

# Program Verification 2012–2013

## Project 3: Bytecode Verification Engine

Danny Bergsma      Jaap van der Plas

April 3, 2013

### 1 Introduction

For PV Project 3, we have designed and implemented a weakest-precondition calculus for a bytecode language. [3] We use this calculus to verify specifications (pre- and post-condition combinations) for programs in this language: the specification is correct if and only if the precondition implies the weakest precondition (WP) with respect to the given postcondition and program (we call this implication the verification condition - VC); the WP is calculated by subsequently applying the rules from the defined WP calculus (see Section 2.3). We check the validity of the VC by converting it to Data.SBV [1] format and subsequently calling SBV's prove function; SBV interfaces with Microsoft's theorem prover (SMT solver) Z3 [2]. Z3/SBV returns whether the VC is valid or not; if the VC is invalid, Z3/SBV gives a counterexample.

We have implemented the verifier in Haskell, using attribute grammars [4]. This report will explain our solution with emphasis on the conceptual ideas. For implementation details, we refer to the Haddock (HTML) documentation and the (documented) code. The included README elaborates on practical matters, like building and running the verifier, how to test the included examples, et cetera. The README also lists all implemented features.

Section 2 explains our solution for the base task. Our 'return from anywhere' extension is described in Section 3; our bounded verification extension, to deal with loops, is described in Section 4.

### 2 Base task

This section presents our solution for the base task: designing and implementing a verification engine for a bytecode language. Section 2.1 describes how we represent programs in this language; Section 2.2 describes how we represent specifications (pre- and postconditions) and convert them to the SBV format.

For a more detailed description of the base task, we refer to the assignment [3].

## 2.1 Programs

The assignment [3] already contained a description of the syntax and semantics of the bytecode language. The representation in Haskell is straightforward; the file `PV3/-Program/ProgramAST.ag` contains the defined AST.

We decided that the type of parameters, local variables, literals and the return value is either *bool* or *int*. To compare two *bools*, we added the new instruction `EQUIV`; we cannot use the existing `EQ` instruction, as our substitution implementation (Section 2.3) needs to know the type of the terms  $stack_T$  and  $stack_{T-1}$  that are substituted for the term  $stack_T$  in the WP rule for `EQ/EQUIV` (see Section 2.3). If we compare two *bools*, both terms would be of type *bool*; if we compare two *ints*, both terms would be of type *int*. The required type information is not available from a type checker, as we do not have one, so we decided to incorporate the type information explicitly into the syntax.

We have not developed a parser for programs.

## 2.2 Conditions

The assignment [3] already outlined what conditions should be accepted by the verifier: simple expressions and first-order universal and existential quantifications. We represent conditions (boolean expressions) again as ASTs, defined in the file `PV3/-Condition/ConditionAST.ag`. More complex boolean expressions can be built out of simpler ones, using the most common logical connectives ( $\vee$ ,  $\wedge$ ,  $\dots$ ). Relational operators like  $>$ ,  $\geq$ ,  $\dots$  are used to build boolean expressions out of integer expressions; more complex integer expressions can be built out of simpler ones, using addition, subtraction, multiplication and (integer) division. Conditions can also consist of integer and boolean literals. Return values (*return*) and pre-invocation parameter values ( $a_i$ ) are explicitly encoded by a separate data constructor.

(Intermediate) Weakest preconditions are represented by the same AST type. As (intermediate) WPs may reference to internal (stack) data, like  $stack_T$ , we defined the attribute grammar `PV3/Condition/External.ag` that checks whether a given condition is “external”, i.e. it does *not* refer to internal (stack) data; this is useful for verification purposes, see Section 2.4.

### 2.2.1 Quantifications

Universal and existential quantification can be used in conditions, but we doubt the usefulness. We included the example `PV3/Examples/Quantification.hs`. Its program’s higher-level counterpart is:

```
int p (int a) {  
  return a;  
}
```

The specification for the example is:

$$pre : 6 \leq a \leq 10$$

$$post : \exists 6 \leq x \leq 10 : return = x$$

When we verify this example, we indeed prove the specification correct: Q.E.D. When we change the postcondition to

$$post : \exists 7 \leq x \leq 10 : return = x$$

we get an counterexample, indicating that the specification is incorrect:

```
Falsifiable. Counter-example:
  a0 = 6 :: SInteger
```

### 2.2.2 Conversion to SBV

Converting from conditions to SBV format is done by the attribute grammar PV3/-Condition/ConvertToSBV.ag. This AG builds a Symbolic computation. Essentially, every operator is replaced with its symbolic counterpart. Variables, including (pre-invocation) parameter values, are looked up in an environment (Map), to ensure that each occurrence of a variable reference refer to the same actual variable. The environment is pre-filled with all parameters ( $a_0, \dots$ ). An (universal/existential) quantification inserts a new variable into this environment and passes the updated environment down to the (boolean) subexpression.

