S, \text{Eloop } p\ e \Rightarrow S'', o}$$

$$\frac{[\![p]\!]_{\mathit{\Gamma},S} \qquad S, e \Rightarrow S', ex(v)}{S, \text{Eloop } p\ e \Rightarrow S', ex(v)} \qquad \frac{}{\mathit{\Gamma}, S, \text{Eraise } ex\ t \Rightarrow S, ex([\![t]\!]_{\mathit{\Gamma},S})}$$

$$\frac{S, e_1 \Rightarrow S', o \qquad o \neq ex}{S, \text{Etry } e_1\ ex\ e_2 \Rightarrow S', o} \qquad \frac{S, e_1 \Rightarrow S', ex(v) \qquad v :: \mathit{\Gamma}, S', e_2 \Rightarrow S'', o}{S, \text{Etry } e_1\ ex\ e_2 \Rightarrow S'', o}$$

$$\frac{\mathit{\Gamma}_f := [[\![t_1]\!]_{\mathit{\Gamma},S}, ..., [\![t_n]\!]_{\mathit{\Gamma},S}] \qquad [\![\mathsf{pre}_f]\!]_{\mathit{\Gamma}_f,S} \qquad \mathit{\Gamma}_f, S, \mathsf{body}_f \Rightarrow S', v \qquad [\![\mathsf{post}_f]\!]_{v::\mathit{\Gamma}_f,S'}}{\mathit{\Gamma}, S, \text{Ecall } f\ (t_1, ..., t_n) \Rightarrow S', v}$$

Fig. 6. Operational semantics of terminating expressions

element with the signature $\langle A, P\rangle$ within a heterogeneous list of types F. This heterogeneous list $\mathit{hlist}_{\mathit{func}\ F,F}$ is precisely the representation of a program pr_F, where each element is a function $\mathit{func}_{F,\langle A,P\rangle}$.

A function $func_{F,\langle A,P\rangle}$ consists of a body $e_{F,1,E,A}$, a pre-condition $p_{0,P}$ and a post-condition $p_{1,A::P}$. In the pre-condition no labels may appear, hence its type has the parameter 0. In the post-condition we allow referring to the pre-state of a function call: in the syntax this corresponds to using the label *Old*. The post-condition may additionally refer to the result of the function, hence its type environment is enriched by A. Note that in the definition of programs the parameter F appears twice: once as parameter of *hlist*, to define the signatures of the functions in the program, and once as parameter of *func* to constrain expressions in function bodies to refer only to functions with a signature appearing in F. This way we ensure the well-formedness of the graph structure of programs.

3.3 Operational Semantics

The operational semantics is defined in big-step style following the approach of Leroy and Grall [20]. A first set of inference rules inductively defines the semantics of terminating expressions (Fig. 6) and a second set defines the semantics of non-terminating expressions, co-inductively (Fig. 7). Judgement $\mathit{\Gamma}, S, e \Rightarrow S', o$ expresses that in environment $\mathit{\Gamma}$ and state S, the execution of expression e terminates, in a state S' with *outcome* o: either a normal value v or an exception $ex(v)$ where v is the value held by it. There are two rules for `let` e_1 `in` e_2 depending on the outcome of e_1. The rule for assignment uses the update operation $S[r/a]$

$$\frac{\Gamma, S, e_1 \Rightarrow \infty}{\Gamma, S, \text{Elet } e_1\ e_2 \Rightarrow \infty} \qquad \frac{\Gamma, S, e_1 \Rightarrow S', v \qquad v :: \Gamma, S', e_2 \Rightarrow \infty}{\Gamma, S, \text{Elet } e_1\ e_2 \Rightarrow \infty}$$

$$\frac{[\![p]\!]_{\Gamma,S} \qquad \Gamma, S, e_1 \Rightarrow \infty}{\Gamma, S, \text{Eif } p\ e_1\ e_2 \Rightarrow \infty} \qquad \frac{\neg[\![p]\!]_{\Gamma,S} \qquad \Gamma, S, e_2 \Rightarrow \infty}{\Gamma, S, \text{Eif } p\ e_1\ e_2 \Rightarrow \infty}$$

