# A Generic Intermediate Representation for Verification Condition Generation *

Manuel Montenegro Ricardo Peña Jaime Sánchez-Hernández
montenegro@fdi.ucm.es {ricardo,jaime}@sip.ucm.es

Universidad Complutense de Madrid, Spain

**Abstract.** As part of a platform for computer-assisted verification, we present an intermediate representation of programs that is both language independent and appropriate for the generation of verification conditions. We show how many imperative and functional languages can be translated to this generic intermediate representation, and how the generated conditions reflect the axiomatic semantics of the original program. At this representation level, loop invariants and preconditions of recursive functions belonging to the original program are represented by assertions placed at certain edges of a directed graph.
The paper defines the generic representation, sketches the transformation algorithms, and describes how the places where the invariants should be placed are computed. Assuming that, either manually or assisted by the platform, the invariants have been settled, it is shown how the verification conditions are generated. A running example illustrates the process.

**Key words:** verification platforms, intermediate representation, verification conditions, program transformation.

## 1 Introduction

In the last few years, verification platforms are becoming more and more popular [15, 1, 10]. Their success is in part due to the increasing power of the underlying proving machinery, the SMT solvers [7, 8]. In these platforms, the user is responsible for giving the source program, its specification in the form of a precondition and a postcondition, and the invariant assertion of each loop. The platform gives support for analysing and proving termination, for generating the verification conditions (VC), and for automatically proving them, whenever this is possible.

A possible drawback is that the source language is usually fixed by the platform and it consists of a restricted subset of a real-life one. For instance, Dafny supports object-oriented programming but not inheritance. WhyML does not support object orientation, nor even has a heap.

The purpose of our platform CAVI-ART[1] is a bit more ambitious. On the one hand, it addresses real-life languages and will support most of, or ideally all, their

* Work partially supported by the Spanish MINECO project CAVI-ART (TIN2013-44742-C4-3-R), Madrid regional project N-GREENS Software-CM (S2013/ICE-2731) and UCM grant GR3/14-910502

[1] CAVI-ART stands for Computer Assisted ValIdation by Analysis, tRansformation and Testing.

complexities and subtleties. Additionally, it will cover both imperative, possibly object-oriented ones, such as C, C++ and Java, and functional ones such as Erlang, SML and Haskell. On the other hand, the platform will assist the user in discovering the loop invariant assertions, or equivalently, the preconditions of the recursive functions. The remaining platform assistance will be similar to that of the other platforms. In fact, we plan to reuse the infrastructure of Why3 to interface different SMTs and proof assistants, by expressing our VCs in the Why3 assertion language.

In Fig. 1 we show a picture of the whole project. A key aspect of it is designing an intermediate representation (IR) of programs to which source programs, written in a variety of languages, can be transformed. Once programs have undergone this transformation, the remaining activities —invariant synthesis, termination analysis, VC generation, VC proving— can be performed in a language-independent way. This transformation yields an abstraction of the control and data flow of the program that relies on a set of language-dependent primitive functions, which are defined via axioms and can be reused among different languages. Moreover, some of them are already present in Why3's standard library of theories, which includes definitions of integers, lists, arrays, real numbers, etc. and their associated functions.

The platform is under construction. We have completed the design, the IR, and a front-end for Java. Our current work mainly focuses on invariant synthesis. In this paper we describe such a generic IR, and show how VCs can be generated from it, guaranteeing that should all the VCs be discharged by the provers, then the original program satisfies all assertions. A key step in mapping imperative programs to the IR is transforming iteration to recursion, so that both are dealt with uniformly. A second step is to detect where invariant assertions would be needed in the resulting IR. Once these assertions have been provided, either by the user or by the platform itself, the VC generation is done automatically.

The plan of the paper is as follows: in Sect. 2 we describe the transformation of several imperative features such as primitive and structured types, classes, and the heap to a common framework. Then we explain how to abstract the control by generating a Control Flow Graph (CFG). In Sect. 3, we briefly remind how functional languages are compiled to a small core representation, which usually is a slight extension of the $\lambda$-calculus. In Sect. 4, we present and justify our IR, and give an axiomatic semantics to it by means of weakest preconditions. Sect. 5 describes the algorithm transforming the CFG to the IR, and detects the locations of the invariants. Sect. 6 explains the VC extraction algorithm. Finally, Sect. 7 draws some conclusions and reviews the related work.

## 2 Imperative Languages

The computation model of imperative languages is given by the execution of a sequence of statements that change the state of the program. Among the diversity of the features provided by modern imperative languages (such as Java, Javascript, Python, etc.) there are two which are shared by most of them: destructive assignment and explicit management of control flow. However, languages differ in the kind of basic values that can be assigned to a variable, and

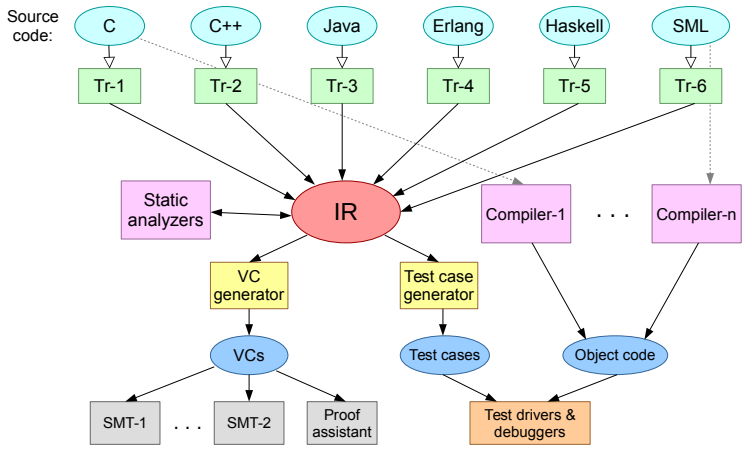


**Fig. 1.** CAVI-ART project overview

the choice of control flow constructions (loops, exceptions, method calls, delegates, etc.) In the following, we shall abstract their common parts in order to determine the constructions needed by the IR. We also identify the language-specific components, so that the IR will be parametric on them.

