Pascal-F Verifier Internal Design Document

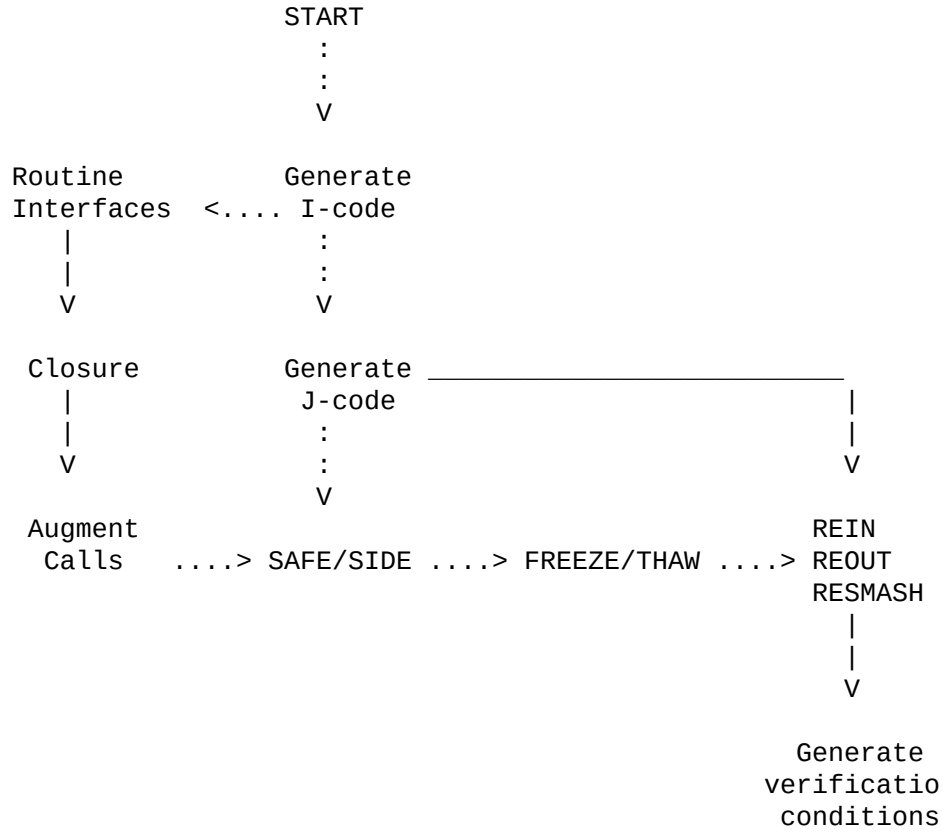
CPCI #3 -- Verification Condition Generator

Scott Johnson

FACC / Palo Alto

1. An overview

The following steps must be performed to generate verification conditions for a program. These steps do not have to be performed in a strictly sequential fashion. The ordering of these operations is shown in the following diagram.



If tasks are connected by a solid arrow, the first task must be completed before the second task is begun. If tasks are connected by a dotted arrow, the first task conceptually precedes the second task, though the two tasks could be combined into the same pass.

1.1 Routine Interfaces

The interface (as described in the Pascal-F manual) must be found for each routine. The interface information includes the entry and exit conditions, the number and types of all parameters, the global variables that are explicitly referenced or modified by the routine, and other routines that are directly called by the routine.

1.2 Closure

Transitive closures are performed on the interface database to determine the variables that are indirectly referenced and modified by each routine, and the routines that are indirectly referenced by each routine.

1.3 Call Augmentation

If a routine references or modifies a global variable, the variable is added as an argument to every call to that routine.

1.4 J-code Generation

From the I-code produced by pass 1 of the Pascal-F compiler another kind of intermediate code, called J-code because it follows the I-code, is generated. This process will be described in detail later.

1.5 SAFE and SIDE elimination

SAFE and SIDE are two types of J-instructions. These are eliminated from the J-code, and in some cases are replaced by other J-instructions.

1.6 FREEZE and THAW elimination

These two J-instructions imply certain syntactic checks. Those checks are performed, and the instructions are eliminated from the J-code.

1.7 REIN, REOUT, and RESMASH elimination

These instructions are eliminated from the J-code, being replaced in some cases by other J-instructions.

1.8 Verfication condition generation

The remaining J-code is used to generate verification conditions.