$$\frac{\Gamma, S\!\uparrow, e \Rightarrow \infty}{\Gamma, S, \text{Elabel } e \Rightarrow \infty} \qquad \frac{[\![p]\!]_{\Gamma,S} \qquad \Gamma, S, e \Rightarrow \infty}{\Gamma, S, \text{Eassert } p\ e \Rightarrow \infty} \qquad \frac{[\![p]\!]_{\Gamma,S} \qquad S, e \Rightarrow \infty}{S, \text{Eloop } p\ e \Rightarrow \infty}$$

$$\frac{[\![p]\!]_{\Gamma,S} \qquad S, e \Rightarrow S', v \qquad S', \text{Eloop } p\ e \Rightarrow \infty}{S, \text{Eloop } p\ e \Rightarrow \infty}$$

$$\frac{S, e_1 \Rightarrow \infty}{S, \texttt{try}\, e_1\ \texttt{catch}\ ex()\ e_2 \Rightarrow \infty} \qquad \frac{S, e_1 \Rightarrow S', ex(v) \qquad v :: \Gamma, S', e_2 \Rightarrow \infty}{S, \text{try } e_1\ ex\ e_2 \Rightarrow \infty}$$

$$\frac{\Gamma_f := [[\![t_1]\!]_{\Gamma,S}, ..., [\![t_n]\!]_{\Gamma,S}] \qquad [\![\mathsf{pre}_f]\!]_{\Gamma_f,S} \qquad \Gamma_f, S, \mathsf{body}_f \Rightarrow \infty}{\Gamma, S, \text{Ecall } f\ (t_1, ..., t_n) \Rightarrow \infty}$$

Fig. 7. Operational semantics of non-terminating expressions

on states which replaces the topmost mapping for r in S. A labeled expression is evaluated in an enriched state $S\!\uparrow$ where the current state is copied on top of the vector. The resulting state $S'\!\downarrow$ is obtained by deleting the second position of the vector what corresponds to "forget" the previously copied current state. The rule for function calls requires the pre-condition to be valid in the starting state and, if the function terminates normally, the validity of the post-condition in the returning state to be valid too.

Judgement $\Gamma, S, e \Rightarrow \infty$ expresses that the execution of expression e does not terminate in environment Γ and state S. Its definition is straightforward: the execution of an expression diverges if the execution of a sub-expression diverges. The interesting cases are for the execution of a loop: starting from a given state S, it diverges either if its body diverges or if its body terminates on some state S' and the whole loop diverges starting from this new state. Of course, non-termination may be caused by infinite recursion of functions, too.

The main feature to notice is that execution blocks whenever an invalid assertion is met: the rules for assertions, loops and function calls are applicable only if the respective annotations are valid. Conversely, as everything is well-typed by construction, the only reason why an expression wouldn't execute is that one of its annotations isn't respected.

Definition 2 (Safe execution). *An expression e* executes safely *in environment Γ and state S, denoted $\Gamma, S, e \overset{\text{safe}}{\Rightarrow}$, if either it diverges: $\Gamma, S, e \Rightarrow \infty$, or it terminates: $S', o,\ \Gamma, S, e \Rightarrow S', o$.*

A program respects its annotations *if for each function f and any Γ, S such that $[\![pre_f]\!]_{\Gamma,S}$ we have $\Gamma, S, body_f \overset{\text{safe}}{\Rightarrow}$ and if $\Gamma, S, body_f \Rightarrow S', o$ then o is a normal outcome v such that $[\![post_f]\!]_{v::\Gamma,S'}$.*