*Example 1.* As a running example, let us consider in this paper the following Java implementation of the insertion sort algorithm:

```
1  public void insertionSort(int[] v) {
2    for (int i = 0; i < v.length; i++) {
3      int e = v[i];
4      int j = i - 1;
5      while ((j >= 0) && (v[j] > e)) {
6        v[j+1] = v[j];
7        j = j - 1;
8      }
9      v[j+1] = e;
10   }
11 }
```

*Basic values* For each language we identify its set of basic values. We classify them into different categories, which will subsequently be mapped to theory types of the underlying proof system.

For a given language, we consider a set of *value categories* $\{\beta_1, \ldots, \beta_n\}$, each one is a pair $\langle B_\beta, \equiv_\beta \rangle$, where $B_\beta$ is the set of values contained within the category $\beta$, and $\equiv_\beta$ is an equivalence relation on these values. This relation is necessary for performing case distinction on the values at the IR level. For every language, we assume the existence of a category $\beta_{\mathbf{Bool}}$ with the set $B_{\mathbf{Bool}} = \{true, false\}$ and the usual equivalence relation.

For instance, we use in Java the set of types given by the semantics of *Jinja* [14]: booleans, integers, pointers, null reference and unit type. We also include the category of floating point numbers and arrays, since some solvers (e.g. Z3 [7]) provide direct support for them.

*Built-in operators and functions* This is another language-dependent component. We encode them in the IR as functions whose behaviour is defined by a set of

axioms. Therefore, for each programming language we define the set of primitive functions and axioms. Both can be specified in terms of already existing theories.

In the case of the translation from Java into the IR, several primitive functions are based on their counterparts defined in the Why3 Standard Library. For instance, we associate the category $\beta_{int}$ of integer values with the `int` type defined within Why3's `Int` theory. The integer-based operators (such as `<=`, `==`, `+`, etc.) are mapped into its corresponding counterparts in this theory. An analogous association is made with booleans and real numbers. Arrays are also translated into the type `array` defined within the `Array` theory of Why3, defined as follows:

```
type array 'a model { length : int; mutable elts : map int 'a }
  invariant { 0 <= self.length }
```

The definition of the built-in operations on arrays is more involved, since a simple access to an array may result in a `NullPointerException` or an `ArrayIndexOutOfBoundsException`. We consider two different policies:

- **Safe array access assumption**. The built-in function `sel-array` has a precondition asserting that the array is not null and that the index lies within the bounds. If this holds, then the selection yields a valid result.

  $\{\mathcal{H}(p) = Array\ a \wedge 0 \leq i < a.length\}$ `sel-array(`$\mathcal{H}$`, p, i)` $\{res =$ `get` $a\ i\}$

  In this specification the $\mathcal{H}$ denotes a heap, $p$ a heap location, and `get` denotes the actual array access function defined in the Why3 library. In a similar way we define `mod-array`, which yields the heap resulting from modifying an array in a given position.
- **Array access with exceptions**. We extend the specification of `sel-array` by considering the possibility that the array access may throw an exception. However, since exception handling is considered as a language-dependent feature, exceptions should not be part of the IR. Exception management is handled with a special type which is similar to the `Either` type of Haskell:

  ```
  data opt_result = Ok value | Exception loc
  ```

  In this definition `value` denotes a union type for basic values, and `loc` is the type of heap locations. Both definitions are language-dependent.

The first policy is simpler, and it works if the prover can establish the validity of all array accesses contained within the method. If it cannot, the correctness of the method is not proved. With the second policy the postconditions of the method can be more precise and assert facts regarding exceptions (for instance, the reasons of an exception being thrown), but makes the resulting IR code more complicated. For the sake of simplicity, we consider the first policy in our running example.

*Heap management* The presence of a mutable memory heap plays an essential role in imperative programs. As a consequence of its physical representation in the memory, virtually all languages consider a heap $\mathcal{H}$ as a mapping from locations to values. The language-dependent element here is the kinds of values represented in the heap. In Java we follow the approach of [14] (extended with array values) and define the set of heap values as follows:

$$
\begin{array}{lll}
\mathit{blk} & ::= \mathit{stm}_1; \ldots; \mathit{stm}_n; \mathit{jump} & \{ \text{ instruction BB } \} \\
& \mid \ \textbf{return}\ x & \{ \text{ exit BB, } x \text{ is a variable } \} \\
\mathit{stm} & ::= x = e & \{ \text{ assignment (single variable) } \} \\
& \mid \ (x_1, \ldots, x_n) = e & \{ \text{ assignment (several variables) } \} \\
\mathit{jump} & ::= \textbf{case}\ x\ \textbf{of}\ c_1 \to n_1 \ldots c_m \to n_m & \{ \text{ conditional jump } (n_1, \ldots, n_m \in \mathbb{N}) \} \\
& \mid \ \textbf{goto}\ n & \{ \text{ unconditional jump } (n \in \mathbb{N}) \} \\
e & ::= a & \{ \text{ atom } \} \\
& \mid \ f(a_1, \ldots, a_n) & \{ \text{ function application } \} \\
a & ::= c & \{ \text{ literal } \} \\
& \mid \ x & \{ \text{ variable } \}
\end{array}
$$

**Fig. 2.** Structure of CFG blocks.

```
data heap_value = Array (array value)
                | Object string (map (string, string) value)
```

where an object instance contains a class name and a map from pairs $(p, c)$ to pointers. In these pairs $p$ denotes the name of an attribute and $c$ the name of the class to which the attribute belongs.

