A Weakest Pre-condition Calculus in Haskell

João Paulo Pizani Flor
Department of Information and Computing Sciences,
Utrecht University - The Netherlands
e-mail: j.p.pizaniflor@students.uu.nl

Thursday 4th April, 2013

1 Introduction

This report describes the solution to one of the programming assignments (project nr. 3) of the master course "Program Verification" at Utrecht University, taught in the 3rd period of the academic year 2012/2013. The assignment involved the implementation of a weakest pre-condition calculus for a simple imperative-like programming language. The weakest pre-condition thus calculated was to be used to verify a Hoare triple in a theorem prover.

By calculating the weakest pre-condition for a program (given a certain post-condition), we reduce the problem of verifying the validity of a Hoare triple to the problem of verifying whether the user-supplied pre-condition implies the calculated weakest pre-condition.

Furthermore, the implication $pre \Rightarrow wp$ is valid *if and only if* the formula $\neg(pre \Rightarrow wp)$ is unsatisfiable. Therefore, if the backend theorem prover answers "unsat", then the implication is valid (which means the Hoare triple is valid), otherwise not.

2 Some design choices

During the conception and implementation of the calculus, some design decisions had to be taken, which impacted the way in which the software works. These are some of the most important design decisions:

Theorem prover The theorem prover we decided to use is Z3, from Microsoft Research. It is in active development, has a simple input format (SMT-Lib) and good documentation.

Theorem prover bindings At first, we tried to use the *SBV* Haskell package as a binding to the Z3 theorem prover. It seemed attractive as it was actively maintained and offered a “high-level” interface to the theorem proving backends. However, the translation of our datatypes to the types used by SBV proved to be too troublesome, and we ended up writing the formulas to standard output in the SMT-Lib2 format, used by Z3 and several other provers.

Arrays as functions After consulting the SMT-Lib reference (input language used in Z3), we found out that there is a “native” way to represent arrays. Even so, we decided to represent arrays as functions. This made easier the “initialization” from the precondition: if we used a “native” SMT-Lib array, we would have to fill it with values from the precondition, but by using functions, we can simply interpret the equalities normally.

Booleans as Integers We decided to make all arrays in the target SMT-Lib model have Integer indices and Integer values, for simplicity purposes. Therefore, boolean values in the stack, parameters and local variables are encoded as Integers, with false being 0 and any positive value being considered true.

3 The Weakest Pre-condition Calculus

In this section, we will go in depth over the implementation of the weakest pre-condition calculus developed. We will do this tour of the rules in a top-down fashion, beginning with the root rules, which define how to calculate the weakest pre-condition of a program as a whole, and of each kind of statement, recursively.

```
wp :: L0Program -> Sp0Formula -> Sp0Formula
wp (L0Program _ _ _ body) q = wp_ body q

wp_ :: L0Statement -> Sp0Formula -> Sp0Formula
wp_ (Inst i) q         = wp_inst i q
wp_ (Stmts ss) q       = wp_stmts ss q
wp_ (IfThenElse t e) q = theni :&: elsei
    where
        topZero = E $ S (X 0) :=: A (I 0)
        theni   = topZero     :=>: wp_ (Stmts [Inst L0Pop, t]) q
        elsei   = Not topZero :=>: wp_ (Stmts [Inst L0Pop, e]) q

wp_stmts :: [L0Statement] -> Sp0Formula -> Sp0Formula
wp_stmts []     q = q
wp_stmts (x:xs) q = wp_ x (wp_stmts xs q)
```

By the definition of the first function (wp), we can see that the weakest pre-condition (hereafter denoted as WP) of a program is simply the WP of its body (a statement). Then we have the definition of the WP for each case of statement. The list-of-statements case is calculated by a simple recursive fold over the list, as seen in the lecture notes. In the if-then-else case, the condition is whether the value on top of the stack is equal to zero; also, we take care to calculate the WP of the "then" and "else" branches with an additional pop instruction (which doesn't appear in the source code but happens in runtime).

Now we can take a look at function `wp_inst`, which calculates the WP for a single instruction, given the post-condition.

```
wp_inst :: L0Instruction -> Sp0Formula -> Sp0Formula
wp_inst (SetLocal k v)  = transformFormula (rewrSetLocal k v)
wp_inst (LoadLocal k)   = transformFormula (rewrLoadLocal k)
wp_inst (StoreLocal k)  = transformFormula (rewrStoreLocal k)
wp_inst (LoadParam k)   = transformFormula (rewrLoadParam k)
wp_inst (StoreParam k)  = transformFormula (rewrStoreParam k)
wp_inst (PushLiteral v) = transformFormula (rewrPushLit v)
wp_inst L0Pop           = transformFormula rewrPop_
wp_inst L0Return        = transformFormula rewrReturn
wp_inst binOp@_         = transformFormula (rewrOp binOp)
```

This function simply applies `transformFormula` to the given post-condition formula, passing as a parameter a *rewrite rule*. A rewrite rule is simply a function of type `Sp0Formula` $\rightarrow$ `Sp0Formula` (with `Sp0Formula` being the logic formula datatype. The `transformFormula` function is a recursive traversal scheme over the Sp0Formula datatype, which will apply the supplied rewrite rule only in the leaves of the tree.