Our decision to accept boolean and *int* parameters complicates this pre-filling of the environment, as type information is not available. Our first strategy was to insert each parameter twice, once into the boolean environment and once into the *int* environment (we need, at least conceptually, two environments, as our condition is typed). Unfortunately, when the verifier gave an counterexample, it listed all parameters twice, as there are two “versions” (a boolean and *int* one). It also listed parameters which did not occur at all in the VC, as the pre-filling was done based on the number of variables *specified by the program*; as these “non-occurring” parameters are irrelevant for a(n eventual) counterexample, we would like to omit them.

Our solution to these issues is to extract type information from the verification condition. This is done by the attribute grammar PV3/Condition/Extract.ag. This AG returns a tuple: the first element a set of indices of all parameters that are used as boolean, the second element a set of indices of all parameters that are used as *ints*. If the intersection of these two sets is non-empty, one or more parameters are used as boolean *and* integer, and an error is issued. In the other case, we now know exactly how to fill the boolean and *int* environments correctly.

We do not insert the return value into either of the environments, as a valid VC does not refer to the return value; if it does, the precondition refers to the return value, which makes no sense, or the postcondition refers to the value, but the program does not always (possibly never) return a value at all. This is at least inconsistent; programs in our bytecode language that do not return a value are also pretty useless, as there is no global state the program could alter. If the VC does refer to the return value, an error is given, as it cannot be found in either of the environments; the same error is issued when any other non-existing variable is being referred.

We have not developed a parser for conditions.

### 2.3 Weakest precondition rules

Essentially, our weakest precondition rules rewind the effects of instructions. Some of the rules were already given in the assignment [3].

$$\begin{aligned} \text{SETLOCAL } k \ x &=_{\text{sem}} \{loc_k := x\} \\ wp(\text{SETLOCAL } k \ x) \ Q &=_{\text{def}} \ Q[x/loc_k] \end{aligned}$$

$$\begin{aligned} \text{LOADLOCAL } k &=_{\text{sem}} \{T := T + 1 ; stack_T := loc_k\} \\ wp(\text{LOADLOCAL } k) \ Q &=_{\text{def}} \ (Q[loc_k/stack_T])[ (T + 1)/T ] \end{aligned}$$

$$\begin{aligned} \text{STORELOCAL } k &=_{\text{sem}} \{loc_k := stack_T ; T := T - 1\} \\ wp(\text{STORELOCAL } k) \ Q &=_{\text{def}} \ (Q[(T - 1)/T])[stack_T/loc_k] \wedge T \geq 0 \end{aligned}$$

$$\begin{aligned} \text{LOADPARAM } k &=_{\text{sem}} \{T := T + 1 ; stack_T := param_k\} \\ wp(\text{LOADPARAM } k) \ Q &=_{\text{def}} \ (Q[param_k/stack_T])[ (T + 1)/T ] \end{aligned}$$

$$\begin{aligned} \text{STOREPARAM } k &=_{\text{sem}} \{param_k := stack_T ; T := T - 1\} \\ wp(\text{STOREPARAM } k) \ Q &=_{\text{def}} \ (Q[(T - 1)/T])[stack_T/param_k] \wedge T \geq 0 \end{aligned}$$

$$\begin{aligned} \text{PUSHLITERAL } l &=_{\text{sem}} \{T := T + 1 ; stack_T := l\} \\ wp(\text{PUSHLITERAL } l) \ Q &=_{\text{def}} \ (Q[l/stack_T])[ (T + 1)/T ] \end{aligned}$$

$$\begin{aligned} \text{POP} &=_{\text{sem}} \{T := T - 1\} \\ wp(\text{POP}) \ Q &=_{\text{def}} \ Q[T - 1/T] \wedge T \geq 0 \end{aligned}$$