In order to specify heap modifications we follow the same approach as in the previous section; they are managed as language-dependent built-in functions, each one with a formal specification via pre- and post-conditions. Therefore every heap-related operation subject to axiomatization, such as method calls, dynamic dispatch, etc. can be used in the IR. Since we avoid the existence of a mutable state, the operations modifying the heap are pure, in the sense that they yield another heap with the corresponding changes.

*Control flow* In order to handle this feature in a language independent way, the source program is transformed into a *control-flow graph* representation (CFG) [2]. In this graph each node is a *basic block* (BB) containing a sequence of program instructions without jumps between them (except calls to other functions or methods).

The information inside a BB is defined by the grammar given in Fig. 2. A BB can be an exit block (**return**) or contain a sequence of basic statements followed by a jump instruction. Statements are assignments whose right-hand side can be a literal, variable or a function application. In the latter case, only atomic arguments are allowed. This requires a flattening transformation on the original program and the addition of new assignments. Jump instructions refer to other BBs in the CFG, each of which is identified by a natural number. Thus, a CFG is a set of numbered BBs $\{(n_1, \mathit{blk}_1), \ldots, (n_r, \mathit{blk}_r)\}$. We can attach to each CFG an assertion which must be satisfied by every execution of the function being analysed. This is useful for specifying loop invariants.

*Example 2.* The transformation of our insertion sort example into a CFG yields the result shown in Fig. 3, where array accesses and basic operations have been replaced by flattened function calls. A new variable $\mathcal{H}$ is introduced to denote explicitly the heap.

Our next step is translating the CFG of the input program into a set of mutually recursive functions, from which the verification conditions will be extracted. In order to obtain a set of functions, we dispose of destructive assignment by

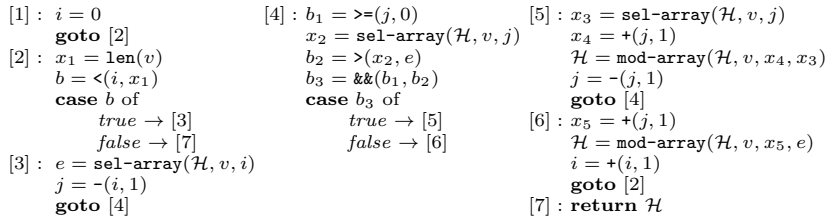


**Fig. 3.** CFG blocks of the *insertionSort* algorithm.

transforming our program into *Static Single Assignment* form (SSA) [3]. After this, each program variable is assigned exactly once, and subsequent assignments are done to different *versions* of the variable, each one having a different name. In our case, the SSA transformation is applied locally to each BB. Instead of having $\phi$ functions in confluence nodes (as usual in SSA), the transformation performs a liveness analysis at the beginning of each node. Let $LV_i$ be the set of live variables at node $i$ (before applying SSA transformation). After applying the local transformation to each node, we have to compute, for every node $j$ pointing to $i$, a substitution $\theta_{j,i}$ that maps each variable $x \in LV_i$ to the last version of that variable occurring in $j$.

*Example 3.* The liveness analysis on the CFG of Example 2 produces:

$$LV_1 = \{v, \mathcal{H}\} \quad LV_3 = \{v, i, \mathcal{H}\} \quad LV_5 = \{v, i, j, e, \mathcal{H}\} \quad LV_7 = \{\mathcal{H}\}$$
$$LV_2 = \{v, i, \mathcal{H}\} \quad LV_4 = \{v, i, j, e, \mathcal{H}\} \quad LV_6 = \{v, i, j, e, \mathcal{H}\}$$

The translation of each BB into SSA leads to the CFG represented in Fig. 4, in which the left-hand sides $\mathcal{H}$ and $j$ of the BB [5] have been replaced by $\mathcal{H}_1$ and $j_1$, respectively. The corresponding mapping from [5] to [4] would be $\theta_{5,4} = [v \mapsto v, i \mapsto i, j \mapsto j_1, e \mapsto e, \mathcal{H} \mapsto \mathcal{H}_1]$.

After this transformation, we translate each BB $i$ of the CFG into a recursive function receiving as arguments the variables in $LV_i$. Its definition is the sequence of BB statements, and the jump branches are calls to the adjacent BBs by using the respective substitutions. This will be shown in Sect. 5.

## 3 Functional Languages

Functional languages are radically different to imperative ones, as they provide neither destructive variable assignment, nor control flow management. There is no notion of state, as it is the case in the imperative paradigm. Their main features are pattern matching for function definition arguments, higher-order functions, recursive definitions for data types and functions, and lambda abstractions. In addition, some of them are polymorphic and strongly typed, and provide a type inference algorithm (Haskell, ML). A few of them are lazy (Haskell).

