Pascal-F Verifier Internal Design Document

CPCI #2 -- Preverification Checker

Special Issues in Jcode Generation

John Nagle

FACC / Palo Alto

Introduction

During the development of the Verifier, some special problems were discovered in the generation of Jcode from Pascal-F. Since the theory behind the code may not be obvious when examining the implementation, this document describes a number of problems found and how they were dealt with.

Calls

The semantics of the routine call are both complex and subtle. Complexity stems from the number of actions that can take place between the beginning of execution of a statement involving a call and the eventual return to the caller. Subtlety is introduced by the interactions between these actions.

A routine* call in Pascal-F may involve any or all of the following significant events:

- Passing of a variable or portion thereof by reference,
- Entry to a monitor,
- Exit from a monitor,
- Entry to one or more nested modules,
- Exit from one or more nested modules,
- Multiprogramming delay within the called routine,
- Access to or modification of global variables by the called routine, (‘‘side effects’’)
- Calls to other routines from within the called routine, including simple recursive calls, mutually recursive calls, and calls into or out of monitors or modules.
- Change of multiprogramming level by the called routine, and
- Startup of waiting processes at the request of the current process.

Our model of a procedure is constructed from the classical base of **ENTRY** and **EXIT** assertions. We verify the body of a routine to be correct relative to the entry and exit assertions; then, when verifying programs containing calls to routines, we use the entry and exit assertions to represent the semantics of the code within the routine. The entry assertions are viewed as constraints upon the caller; the exit assertions are the only information available about the result variables after return from the routine.

Determining which variables are affected by a given routine call turns out to be non-trivial. A routine can, of course, have side effects. A routine can also call other routines, which in turn may have side effects. Any routine in the call tree may wait for another process*, which process may change variables visible to the routines in the call tree at the moment of the wait. Further, changes to variables are directly relevant only to those routines which access them. In particular, where scope rules forbid access, changes caused by side effects are irrelevant.

A final complication is that we choose not to believe that the compiler enforces the scope rules of the language; since we are working from the output of the compiler we can check up on the compiler and see how variables are actually used.

We refer to the list of variables that are both affected by a call and relevant to the caller as the ‘‘relevant set-list’’ of the call. This relevant set-list may be different for calls to the same routine made from different scopes. This list is constructed in stages.

The first step in constructing the relevant set-list is to determine the ownership of every variable in the program. The owner of a variable is the dominator of its referents. That is, the block owning a variable is the least common block of all blocks containing references (set or use) to the variable. The owning block may be the block in which the variable is declared, or it may be a block within that block. For example, a variable declared in the main program but only referenced in one module is considered to be owned by that module, not by the main program. References in assertions are considered in determining the owning block.

The utility of the owning block concept lies in the fact that it is safe to assume that a variable will not

* The term ‘‘routine’’ will be understood to subsume ‘‘procedure’’ and ‘‘function’’.

* The present Pascal-F implementation forbids **wait** operations within procedures, but this is a temporary restriction of the compiler stack allocation algorithm.

change when control is outside the owning block of the variable. This is an implicit invariant of the verification. Explicit invariants, defined in **invariant** statements, may appear in any block. It is an error for an invariant statement to refer to a variable of an owning block different than that of the invariant. We must thus require that, when leaving a block, any invariants associated with that block be true. Conversely, when entering a block, we may assume the truth of the invariants associated with that block. Note that this approach requires that invariants be true when the block is entered for the first time. We enforce this by requring that the initialization section of the block be executed before calling any routines exported from the block.

There is some question as to the meaning of ‘‘leaving the block’’. We distinguish between temporary exits and calls which leave the block. A temporary exit is a call to a routine external to the block where the call cannot result in re-entry to the block by any means before the call in question returns. The invariant of a block need not hold during a temporary exit. A call which crosses a block boundary in an outward direction is considered a temporary exit if and only if all the following hold:

- The list of reachable routines for the routine being called does not include any routine exported from the block.
- The reachable routines do not perform any ‘‘wait’’ or ‘‘signal’’ operations.
- No call made by the reachable routines results in a decrease of priority level at call time.
- The call out of the block does not include a **var** argument where the actual argument is other than a procedure-local variable local to a block within the block being exited.

Calls for which control leaves the block but the conditions for temporary exit do not hold are considered calls to without the block. The invariants of the block must be proven true before such calls and may be assumed true after return.

Having determined the owning block for each variable, we must next determine the set-list of each routine. The set-list of a routine is the list of every variable which can be changed during a call to that routine. This includes all variables assigned by the routine, control variables of **for** statements, variables on the set-lists of routines called by the routine being examined, and variables passed as output **var** arguments. It also includes, for a procedure in a monitor, every variable owned by the monitor (?).