$$\begin{aligned}
\mathsf{WP}\ (\text{Eterm}\ t)\ Q\ R\ \Gamma\ S &= Q\ S\ [\![t]\!]_{\Gamma,S}\\
\mathsf{WP}\ (\text{Elet}\ e_1\ e_2)\ Q\ R\ \Gamma\ S &= \mathsf{WP}\ e_1\ (\lambda S\ a,\ \mathsf{WP}\ e_2\ Q\ R\ (a :: \Gamma)\ S)\ R\ \Gamma\ S\\
\mathsf{WP}\ (\text{Eassign}\ r\ t)\ Q\ R\ \Gamma\ S &= Q\ (S[r/[\![t]\!]_{\Gamma,S}])\ [\![t]\!]_{\Gamma,S}\\
\mathsf{WP}\ (\text{Eassert}\ p\ e)\ Q\ R\ \Gamma\ S &= [\![p]\!]_{\Gamma,S} \land \mathsf{WP}\ e\ Q\ R\ \Gamma\ S\\
\mathsf{WP}\ (\text{Eraise}\ ex\ t)\ Q\ R\ \Gamma\ S &= R\ S\ ex\ [\![t]\!]_{\Gamma,S}\\
\mathsf{WP}\ (\text{Eif}\ p\ e_1\ e_2)\ Q\ R\ \Gamma\ S &= ([\![p]\!]_{\Gamma,S} \to \mathsf{WP}\ e_1\ Q\ R\ \Gamma\ S)\\
&\quad \land (\neg[\![p]\!]_{\Gamma,S} \to \mathsf{WP}\ e_2\ Q\ R\ \Gamma\ S)\\
\mathsf{WP}\ (\text{Eloop}\ p\ e)\ Q\ R\ \Gamma\ S &= [\![p]\!]_{\Gamma,S} \land \forall S', S \overset{\text{writes}\ e}{\rightsquigarrow} S' \to [\![p]\!]_{\Gamma,S'} \to\\
&\qquad \mathsf{WP}\ e\ (\lambda S''\ v,\ [\![p]\!]_{\Gamma,S''})\ R\ \Gamma\ S'\\
\mathsf{WP}\ (\text{Etry}\ e_1\ ex\ e_2)\ Q\ R\ \Gamma\ S &= \mathsf{WP}\ e_1\ Q\ (\lambda S'\ ex'\ a,\ \mathsf{if}\ ex = ex'\\
&\qquad \mathsf{then}\ \mathsf{WP}\ e_2\ Q\ R\ (a :: \Gamma)\ S'\ \mathsf{else}\ R\ ex'\ a)\ \Gamma\ S\\
\mathsf{WP}(\text{Elabel}\ e)\ Q\ R\ \Gamma\ S &= \mathsf{WP}\ e\ Q\ R\ \Gamma\ S\!\uparrow\\
\mathsf{WP}\ (\text{Ecall}\ f\ (t_1, ..., t_n))\ Q\ R\ \Gamma\ S &= \ [\![pre_f]\!]_{\Gamma_{args},S} \land \forall S'\ a, S \overset{\text{writes}\ f}{\rightsquigarrow} S' \to\\
&\qquad [\![post_f]\!]_{(a::\Gamma_{args}),(S',S)} \to Q\ S'\ a\\
&\qquad \text{where}\ \Gamma_{args} := [[\![t_1]\!]_{\Gamma,S}, ..., [\![t_n]\!]_{\Gamma,S}]
\end{aligned}$$

Fig. 8. Recursive definition of the WP-calculus

Our semantics is quite unusual, in particular it is not executable. Although, if annotations are removed then it becomes executable (indeed only if the propositional guards in if-then-else blocks are decidable) and coincides with a natural semantics. This approach makes obsolete a distinct set of rules for axiomatic semantics *à la* Hoare: the soundness of the verification condition generator will be stated using this definition of safe execution. Moreover this notion of safe execution is indeed stronger than the usual notion of partial correctness: a safe program that does not terminate will still satisfy its annotations forever.[1]

4 Weakest Precondition Calculus

We calculate the weakest pre-condition of an expression given a post-condition by structural recursion over expressions (Fig. 8). We admit several post-conditions, $Normal_{L,A} : state_L \to dentype_A \to \mathsf{Prop}$ for regular execution and $Exceptional_L : state_L \to \forall B, exn_B \to dentype_B \to \mathsf{Prop}$ for exceptional behavior. So our calculus has the type $\mathsf{WP} : e_{L,E,A} \to Normal_{L,A} \to Exceptional_L \to env_E \to state_L \to \mathsf{Prop}$. In the case of a loop, the pre-condition is calculated using the loop invariant and in the case of a function call we use the pre- and post-condition of that function. In both cases, as it is classical in WP calculi, we need to quantify over all states that may be reached by normal execution starting

[1] Total correctness is not considered in this paper; however it is clear that one could add annotations for termination checking: variants for loops and for recursive functions as in ACSL [4].

from the given state S: these are the states S' which differ from S only for the references that are modified in the loop or the function's body. This is denoted as $S \overset{s}{\leadsto} S' := \forall r : ref_A, r \notin s \rightarrow (Here\ S'\ r = Here\ S\ r \wedge \forall l, At\ l\ S'\ r = At\ l\ S\ r)$. The function *writes* computes the references modified by some expression, it is shown correct [16] in the sense that if $\Gamma, S, e \Rightarrow S', o$ then $S \overset{writes\ e}{\leadsto} S'$.