From a theoretical point of view, functional languages emerge from the $\lambda$-calculus, which in fact can be seen as a minimalistic core language for all of them. But from a practical point of view, the core for real functional languages

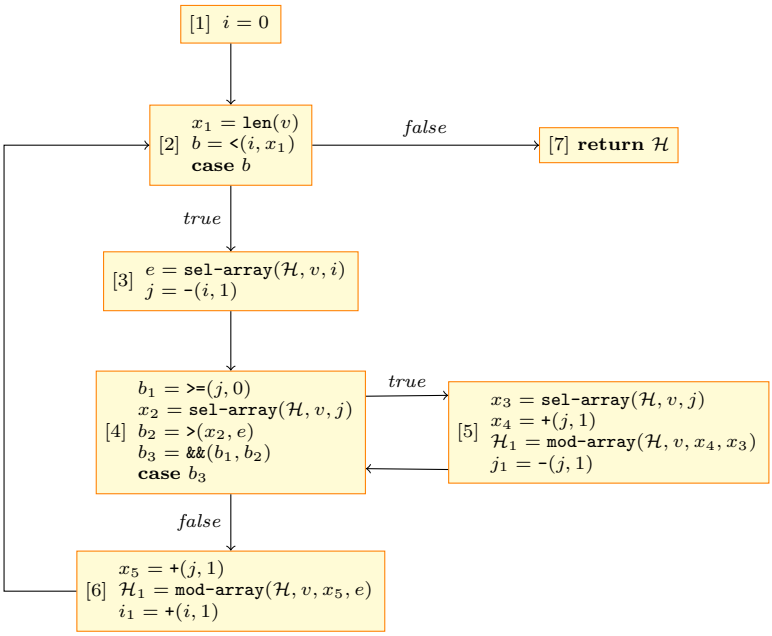


**Fig. 4.** Representation of the CFG/SSA of the insertion sort algorithm

usually includes constructor application, **let**-expressions for local definitions, recursive **letrec**-expressions, and **case**-expressions, a normalized form of pattern matching. This is the case of the enriched lambda-expressions used in [16], and also in the core languages of the Glasgow Haskell Compiler [18], ML [17] and Erlang [4].

Sometimes it may be useful to enrich the core syntax in order to facilitate the compiler code generation, or to reduce it in order to simplify formal reasoning. For example, $\lambda$-abstractions can be removed from the core with the well-known *lambda lifting* transformation [16], which transforms $\lambda$-abstractions into ordinary (named) functions. Also, the applicative notation can be flattened in order to avoid complex nesting of expressions. Moreover, nested pattern matching can be compiled in such a way [16], that it is converted in a sequence of nested **case**-expressions, each one with flat and mutually exclusive patterns, and covering all datatype constructors.

## 4 The Intermediate Representacion

From the precedent sections, it is clear that a minimal (core) functional language can serve both to represent imperative programs which have undergone an SSA transformation and functional programs which have been previously desugared. The minimal common elements of this core language are the followings:

- Sequential **let** expressions, which also represent imperative SSA assignments.
- Recursive **letrec** expressions, needed to define mutually recursive functions.
- $\lambda$-abstractions and applications, needed to define and apply functions.

```
a ->                                -- atom
    c                               -- constant
  | x                               -- variable
ae ->                               -- atomic expression
    a                               -- atom
  | f a1 ... an                     -- primitive operator/function application
  | C a1 ... an                     -- constructor application
e ->                                -- structured expression
    ae                              -- atomic expression
  | let p = ae in e                 -- sequential let
  | letfun fd1 ... fdn in e         -- function definition block
  | case a of alt1 ... altn         -- algebraic type or primitive type case
            [ _ -> e]               -- optional default clause
alt ->                              -- case alternative
    C x1 ... xr -> e                -- algebraic type alternative
    c           -> e                -- primitive type alternative
p ->                                -- pattern
    x                               -- variable pattern
  | (x1,...,xn)                     -- tuple pattern
fd -> f x1 ... xn = e               -- function definition. The name f is global
```

**Fig. 5.** Abstract syntax of the CAVI-ART Intermediate Representation

- **case** expressions, which can serve both to mimic imperative **switch** statements and to express functional pattern matching.

In addition to this, imperative languages need support for structured data types such as arrays and records, and functional languages need support for algebraic data types, polymorphism, and higher-order. Taking all this into account, we have defined an intermediate representation (IR) that gives support to most of the features one can find in imperative and functional languages. In Fig. 5 we show the abstract syntax of our IR.

We justify some of the decisions leading to this IR. Firstly, the arguments of applications and **case** discriminants are restricted to be atoms. This facilitates the renaming of predicate arguments when propagating assertions, and also makes the definition of weakest preconditions for the **case** construction simpler. We make note that an **if** construction is not needed as it is a particular instance of **case**. Secondly, function definitions are confined to be in a **letfun** expression, and they are by default mutually recursive. A **letfun** can be considered as syntactic sugar for a functional **letrec** expression in which each variable is bound to a lambda abstraction. Thirdly, expressions are in the so-called A-normal form [11]. In particular, this implies that in **let** bindings, applications occur as stand alone expressions. Finally, **case** patterns are flat and they exclude each other, so that only one alternative is possible. If a **case** does not include an alternative for each constructor, it necessarily has a default clause. The purpose of all these restrictions is again to facilitate the definition of weakest preconditions and the generation of verification conditions.