This list is determined by performing a transitive closure on the operation of calling, and then, for each routine, computing the union of the

We thus obtain the set-list for each routine. We require the relevant set-list. The criterion for relevance is negative; a variable is relevant unless ruled irrelevant by one of the following rules:

- A variable is irrelevant to a routine if the routine is not in the owning block of the variable, because in such a case the routine cannot mention the variable.
- A variable is irrelevant to a routine if the variable is local to the dynamic scope of the routine block, because the specification of the routine (the entry and exit conditions) cannot mention such a variable.
- A variable is irrelevant to a given call to a routine if the *caller* is not in the owning scope of the variable, because in such a case the caller cannot mention the variable, except that **VAR** arguments are always relevant to the caller.

The relevant set-list thus constructed, then, is the list of variables whose values are considered to have changed after return from the routine call. Only the information contained in the **exit** assertion of the routine is available after the call.

Module Invariants

Module invariants present a number of special problems. Jcode is generated for module invariants in four circumstances; when beginning the processing of a routine, when processing the end of a routine, when processing a call to a routine, and when processing the end of a a module body. These cases are handled as follows:

- At the beginning of a routine: examine all invariants of modules out to the dominant module. For each such invariant, if the invariant mentions any variable in the input or output list of the procedure, add the invariant to the assertions assumed true at entry to the procedure.
- At the end of a routine: examine all invariants of modules out to the dominant module. For each invariant, if the invariant mentions any variable in the output list of the routine, generate a REQUIRE for the invariant.
- At a call to a routine: examine all invariants of modules from the current module out to the dominant module of the callee. For each invariant, if the invariant mentions any variable in the output list of the routine, generate a REQUIRE for that invariant.
- At the end of a module body: generate a REQUIRE for every invariant of every module from the module being examined out to the dominant module of all INIT operations performed on the module.

By ‘‘dominant module’’ we mean the innermost module which contains all references to the routine being called or the module being initialized.

When an invariant is examined and found to be relevant, in accordance with one of the four cases above, every variable in the invariant becomes an input variable to the procedure and a declaration for that variable must be generated if not already present. This process may add variables to the input list of the procedure, in which case further invariants may be brought in. A transitive closure step is required to implement this. Fortunately, this step can be implemented during the transitive closure phase, and by adding to the input lists of routines during this phase, an input list can be constructed which contains all relevant variables. Then, when processing of a routine begins, invariants need simply be tested against the input list; all relevant invariants will be brought in at that time.

Modules are divided into two classes; those with module bodies and those without. Modules without module bodies are forbidden to have local variables or invariants, and an INIT operation on such a module is illegal. For modules with bodies, it is illegal to call a procedure exported from the module unless the INIT operation has previously been performed on the module. This is enforced by generating a REQUIRE at each call to an exported procedure requiring that the name of the module is defined. Conversely, after an INIT, the name of the module is announced as defined in the NEW. The name of the module is of type ‘‘module’’, and is considered to have a ‘‘d’’ (defined) part of a simple boolean, but no ‘‘v’’ part.

INIT statements

Our basic assumption about initialization of modules is that modules are initialized from the inside out. We have an implicit invariant that if module A contains module B, and module A is initialized, then module B is initialized. This invariant is made valid by generating Jcode in accordance with the statements following.

- At entry to the initialization block of a module, it can be assumed that the modules of the next level inward are not initialized. The big NEW at the beginning of a junit reflects this.
- At exit from an initialization block, all inner modules must have been initialized. REQUIRE jcode is generated to check this.
- It is required that a module not be initialized when its INIT statement is executed; a REQUIRE is generated by an INIT for this purpose.
- Conversely, after the return from the INIT statement, one may assume that the module is initialized, and the NEW generated by the INIT reflects this.

Definedness

Our basic model of definedness associates a Boolean flag with each variable in the program. Initially, the value of the flag is unknown. Whenever a value is assigned to the variable, the flag is set to true. At any reference to the variable in code, the flag is tested and if the flag is not known to be true, an error is indicated.

Note the assumptions here. It is assumed that variables, once defined, stay defined. The one exception to this rule is the FOR statement, for which the value of the definedness flag related to the iteration variable is explictly forced to FALSE at exit from the FOR loop.

Note also that errors of ommission by the Jcode generator in checking for definedness could easily result in unsound verifications, since the type rules in the theorem prover essentially state as axioms that variables are always in range for their types.

There are some nagging questions about soundness here, in that we check for definedness as a separate operation applicable only to executable code and then assume that all variables are in range. Our argument for soundness here comes from the fact that that the verification would be sound if, at initialization, all variables were initialized to unknown but inrange values. Although the implementation does not work that way, the user cannot discover that fact because, in executable code, the verifier prohibits the program from using a variable whose defined flag is not set. Therefore it is permissible to verify against the fiction that variables come up with inrange but unknown values.

Structured variables require special treatment. Records and arrays are considered to have corresponding records of booleans and arrays of booleans associated with them as definededness flags. We allow the user to express such concepts as ‘‘defined(arrayname)’’ but there is no flag variable associated with the entire array. Even worse, an array may be composed of further structures. Bearing in mind that our system does not support quantifiers, this is a difficult problem to deal with.