We have exactly one rewrite rule per instruction in the language, along with some helper functions used to define them. After some thought, we could express the rewrite rules in a reasonably concise way. Let's first take a look at the helper functions:

```
rewrStackSize :: (SSize -> SSize) -> Sp0Formula -> Sp0Formula
rewrStackSize g = rewrForm_ isT (update g)
    where
```

```
        update g (T (I n)) = T (I (g n))
        update _ _      = error "rewrStackSize: should never match this pattern"

rewrStackIdx :: (Idx -> Idx) -> (Idx -> Bool) -> Sp0Formula -> Sp0Formula
rewrStackIdx g guard = rewrExpr isS guard (update g)
    where
        update g (S (X k)) = S (X (g k))
        update _ _      = error "rewrStackIdx: should never match this pattern"

rewrPush :: Sp0Formula -> Sp0Formula
rewrPush = rewrStackSize (subtract 1) . rewrStackIdx (subtract 1) (> 0)

rewrPop :: (Idx -> Bool) -> Sp0Formula -> Sp0Formula
rewrPop guard = rewrStackSize (+ 1) . rewrStackIdx (+ 1) guard
```

As you can see, the rewrite rules for pushing and popping to/from the stack adjust both the size of the stack and the indices of references to the stack. When pushing, we want to ONLY decrement the references in which indices are greater than zero, and also when popping we provide an extra parameter (used in later definitions) which “guards” the update.

Now we have the actual rewrite rules that correspond one-to-one to the “assembly” instructions in the language:

```
rewrSetLocal :: Idx -> L0Value -> Sp0Formula -> Sp0Formula
rewrSetLocal k v = rewrExprIdx isL k (const $ lang0ToSpec0Value v)
```

Here, we rewrite expressions by replacing all references to local variable k with the provided literal.

```
rewrLoadLocal :: Idx -> Sp0Formula -> Sp0Formula
rewrLoadLocal k = rewrPush . rewrExprIdx isS 0 (const $ L (X k))

rewrLoadParam :: Idx -> Sp0Formula -> Sp0Formula
rewrLoadParam k = rewrPush . rewrExprIdx isS 0 (const $ P (X k))

rewrPushLit :: L0Value -> Sp0Formula -> Sp0Formula
rewrPushLit v = rewrPush . rewrExprIdx isS 0 (const $ lang0ToSpec0Value v)
```

In the instructions which also “push” something onto the stack, we first replace references to the top of the stack with the appropriate substitute, and only then we perform the push rewrite. The order of the rewrites is important.

```
rewrPop_ :: Sp0Formula -> Sp0Formula
rewrPop_ = rewrPop (const True)

rewrStoreLocal :: Idx -> Sp0Formula -> Sp0Formula
rewrStoreLocal k = rewrExprIdx isL k (const $ S (X 0)) . rewrPop_

rewrStoreParam :: Idx -> Sp0Formula -> Sp0Formula
rewrStoreParam k = rewrExprIdx isP k (const $ S (X 0)) . rewrPop_

rewrReturn :: Sp0Formula -> Sp0Formula
rewrReturn = rewrExpr_ isR (const $ S (X 0))
```

In the other hand, for the instructions which perform a pop as side-effect, we first do the pop rewrite, then introduce a reference to the top of the stack. Again, the order of application of the rewriting rules is important.

```
rewrOp :: L0Instruction -> Sp0Formula -> Sp0Formula
rewrOp op = rewrExprIdx isS 0 (const $ op_ (S $ X 0) (S $ X 1)) . rewrPop (> 0)
    where op_ = case op of
        L0Add -> (:+:)
        L0Sub -> (:-:)
        ...
```

Finally, in the general rewriting rule for all binary stack operators, we replace references to the top of the stack with the operator applied to the TWO topmost elements of the stack. This replacement is only done after a *guarded* pop rewrite, which ensures that references to the top of the stack remain intact.

4 Running the examples

In the Haskell Cabal package provided in this distribution (called "lang0-wpverify"), an executable is defined which, when run, will produce SMT-Lib2 code in the standard output, suitable to be directly fed into the Z3 theorem prover for verification. The executable is itself called "lang0-wpverify", and can be executed together with Z3 like follows (assuming you compiled the package using the standard Cabal procedure):

```
./dist/build/lang0-wpverify/lang0-wpverify | z3 -smt2 -in
```

If the string "unsat" is printed, then it means that verification succeeded, i.e., the Hoare triple provided is valid. Otherwise, the execution of the program, given the pre-conditions, does NOT lead to the post-conditions according to the WP calculus implemented.

The source code for this executable is located under the path `src/UU/HoareLogic/Tests/Z3Tests.hs`, inside the `lang0-wpverify` package directory. New examples of Hoare triples can be added to this file by copy-pasting and modifying the one already present there.