The IR is strongly typed and the type system is polymorphic in a Hindley-Milner style, similar to that of the logical language Why3 [9] in which the assertions are expressed. This type system supports both polymorphic functional languages such as SML and Haskell, untyped functional languages such as Erlang, monomorphic imperative languages such as C, and polymorphic imperative languages such as C++ or Java.

Arrays and records are not built-in data types of the IR, but they can be defined in a language-specific way as explained in Sect. 2 for arrays, and similarly for records. Algebraic data types (ADT) can be defined in the IR, and pattern matching is supported by **case** expressions. All these types (i.e. arrays, records and ADTs), and their primitive operators, are directly supported by the SMTs underlying the CAVI-ART platform. They contain a rich set of axioms allowing to reason about the formulas using them.

Other features which are present in a particular language but not in others, can be mapped to the IR by the front-end of each particular language, either by introducing new primitive types and operators, supported by their corresponding theories, or by representing them in the IR built-in types. An example of this is the mapping of the OO-language heap into an array variable that is passed around as an additional argument of methods, as it has been illustrated in our running example.

The definition of the axiomatic semantics of the IR, given as weakest preconditions, is as follows:

$$
\begin{array}{lcl}
wp(\textbf{let } x = e_1 \textbf{ in } e_2, R) & \stackrel{\text{def}}{=} & Dom(e_1) \wedge (x = e_1) \rightarrow wp(e_2, R) \\
wp(\textbf{case } a \textbf{ of } \ldots C\ x_1 \cdots x_n \rightarrow e \ldots, R) & \stackrel{\text{def}}{=} & (a = C\ x_1 \cdots x_n) \rightarrow wp(e, R) \\
 & & \wedge \text{ the remaining alternatives} \\
wp(\textbf{case } a \textbf{ of } \mathit{true} \rightarrow e_1; \mathit{false} \rightarrow e_2, R) & \stackrel{\text{def}}{=} & (a \rightarrow wp(e_1, R)) \wedge (\neg a \rightarrow wp(e_2, R))
\end{array}
$$

For many primitive applications (e.g. $e \equiv x + y$), $Dom(e)$ is assumed to be *true*. But some others are partial functions and require a precondition. Function definitions are assumed to be annotated with their respective precondition and postcondition. Let $f\ x_1 \cdots x_n = e$ be the definition of a function $f$, and let respectively $Q(x_1, \ldots, x_n)$ and $R(x_1, \ldots, x_n, res)$ be its precondition and postcondition, where *res* stands for $f$'s result. Then, in an application such as $f(a_1, \ldots, a_n)$, it must be proved that $Q(a_1, \ldots, a_n)$ holds before reaching this call, and it can be assumed that $R(a_1, \ldots, a_n, res)$ holds when $f$ returns.

In principle, we do not need to define an operational semantics for the IR, since its aim is not to be executed, but rather to be used for verification. When we define $wp(e, R) \stackrel{\text{def}}{=} Q$ for an expression $e$ with free variables $\overline{x}$, and predicates $Q(\overline{x})$ and $R(\overline{x}, v)$, we mean as usual that the set of all the initial states for the variables $\overline{x}$ guaranteeing that the value $v$ to which $e$ is evaluated satisfies $R(\overline{x}, v)$ is exactly that specified by $Q(\overline{x})$. This logical definition is supposed to capture the semantics of $e$ independently of the details of its evaluation. In this way, it is not important whether the evaluation mechanism is imperative or functional, whether there is, or is not, internal sharing during $e$'s evaluation, or even whether the evaluation order is lazy or eager.

## 5 Determining the Invariant Locations

In order to set the invariant conditions in the appropriate places, the CFG must be structured according to its strongly connected components (SCC) and subcomponents. Formally, the *Connected Components Structure* (CCS) of a graph

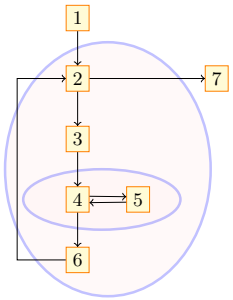


**Fig. 6.** CCS for the CFG of Fig. 4

$G$ is a list of components, where each one is either a single node, or a pair with an *entry point* and a list of components:

$$\begin{aligned} CCS &::= [COMP] \\ COMP &::= node \mid (entry\_node, CCS) \end{aligned}$$

This structure is built up by computing the maximal SCCs of a graph, then the SCCs inside these components, and so on. The resulting structure contains all the nodes of the graph grouped according to the loops of the original program. For any pair $(entry\_node, ccs)$ in the structure, the subgraph of $G$ corresponding to the nodes of $ccs$ is a connected one, and $entry\_node$ is the only entry point to this subgraph. Moreover, for any component $c$ of a CCS, except for the outermost one, there is a component $c'$ in the immediate prior level which contains some node connected to the entry node of $c$.

*Example 4.* Considering the CFG of Fig. 4, and disregarding for simplicity the node contents, the CCS is $[1, (2, [3, (4, [5]), 6]), 7]$, which is represented in Fig. 6. We get the external components: 1, 7, and the node set $[2, 3, 4, 5, 6]$, which has 2 as entry point, and then another internal component with the nodes $[4, 5]$, which has 4 as entry point. The invariants should be placed before the entry points 2 and 4, which correspond to the entry points of the loops of Ex. 1.