$$\begin{aligned} \text{ADD} &=_{\text{sem}} \{stack_{T-1} := stack_{T-1} + stack_T ; T := T - 1\} \\ wp(\text{ADD}) \ Q &=_{\text{def}} \ (Q[(T - 1)/T])[ (stack_{T-1} + stack_T)/stack_{T-1} ] \wedge T \geq 1 \end{aligned}$$

$$\begin{aligned} \text{MIN} &=_{\text{sem}} \{stack_{T-1} := stack_{T-1} - stack_T ; T := T - 1\} \\ wp(\text{MIN}) \ Q &=_{\text{def}} \ (Q[(T - 1)/T])[ (stack_{T-1} - stack_T)/stack_{T-1} ] \wedge T \geq 1 \end{aligned}$$

$$\begin{aligned} \text{MUL} &=_{\text{sem}} \{stack_{T-1} := stack_{T-1} * stack_T ; T := T - 1\} \\ wp(\text{MUL}) \ Q &=_{\text{def}} \ (Q[(T - 1)/T])[ (stack_{T-1} * stack_T)/stack_{T-1} ] \wedge T \geq 1 \end{aligned}$$

$$\begin{aligned} \text{EQ} &=_{\text{sem}} \{ \text{stack}_{T-1} := \text{stack}_{T-1} \equiv \text{stack}_T ; T := T - 1 \} \\ \text{wp}(\text{EQ}) Q &=_{\text{def}} (Q[(T-1)/T])[(\text{stack}_{T-1} \equiv \text{stack}_T) / \text{stack}_{T-1}] \wedge T \geq 1 \end{aligned}$$

$$\begin{aligned} \text{LT} &=_{\text{sem}} \{ \text{stack}_{T-1} := \text{stack}_{T-1} < \text{stack}_T ; T := T - 1 \} \\ \text{wp}(\text{LT}) Q &=_{\text{def}} (Q[(T-1)/T])[(\text{stack}_{T-1} < \text{stack}_T) / \text{stack}_{T-1}] \wedge T \geq 1 \end{aligned}$$

$$\begin{aligned} \text{LTE} &=_{\text{sem}} \{ \text{stack}_{T-1} := \text{stack}_{T-1} \leq \text{stack}_T ; T := T - 1 \} \\ \text{wp}(\text{LTE}) Q &=_{\text{def}} (Q[(T-1)/T])[(\text{stack}_{T-1} \leq \text{stack}_T) / \text{stack}_{T-1}] \wedge T \geq 1 \end{aligned}$$

$$\begin{aligned} \text{GT} &=_{\text{sem}} \{ \text{stack}_{T-1} := \text{stack}_{T-1} > \text{stack}_T ; T := T - 1 \} \\ \text{wp}(\text{GT}) Q &=_{\text{def}} (Q[(T-1)/T])[(\text{stack}_{T-1} > \text{stack}_T) / \text{stack}_{T-1}] \wedge T \geq 1 \end{aligned}$$

$$\begin{aligned} \text{GTE} &=_{\text{sem}} \{ \text{stack}_{T-1} := \text{stack}_{T-1} \geq \text{stack}_T ; T := T - 1 \} \\ \text{wp}(\text{GTE}) Q &=_{\text{def}} (Q[(T-1)/T])[(\text{stack}_{T-1} \geq \text{stack}_T) / \text{stack}_{T-1}] \wedge T \geq 1 \end{aligned}$$

$$\begin{aligned} \text{EQUIV} &=_{\text{sem}} \{ \text{stack}_{T-1} := \text{stack}_{T-1} \leftrightarrow \text{stack}_T ; T := T - 1 \} \\ \text{wp}(\text{EQUIV}) Q &=_{\text{def}} (Q[(T-1)/T])[(\text{stack}_{T-1} \leftrightarrow \text{stack}_T) / \text{stack}_{T-1}] \wedge T \geq 1 \end{aligned}$$

$$\begin{aligned} \text{return} &=_{\text{sem}} \{ \text{return} := \text{stack}_T ; T := T - 1 \} \\ \text{wp}(\text{return}) Q &=_{\text{def}} (Q[T-1/T])[\text{stack}_T / \text{return}] \wedge T \geq 0 \end{aligned}$$

$$\text{wp}(s_1; s_2; \dots; s_n) Q =_{\text{def}} \text{wp } s_1 (\text{wp } s_2 (\dots (\text{wp } s_n Q)))$$

$$\text{wp}(\text{iftrue } s_1 \text{ else } s_2) Q =_{\text{def}} ((\text{wp } s_1 Q)[T-1/T] \wedge \text{stack}_T) \vee ((\text{wp } s_2 Q)[T-1/T] \wedge \neg \text{stack}_T) \wedge T \geq 0$$

$$\text{wp}(\text{prog } P \text{ } n \text{ } s) Q =_{\text{def}} ((\text{wp } s (Q \wedge T \equiv -1))[a_0 / \text{param}_0][a_1 / \text{param}_1] \dots [a_{n-1} / \text{param}_{n-1}] [-1/T])$$