The verification conditions, respectively for one function and for a whole program, are

$$\begin{aligned} \mathsf{VC}(f) &:= [\![pre_f]\!]_{\Gamma,S} \rightarrow \mathsf{WP}\ body_f\ (\lambda S'\ v, [\![post_f]\!]_{v::\Gamma,S'})\ (\lambda S'\ ex\ v, \mathsf{False})\ \Gamma\ S \\ \mathsf{VCGEN} &:= \forall f : idx_{\langle A,P\rangle,F}\ \Gamma\ S,\ \mathsf{VC}(f) \end{aligned}$$

The False as exceptional post-conditions requires that no function body exits with an exception.

We now state that if the VCs hold for all functions then any expression having a valid WP executes safely. It is proved by co-induction, using the axiom of excluded middle to distinguish whether the execution of an expression does or does not terminate, following the guidelines of Leroy and Grall [20]. Notice that it is enough to prove the verification conditions for each function separately, even if functions can be mutually recursive, there is no circular reasoning.

Lemma 1 (safety of expressions). *If VCGEN holds then for any Γ, S, e, Q, R, if ($WP\ e\ Q\ R\ \Gamma\ S$) then $\Gamma, S, e \overset{safe}{\Rightarrow}$.*

The important corollary below states that if the VCs hold for all functions then any of their bodies execute safely. By definition of the semantics, this implies that all assertions, invariants and pre- and post-conditions in a given program are verified if the verification conditions are valid.

Theorem 1 (soundness of VCGEN). *If VCGEN holds then the program respects its annotations, as defined in Def. 2*

5 Extraction of a Certified Verification Tool

The obtained Coq function for generating verification conditions is not *extractable*: given a program pg we obtain a Coq term ($\mathsf{VCGEN}\ pg$) of Coq type Prop which must be proved valid to show the correctness of the program. The process thus remains based on Coq for making the proofs. In this section we show how to extract the calculus into a separate tool so that proofs can be performed with other provers, e.g. SMT solvers.

5.1 Concrete WP Computation

To achieve this we need the WP calculus to produce a formula in the abstract syntax of Fig. 2 instead of a Coq Prop. We define another function

$$\mathsf{wp} : e_{L,E,A} \rightarrow p_{L,A::E} \rightarrow (exn_B \rightarrow p_{L,B::E}) \rightarrow p_{L,E}$$

which, given an expression e, a normal post-condition Q and a family of exceptional post-conditions R, returns a weakest pre-condition. It is defined recursively on e similarly to WP in Fig. 8, but this time Q, R and the result are syntactic propositions which are concretely transformed (see [16]).

Lemma 2. *If* $[\![\mathsf{wp}\ e\ Q\ R]\!]_{\Gamma,S}$ *then*

$$\mathit{WP}\ e\ \ (\lambda S\ v, [\![Q]\!]_{v::\Gamma,S})\ \ (\lambda S\ ex\ v, [\![R\ ex]\!]_{v::\Gamma,S})\ \ \Gamma\ S$$

From wp we now define a concrete verification-condition generator vcgen.

Definition 3. *The concrete VCs of a program pg is the list* (vcgen *pg*) *of concrete formulas* (Abstr(Pimply pre_f (wp body_f post_f Pfalse))) *for each function f of pg. Abstr is a generalization function: it prefixes any formula* $p_{L,E}$ *by as many* Pforall *as elements of E to produce a* $p_{L,[]}$ *formula.*

Theorem 2. *If for all p in the list* (vcgen *pg*) *and for all state S,* $[\![p]\!]_{[],S}$ *then* (*VCGEN pg*).

That is, the hypothesis of Theorem 1 is valid if we prove the formulas generated by vcgen valid in any state.

5.2 Producing Concrete Syntax with Explicit Binders

Still, formulas of vcgen are represented by a de Bruijn-style abstract syntax. To print out such formulas we need to transform them into concrete syntax with identifiers for variables by generating new names for all the binders. This could be done on the fly in an unproven pretty-printer. Though, being a non trivial transformation it is better to do it in a certified way directly after the generation.

We therefore formalize a back-end syntax, along with its semantics for well-typed terms and propositions. It is similar to Fig. 2 where we replace Tconst, Tvar and Tderef by a new constructor Tvar with an identifier as argument, and Tlet and Pforall binders are also given an explicit identifier. We define a compilation from *de Bruijn*-style terms and propositions to the back-end syntax and prove preservation of semantics.