The function *cfg_to_ccs(G)* of Fig. 7 computes the CCS of a given control flow graph $G$. It decomposes the graph into connected subgraphs in successive recursive calls, until it reaches the base case of a single node (line 2). Otherwise, it searches for the entry point of the graph (line 5) which is guaranteed to exists since the graph is a CFG. Then, it considers the subgraph $G'$ (line 6) obtained by removing the entry point and its edges, and computes their strongly connected components (see [5]) of this subgraph (line 7). These components are then sorted by the function *sort* (line 8) as follows:

- Collapse each strongly connected component into a single node;
- Compute a topological sort of the resulting graph in a list (this sorting is always possible as cycles have been collapsed in the previous step);
- Uncollapse the strongly connected components;

Then for each list component (line 9), the algorithm obtains the corresponding subgraph (line 10) and the corresponding CCS for them (line 11). Finally, it returns the list of components, together with their entry point (line 13).

```
1  cfg_to_ccs(G):
2  if G is a single node then
3    return [G]
4  else
5    In ← entry_point(G)
6    G' ← G − {In}
7    Comps ← strongly_connected_components(G')
8    [C_1, ..., C_n] ← sort(Comps)
9    for all C_i in [C_1, ..., C_n] do
10     G_i ← subraph of G with the nodes of C_i
11     CCS_i ← cfg_to_ccs(G_i)
12   end for
13   return (In,[CCS_1, ..., CCS_n])
14 end if
```

**Fig. 7.** Algorithm computing the CCS of a graph

## 6 Generating the Verification Conditions

The verification of a complex program is usually done in a modular way, procedure by procedure. Indeed, this is the whole purpose of defining pre-post assertions for every user procedure: to make it possible the verification of each one independently of the others. We concentrate then in the activities associated to generating the VCs for a single user procedure. By this we mean a user unit, together with its pre-post assertions, disregarding whether it comes from an imperative or a functional input language.

After the transformation of Sect. 5, invariant assertions are placed as preconditions of some IR nodes. The CAVI-ART platform will help the user in this task, either by synthesizing parts of the invariants, or by completing the incomplete ones given by the user. The description of this part of the project is beyond the purpose of this paper. In what follows, we assume that the invariants have been placed by someone in the locations computed by the algorithm of Sect. 5.

Summarizing the result of the transformations described in sections 2 and 3, given a procedure we get an IR consisting of:

1. A function definition for every basic block (BB).
2. Each BB consists of a sequence of **let** expressions, ended in a jump. Each **let** binding is either an atom, or an application. A jump is simple, or it is a **case** with a simple jump at each of its branches. A simple jump to the *exit* node consists of a tuple expression returning the relevant variables. Otherwise, it is a call to another BB, passing the relevant variables as arguments.
3. The postcondition assertion, annotated in every arc to the *exit* node.
4. The precondition assertion, annotated in the only arc leaving the *entry* node.
5. An invariant assertion as precondition of the entry node of every CCS.

The IR may have a hierarchical structure reflecting the decomposition of an imperative CFG into its constituent CCSs. In this section, we look at it as a flat set of BBs recursively calling to each other, or as a control flow graph consisting of a set of nodes and a set of directed arcs between them.

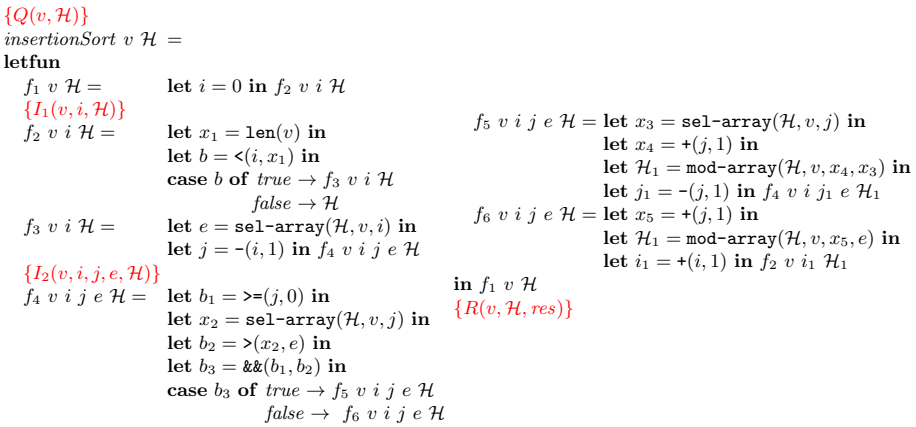


**Fig. 8.** IR of the insertion sort algorithm

*Example 5.* In Fig. 8 we show the flattened version of the IR corresponding to the CFG/SSA of the example of Fig. 4. In that IR, the locations of assertions $I_1$ and $I_2$ —i.e. the preconditions of $f_2$ and $f_4$— are indicated, and they correspond to the invariants. The precondition $Q$ and the postconditon $R$ are also indicated. For this example, a typical postcondition $R$ will assert that the output vector is sorted and that it is a permutation of the original one. A typical invariant $I_1$ of the **for** loop will assert also the second property, and that sortedness holds up to the element in position $i-1$. Invariant $I_2$ is a bit more involved.

The VC generation has two phases: (1) Assertion propagation, and (2) VC extraction.