The key to this problem is to provide a constructor in Jcode which will allow the construction of arbitrary structures containing elements entirely of value ‘‘true’’. We begin by defining

```
(emptyobject!)
```

as a dummy array object about which we know nothing. This gives us something to use ‘‘storea!’’ and ‘‘storer!’’ on when constructing constants. For example, with the definitions

```
type rec = record
        f1: integer;
        f2: char;
        end;

var rr, ss: rec;
```

and the statement

```
rr := ss;
```

where ss is alread known to be defined, we can generate the Jcode

```
ASSIGN (rr) (rr) (storer!
   (storer! (emptyobject!) rec f1 (true!)) rec f1 (true!))
```

which will create the appropriate definedness object for rr. Note that this will work for replacements into arbitrary record substructures.

Arrays are harder. We define

```
(arrayconstruct! v i j)
```

by the recursive Boyer-Moore definition

```
DEFN(arrayconstruct! (i j v)
  (IF (LESSP J I)
        (emptyobject!)
        (storea! (arrayconstruct! (ADD1 i) j) i v)))
```

with which we can clearly create sequential arrays of ‘‘true’’. Note that since ‘‘v’’ is untyped, we can use ‘‘arrayconstruct!’’ to create arrays of arrays, arrays of structures, and so forth. For example,

```
var tab: array [1..100] of rec;
```

(remembering the definition of ‘‘rec’’ above), if fully defined, would have a definedness part value of

```
(arrayconstruct! 1 100
  (storer! (storer! (emptyobject! rec f1 (true!)) rec f2 (true!))))
```

With these constructs, we can create the definedness value of any Pascal-F object. We now need means of testing the definedness of arbitrary objects. For simple variables, we would like the definedness part to have the value ‘‘true’’. This also applies to any leaf of a variable, i.e. the lowest level record fields or array elements. Since we create the definedness values for records with our usual ‘‘storer!’’ primitive, record field definedness needs no special support. Array element definedness does.

Let us define

```
(alltrue! x)
```

to the simplifier such that, based on the available type information,

```
if x is a simple variable, (alltrue! x) = x
if x is an array, (alltrue! x) = (arraytrue! x lowbound highbound)
if x is a record, (alltrue! x) =
  (and! (alltrue! x.f1) (alltrue! x.f2 ...) for all fields.
```

It is clear that the ‘‘defined’’ operator at the user level translates into Jcode of

```
defined(x)  -->  (alltrue! (defined! x))
```

in the general case. However, we will continue to generate Jcode such that

```
defined(x)  -->  (defined! x)
```

when x is a simple variable.

We define

```
(arraytrue! a i j)
```

by

```
DEFN(arraytrue! (a i j)
  (IF (lti! j i)
        T
        (IF (alltrue! (selecta! a i))
          (arraytrue! a (addi! i 1) j)
          F)))
```

and we make ‘‘alltrue!’’ an uninterpreted function to the Boyer-Moore prover with

```
DCL(alltrue! (A I J))
```

The user needs access to ‘‘arraytrue!’’ in Pascal-F so that loop invariants can be written. We will allow a 3-argument form of ‘‘defined’’ in Pascal-F such that

```
defined(a,i,j)
```

(which will be allowed only when a is an array) generates

```
(arraytrue! (defined! a) (i) (j))
```

in Jcode. Thus, a user might write, when initializing an array in Pascal-F,

```
var tab: array [1..100] of integer;
   i: 0..100;
...
   for i := 1 to 100 do begin
           tab[i] := 0;
           state(defined(tab,1,i));
           end;
   assert(defined(tab));
```

We would need some lemmas to prove the above correct. These are

```
PROVE.LEMMA(arraytrue.first.rule (REWRITE)
   (IMPLIES (alltrue! (selecta! a i))
              (arraytrue! a i i)))
```

which handles the base case (i = 1),

```
PROVE.LEMMA(arraytrue.extend.rule (REWRITE)
   (IMPLIES (AND (arraytrue! a i j)
                     (alltrue! (selecta! a i (addi! 1 j))))
              (arraytrue! a i (addi! 1 j)))
```

which handles the induction step (i := i + 1), and

```
PROVE.LEMMA(allarray.unchanged.rule (REWRITE)
   (IMPLIES (AND (arraytrue! a i j)
                     (OR (gti! x i) (lti! x j)))
              (arraytrue! (storea! a x y) i j)))
```

which is used to show that the ‘‘arraytrue’’ property is not destroyed by the addition of the new element. The proof of the final assertion ‘‘defined(tab)’’ can be handled by the information built into the simplifier.

The lemmas given above are sufficiently useful that they should be part of the standard database. These suffice for the initialization of arbitrary structures, provided that initialization proceeds from the inside out (one must prove ‘‘defined(a.b)’’ before ‘‘defined(a)’’) and arrays are initialized in order starting with the first element. More elaborate schemes require proof of more lemmas with the Boyer-Moore prover, but do not require new definitions.