Finally, we define a *proof task* as a triple (d, h, g) where d is a finite map from identifiers to their type, h is a set of hypotheses and g is a list of goals. Such a task is said valid if the goals are logical consequences of the hypotheses, whatever the interpretation of symbols in d. The complete process of VC generation is to produce, from a logical context C and a program pg, the proof task $T(C, pg) = (d, h, g)$ where d are the declarations of C that appear in pg, h the compilation of axioms of C, and g is the compilation of vcgen(pg).

Theorem 3 (Main soundness theorem). *For all logical context C and program p, if the proof task T(C, p) is valid then for any interpretation of the context in which the axioms are valid, p executes safely.*

Notice that this statement is independent of the underlying proof assistant Coq: the validity of logical formulas in the proof task can be established by any theorem prover. The only hypothesis is that the backend theorem prover in use must agree with our definition of the interpretation of logical contexts. But this is just the classical first-order logic with equality, with standard predefined theories like integer arithmetic. All the off-the-shelf theorem provers, e.g SMT solvers, agree on that.

5.3 Extraction and Experimentation

For experimentation purposes we also defined a compilation in the opposite direction, i.e. from programs in front-end syntax to the corresponding program in *de Bruijn* syntax, provided that the former is well typed. We then use the extraction mechanism of Coq to extract an Ocaml function that, given an AST of our front-end syntax representing a program, produces a list of ASTs representing the proof task. We finally combine this with the Why3 parser for input programs and a hand-written pretty-printer that produces Why3 syntax [6], allowing us to call automated provers on the proof task.

We made experiments to validate this process. On our selection sort example, the two VCs for functions `swap` and `selection_sort` are generated in a fraction of a second by the standalone VC generator. These are sent to the Why3 tool, and they are proved automatically, again in a fraction of a second, by a combination of SMT solvers (i.e. after splitting these formulas, which are conjunctions, into parts [6]). For details see the Coq development at the URL given in the introduction.

6 Conclusions, Related Works and Perspectives

We formalized a core language for deductive verification of imperative programs. Its operational semantics is defined co-inductively to support possibly non-terminating functions. The annotations are taken into account in the semantics so that validity of a program with respect to its annotations is by definition the progress of its execution. We used an original formalization of binders so that only well-typed programs can be considered, allowing us to simplify the rest of the formalization. Weakest precondition calculus is defined by structural recursion, even in presence of mutually recursive functions, assuming the given function contracts. Even if there is an apparent cyclic reasoning, this approach is shown sound by a co-inductive proof. By additionally formalizing an abstract syntax for terms and formulas, and relating their semantics with respect to the Coq propositions, we defined a concrete variant of the WP calculus which can be extracted to OCaml code, thus obtaining a trustable and executable VC generator close to Why or Boogie.

As explained in the introduction, two kinds of approaches for deductive verification exist depending on the use of a deep embedding of the programming language or not. The approaches without deep embedding typically allows the user to discharge proof obligations using automatic provers, but are not certified correct. Our work fills this gap. Among deep-embedding-based approaches, the SunRise system of Homeier et al. [17,18] is probably the first certified program verifier, and uses a deep embedding in the HOL proof environment. They formalize a core language and its operational semantics, and prove correct a set of Hoare-style deduction rules. Programs are thus specified using HOL formulas and proved within the HOL environment. Later Schirmer [26] formalized a core language in Isabelle/HOL, and Norrish formalized the C programming language [24], with similar approaches. More recently, similar deep-embedding-based approaches were proposed using Coq like in the Ynot system [23,9], which can deal with "pointer" programs via separation logic, and also supports higher-functions.

A major difference between the former approaches and ours is that we use a deep embedding not only for programs but also for propositions and thus for specifications. This allows us to extract a standalone executable, and consequently to discharge VCs using external provers like SMT solvers. Our approach is a basis to formalize specification languages like JML and ACSL defined on top of mainstream programming language, which allows a user to specify and prove Java or C programs without relying on the logic language of a specific proof assistant.

Another difference is that we do not consider any Hoare-style rules but formalize a Dijkstra-style VC generator instead. This way to proceed is motivated by the choice of defining the meaning of "a program satisfies its annotations" by safety of its execution.