*Assertion propagation* Let us start with a simple case, a BB having a simple jump at its end, and the rest of the BB consisting of a **let** sequence in which each bound expression is a primitive operator application, i.e. it has the form:

$$\textbf{let } x_1 = e_1 \textbf{ in } \ldots \textbf{ let } x_n = e_n \textbf{ in } \{Q\}\ f\ \overline{a}$$

where each $e_i$ represents a primitive application. Moreover, we know the assertion $Q$ that must hold in the output arc, i.e. the precondition $Q(\overline{y})$ of function $f$. Then, the precondition $Q_1$ propagated to the beginning of this BB, assuming that $Dom(e_i) = true$ for all $i$, is simply:

$$Q_1 \equiv (x_1 = e_1) \rightarrow \cdots \rightarrow (x_n = e_n) \rightarrow Q(\overline{a})$$

Let us assume now that the $i$-th bound expression of the BB is a call $g\ \overline{a'}$ to an external function $g$ for which we know its precondition $Q_g(\overline{y})$, and its postcondition $R_g(\overline{y}, res)$. Then, the propagation is split into two parts:

$$Q_1 \equiv (x_1 = e_1) \rightarrow \cdots \rightarrow (x_{i-1} = e_{i-1}) \rightarrow Q_g(\overline{a'})$$
$$R_1 \equiv (x_i = res) \rightarrow (x_{i+1} = e_{i+1}) \rightarrow \cdots \rightarrow (x_n = e_n) \rightarrow Q(\overline{a})$$

The following VC is also generated: $(x_1 = e_1) \to \cdots \to (x_{i-1} = e_{i-1}) \to R_g(\overline{a'}, res) \to R_1$. We proceed in a similar when more than one external call is present in the BB.

If the BB ends in a jump such as **case** $a$ **of** $true \to \{Q_1\} \ldots$ $false \to \{Q_2\}$ ..., where we know the assertions $Q_1$ and $Q_2$ holding at each output jump, the assertion propagated just before the **case** is: $(a \to Q_1) \wedge (\neg a \to Q_2)$. The rest of the backwards propagation through the BB is as above.

Finally, if the BB ends in a jump such as **case** $a$ **of** $\ldots C_i\ x_1 \cdots x_n \to \{Q_i\} \ldots$, where we know the assertion $Q_i$ holding at each output jump, the asssertion propagated just before the **case** is:

$(a = C_i\ x_1 \cdots x_n \to Q_i) \wedge \ldots$ a similar conjunction for each remaining branch

If a default clause is present, the branch assertion $Q$ at this jump is known, and there are $k$ prior branches, the conjunction for this clause is a bit more complex:

$$(a \neq C_1\ x_{11} \cdots x_{1n_1}) \wedge \cdots \wedge (a \neq C_k\ x_{k1} \cdots x_{1n_k}) \to Q$$

*VC extraction* After the propagation phase, we get an assertion propagated just before every BB body, and also some VCs coming from the calls to external procedures. The remaining VCs belong to one of the two following cases:

1. The user procedure precondition $Q_P$ must be stronger than or equal to the assertion $Q$ propagated to the single arc leaving the *entry* node, i.e. the verification condition $Q_P \to Q$ is generated.
2. If the BB precondition is an invariant $I$, then this invariant must be stronger than or equal to the assertion $Q$ propagated just before the BB body, i.e. the verification condition $I \to Q$ is generated for each invariant $I$.

*Example 6.* For the example of Fig. 8, the following VCs are generated:

1. $Q(v, \mathcal{H}) \to (i = 0) \to I_1(v, i, \mathcal{H})$
2. $I_1(v, i, \mathcal{H}) \to (x_1 = \texttt{len}(v)) \to (b = i < x_1) \to \neg b \to R(v, \mathcal{H}, \mathcal{H})$
3. $I_1(v, i, \mathcal{H}) \to (x_1 = \texttt{len}(v)) \to (b = i < x_1) \to b \to$
   $(\mathcal{H}(v) = Array\ a \wedge 0 \leq i < a.length)$
4. $I_1(v, i, \mathcal{H}) \to (x_1 = \texttt{len}(v)) \to (b = i < x_1) \to b \to (e = \texttt{get}\ a\ i) \to$
   $(j = i - 1) \to I_2(v, i, j, e, \mathcal{H})$
5. $I_2(v, i, j, e, \mathcal{H}) \to (b_1 = j \geq 0) \to (\mathcal{H}(v) = Array\ a \wedge 0 \leq j + 1 < a.length)$
6. $I_2(v, i, j, e, \mathcal{H}) \to (b_1 = j \geq 0) \to (x_2 = \texttt{get}\ a\ j) \to (b_2 = x_2 > e) \to$
   $(b_3 = b_1\ \&\&\ b_2) \to b_3 \to (x_3 = \texttt{get}\ a\ j) \to (x_4 = j + 1) \to$
   $(a_1 = \texttt{set}\ a\ x_4\ x_3) \to (\mathcal{H}_1 = \texttt{set}\ \mathcal{H}\ v\ a_1)) \to (j_1 = j - 1) \to I_2(v, i, j_1, e, \mathcal{H}_1)$
7. $I_2(v, i, j, e, \mathcal{H}) \to (b_1 = j \geq 0) \to (\mathcal{H}(v) = Array\ a \wedge 0 \leq j + 1 < a.length)$
8. $I_2(v, i, j, e, \mathcal{H}) \to (b_1 = j \geq 0) \to (x_2 = \texttt{get}\ a\ j) \to (b_2 = x_2 > e) \to$
   $(b_3 = b_1\ \&\&\ b_2) \to \neg b_3 \to (x_5 = j + 1) \to (a_1 = \texttt{set}\ a\ x_5\ e) \to$
   $(\mathcal{H}_1 = \texttt{set}\ \mathcal{H}\ v\ a_1)) \to (i_1 = i + 1) \to I_1(v, i_1, \mathcal{H}_1)$

When $Q$, $I_1$, $I_2$ and $R$ are replaced by actual predicates, the resulting VCs could be automatically discharged by a platform such as Why3. Its gallery of verified programs (see `http://why3.lri.fr/`), includes an insertion sort algorithm with VCs very similar to ours which are easily discharged.