Note that we enforce that the stack is empty after returning (which includes a POP) by incorporating the conjunct  $T \equiv -1$  into the WP rule for programs; we enforce that the stack is empty before entering the program by substituting -1 for T in the same rule. The weakest precondition of the body of the program may reference to (current) parameter values ( $\text{param}_i$ ). As these values are now equal to the pre-invocation ones, we substitute  $a_i$  for each  $\text{param}_i$ ; they have now the same name as pre-invocation parameter values, if present, in the given pre- and postcondition.

The WP rules are straightforwardly implemented in the attribute grammar PV3/-WP.ag. The actual substitutions are done by the attribute grammar PV3/Condition/-Replace.ag; this AG is called from within the AG WP.ag.

## 2.4 Verification

The actual verification consists of these steps:

1. `WP.ag`: calculating the weakest precondition with respect to the given postcondition and program
2. `PV3/Verification.hs`: generating the VC: precondition  $\rightarrow$  wp from step 1
3. `External.ag`: checking the VC for being “external” (see below)
4. `Extract.ag`: extracting the parameter type information from the VC
5. `Verification.hs`: pre-filling the environment(s)
6. `ConvertToSBV.ag`: converting the VC to a Symbolic SBV computation, using the pre-filled environment(s)
7. `Main.hs`: calling SBV’s prove function to prove the Symbolic VC from step 6 correct
8. `SBV`: calling Z3 and returning the result of the verification (correct or not); if not correct, giving counterexample
9. `Main.hs`: outputting result of the verification and counterexample, if present

The function

```

verify :: Cond          -- Precondition of specification.
        → Program       -- Program of specification.
        → Cond          -- Postcondition of specification.
        → Symbolic SBool -- Resulting verification condition.

```

from `Verification.hs` drives the first part (the first six steps) of the verification; *verify* is called by the *main* function in `Main.hs`.

Consider the malformed program:

```

P (0) {
  return ;
}

```

with as postcondition:

*post* : *return* = 2

The weakest precondition with respect to this program and postcondition refers to *stack<sub>T</sub>*, as the *return* term in the postcondition is replaced with *stack<sub>T</sub>*. This indicates that the program is malformed and no specification for the program can be proven correct. Every weakest precondition (and, consequently, VC), with respect to a given program, that refers to internal (stack) data indicates that the given program is malformed and that no specification for this program can be proven correct. So, if the VC is

not “external”, we do not need to convert it to SBV and give it to Z3; instead, we issue the error that the given program is malformed.

Note that a counterexample is only given when one or more (pre-invocation) values ( $a_i$ ) and/or universally/existentially quantified variables appear in the (invalid) VC; if this is not the case, the invalidity of the VC does not depend on these values/variables and `Falsifiable` is the only output given.

### 3 ‘Return from anywhere’ extension

### 4 Bounded verification extension

## References

- [1] L. Erk  k, 2013. *sbv-2.10: SMT Based Verification: Symbolic Haskell theorem prover using SMT solving*. Retrieved April 3, 2013 from Hackage: <http://hackage.haskell.org/package/sbv>.
- [2] Microsoft Research, 2013. *Z3 - Home*. Retrieved April 3, 2013 from CodePlex: <http://z3.codeplex.com/>.
- [3] S.W.B. Prasetya, 2013. *Project 3: Bytecode Verification Engine*. Retrieved April 3, 2013 from the ‘Program Verification’ course page: [http://www.cs.uu.nl/docs/vakken/pv/1213/stuffs/PV\\_P3\\_1213.pdf](http://www.cs.uu.nl/docs/vakken/pv/1213/stuffs/PV_P3_1213.pdf).
- [4] Utrecht University, 2009. *Attribute Grammar System*. Retrieved April 3, 2013 from cs.uu.nl’s wiki: <http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem>.