There are also some technical novelties in our approach with respect to the systems mentioned above. Our core language supports exceptions, which is useful for handling constructs of front-ends like `break` and `continue`, or Java exceptions. Specifications can also use labels to refer to former states of executions, with constructs like `\old` and `\at` constructs of JML and ACSL. This provides a handy alternative to the so-called *auxiliary* or *ghost* variables used in deep-embedding-based systems above. Indeed in the context of VC generation instead of Hoare-style rules, the semantics of such variables is tricky, e.g. when calling a procedure, the ghost variables should be existentially quantified, which results in VCs difficult to solve by automated provers. We believe that the use of labels is thus better.

Our main future work is to certify the remaining part of a complete chain from ACSL-annotated C programs to proof obligations. A first step is the formalization of a front-end like Frama-C/Jessie which compiles annotated C to intermediate Why code. We plan to reuse the C semantics defined in CompCert [19] and incorporate ACSL annotations into it. The main issue in this compilation process is the representation of the C memory heap by Why global references using a memory heap modeling. In particular, first-order modeling of the heap, mainly designed to help automatic provers, raised consistency problems in the past [27]. In our approach where the axioms of the logical context are realized in Coq, the consistency is guaranteed. Finally, another part of the certification of the tool chain is the certification of back-end automatic provers, for which good progress was obtained recently, see e.g. [21].

Acknowledgments. We would like to thanks Jean-Christophe Filliâtre and Julien Signoles for their remarks on a preliminary version of this paper.

References

1. Abrial, J.-R.: The B-Book, assigning programs to meaning. Cambridge University Press (1996)
2. Barnett, M., DeLine, R., Jacobs, B., Chang, B.-Y.E., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.4 (2009), `http://frama-c.cea.fr/acsl.html`
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer, Heidelberg (2004)
6. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011)
7. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) 7(3), 212–232 (2005)
8. Chlipala, A.: Certified Programming with Dependent Types. MIT Press (2011), `http://adam.chlipala.net/cpdt/`
9. Chlipala, A.J., Malecha, J.G., Morrisett, G., Shinnar, A., Wisnesky, R.: Effective interactive proofs for higher-order imperative programs. In: Hutton, G., Tolmach, A.P. (eds.) ICFP, pp. 79–90. ACM, Edinburgh (2009)
10. Cok, D.R., Kiniry, J.R.: ESC/Java2: Uniting eSC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
11. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: VCC: Contract-based modular verification of concurrent C. In: 31st International Conference on Software Engineering Companion, ICSE 2009, Vancouver, Canada, May 16-24, pp. 429–430. IEEE Comp. Soc. Press (2009)
12. Filliâtre, J.-C.: Verification of non-functional programs using interpretations in type theory. Journal of Functional Programming 13(4), 709–745 (2003)
13. Filliâtre, J.-C., Marché, C.: Multi-Prover Verification of C Programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
14. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)

15. Herms, P.: Certification of a chain for deductive program verification. In: Bertot, Y. (ed.) 2nd Coq Workshop, Satellite of ITP 2010 (2010)
16. Herms, P., Marché, C., Monate, B.: A certified multi-prover verification condition generator. Research Report 7793, INRIA (2011), `http://hal.inria.fr/hal-00639977/en/`
17. Homeier, P.V., Martin, D.F.: A mechanically verified verification condition generator. The Computer Journal 38(2), 131–141 (1995)
18. Homeier, P.V., Martin, D.F.: Mechanical Verification of Mutually Recursive Procedures. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 201–215. Springer, Heidelberg (1996)
19. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009)
20. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Inf. Comput. 207, 284–304 (2009)
21. Lescuyer, S.: Formalisation et développement d'une tactique réflexive pour la démonstration automatique en Coq. Thèse de doctorat, Université Paris-Sud (2011)
22. Marché, C., Paulin-Mohring, C., Urbain, X.: 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`
23. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: Reasoning with the awkward squad. In: Proceedings of ICFP 2008(2008)
24. Norrish, M.: C Formalised in HOL. PhD thesis, University of Cambridge (November 1998)
25. Randimbivololona, F., Souyris, J., Baudin, P., Pacalet, A., Raguideau, J., Schoen, D.: Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1798–1815. Springer, Heidelberg (1999)
26. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)
27. Wagner, M., Bormer, T.: Testing a verifying compiler. In: Beckert, B., Marché, C. (eds.) Formal Verification of Object-Oriented Software, Papers Presented at the International Conference, Karlsruhe Reports in Informatics, Paris (2010), `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000019083`