## 7 Conclusions and related work

Many other intermediate representations of programs have been defined with different purposes. Restricting us to IRs for verification platforms, it has become popular the so-called IVLs (*Intermediate Verification Languages*). An example of these is Boogie2, used in Dafny [15]. Its semantics is given in terms of sets of traces, and it is very much tied to imperative languages. Its type system is more powerful than Hindley-Milner polymorphism, and this feature has shown to be very convenient for modeling the OO-languages heap. But it is not clear how functional languages could be mapped to it. The Why3 platform [10] offers WhyML as IR. In fact, this is a high level language, a kind of Standard ML with state and loops, but it has been used as IR for verifying C and Java programs. Due to its lack of support, some features of these languages, notably the heap, has been modeled in an awkward way.

A third related IR is LLVM[2]. Its purpose is to serve as IR for both imperative and functional languages in order to promote portability of these languages to different machines, interoperability between different paradigms, and to take profit of common static analyses targeted towards runtime performance. At first, we considered LLVM as IR for out platform, but we did not like the way in which functional languages can be translated into it. For instance, the case distinction provided by our `case` is closer to the pattern matching translation of Haskell (based on data types) and can be subsequently translated into a Why3 theory in a straightforward way. In the LLVM, however, we would need to perform a case distinction (`switch`) on the tag of the constructor and then assign the variables bound by each pattern in each branch. In addition, the LLVM provides a considerable amount of low level operations, whereas our purpose was quite the opposite: provide a reduced set of language dependent primitive functions whose behaviour will be specified in a language dependent theory. This allows us to express their properties in a way that is closer to the source language. The same applies to array indexing, which is built in the LLVM IR via the `getelementptr`/`extractvalue` instructions, whereas in our approach is another language dependent primitive whose behaviour may vary between different languages, especially when indexing beyond the array bounds.

A last related formalism is that of Constrained Horn Clauses (CHC) [13, 12]. They have been successfully used to express properties that a program must satisfy, such as termination or functional correctness. There are sophisticated algorithms which may decide, whenever this is possible, whether a CHC set is satisfiable or not, and hence whether the desired property holds. In this sense, CHC can be seen as a machinery complementary to that of SMT solvers, in order to automatically verify properties. It is more questionable whether CHC can play the role of our IR. In [6] it is shown how to encode the semantics of a subset of C into CHC programs, and how to transform a C program into a semantically equivalent CHC one by specializing the semantics with respect to the C source. For our purposes, however, this kind of representation is too low level and introduces many details which obscure the generation of verification conditions based on assertions and weakest preconditions.

[2] LLVM stands for Low Level Virtual Machine. See `http://llvm.org/`

Our IR supports most features of imperative and functional languages, including all varieties of control statements, exceptions, recursion, object orientation, heap modeling, arrays, algebraic data types, pattern matching, polymorphism, and higher-order. Moreover, the VCs we generate are very much adapted to what current SMTs expect. For the moment, we do not support concurrency and reflection in the sense of languages such as Java.

## References


1. W. Ahrendt et al. The KeY platform for verification and analysis of Java programs. In *VSTTE 2014*, volume 8471 of *LNCS*, pages 1–17. Springer-Verlag, 2014.
2. F. E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.
3. A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd edition, 2003.
4. R. Carlsson. An introduction to core erlang. In *Proceedings of the PLI01 Erlang Workshop*, 2001.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
6. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions by program specialization. In *PPDP 2015*, pages 91–102. ACM, 2015.
7. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, pages 337–340. Springer, 2008.
8. M. Deters, A. Reynolds, T. King, C. W. Barrett, and C. Tinelli. A tour of CVC4: how it works, and how to use it. In *FMCAD 2014*, page 7. IEEE, 2014.
9. J.-C. Filliâtre. One logic to use them all. In M. P. Bonacina, editor, *24th International Conference on Automated Deduction, CADE-24*, volume 7898, pages 1–20. Springer, 2013.
10. J.-C. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In *ESOP 2013*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.
11. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI 1993*, pages 237–247. ACM, 1993.
12. J. P. Gallagher and B. Kafle. Analysis and transformation tools for constrained horn clause verification. *CoRR*, abs/1405.3883, 2014.
13. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI 2012*, pages 405–416. ACM, 2012.
14. G. Klein and T. Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.
15. K. R. M. Leino. Developing verified programs with dafny. In B. Brosgol, J. Boleng, and S. T. Taft, editors, *HILT*, pages 9–10. ACM, 2012.
16. S. L. Peyton Jones and D. R. Lester. *Implementing functional languages*. Prentice Hall international series in computer science. Prentice Hall, Impr., New York, 1992.
17. D. Rémy. Using, Understanding, and Unraveling the OCaml Language From Practice to Theory and vice versa. In *Applied Semantics*, volume 2395 of *LNCS*, pages 413–536. Springer, 2002.
18. G. Team. Glasgow Haskell Compiler core Language. https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/CoreSynType, Online; accessed 30-April-2015.