2. J-code

The I-code produced by the first pass of the Pascal-F compiler will be processed to form a database described herein. It is the intention that this database, information about types of variables, and some interface information about each procedure will be sufficient to generate the verification conditions.

The database will consist of a sequence of "J-instructions," collectively known as "J-code." In this document, J-code will be written in a function-operands notation similar to assembly language code. In fact, the techniques used to generate J-code from program text are quite similar to those used to by a compiler to produce assembly code. However, it should be made clear that the J-code is not a translation of the program into a form that could be executed by an interpreter. Rather, it is the input to a program that generates verification conditions for the program. For the time being, the actual machine representation for J-code will be left unspecified.

2.1 Basic J-code

We will begin the discussion of J-code by introducing the simplest of the J-instructions, and describing how to obtain verification conditions from a sequence of these instructions. The instructions that will be discussed are:

- PROCLAIM P, where P is a formula
- REQUIRE P, where P is a formula
- SMASH V, where V is a variable
- LABEL L, where L is a unique label
- FORK L1,L2,...Ln, where L1,L2,...Ln are one or more labels
- BREAK

No two LABEL instructions may contain the same label, and every label on every FORK must refer to a LABEL instruction.

Roughly speaking, a PROCLAIM corresponds to a fact about the program that can be deduced from the semantics of the programming language. A REQUIRE is a fact that must be true for the program to be

correct. A SMASH represents the modification of a variable. The branching information about the program is expressed by FORK and LABEL; LABELs correspond to GOTOs and FORKs correspond to comefroms [sic].

Every verification condition will be produced from a sequence of REQUIREs, PROCLAIMs, and SMASHes in which there is exactly one REQUIRE, which appears at the end of the sequence.

The first step in creating the verification condition is the elimination all the SMASH instructions. Begin this step by prepending to the sequence a SMASH instruction for every variable in every PROCLAIM and REQUIRE in the sequence. Now number the SMASHes 1, 2 ... n. Some scheme must be used to associate a name with each SMASH; the combination of the name and the variable being SMASHed must uniquely identify the SMASH. The indices just assigned would work, but it is more mnemonic to use, say, the line number of the statement from which the SMASH was generated. The notation NAME[i] will be used to denote the name of the i'th SMASH.

Next, for each instruction of the form PROCLAIM Q or REQUIRE Q in the sequence, every program variable X in Q is replaced by X@NAME[j], where j is the largest integer such the j'th SMASH precedes the instruction containing Q. (Because of the prepended SMASHes, an appropriate j will always exist.) Some J-instructions will contain variables of the form X(+) or X(-), where (+) and (-) are special markings on the variable. These variables are replaced by are replaced by X@NAME[j+1] and X@NAME[j-1] respectively.

Because the user may see (and the implementers will certainly see) verification conditions, it is desirable that variables not be unnecessarily decorated. Therefore, for each class of variables:

{X@name[J1], X@NAME[J2], ... X@NAME[Jn]}

appearing in the sequence, let M be the maximum of J1, J2, ... Jn and replace every occurrence of X@NAME[M] by just X. After this modification, the unembellished version of the variable X will represent the "most recent" value.

After this renaming operation is complete, let P1, P2, ... Pn be the formulas in the modified PROCLAIMs and Q be the formula in the modified REQUIRE. The complete verification condition is then:

P1 AND P2 ... AND Pn IMPLIES Q

As we will see later, there are ways of simplifying this formula.

Here is how to derive all the verification conditions implied by a sequence of J-instructions. A set of verification conditions is derived from each REQUIRE in the sequence by building a collection of J-instruction sequences and deriving a verification condition from each sequence. A J-instruction sequence is obtained by beginning at the REQUIRE and reading the J-code in reverse. As PROCLAIM and SMASHes are encountered they are added to the beginning of the sequence being built. When a LABEL is encountered, it is skipped over. When a FORK is encountered, one of the labels on the FORK is nondeterministically chosen, and that label is branched to. When a BREAK is reached, the sequence being built is complete. If r is a REQUIRE instruction, let S(r) be the set of J-code sequences obtainable by starting at r and making all possible choices at the FORKs. For every REQUIRE r in the J-code, a verification condition is generated for every member of S(r).

For the verification condition generation process to be well-defined, it is necessary that the LABELs and FORKs must be positioned so that they do not introduce loops, and that the first instruction in the J-code is a BREAK.

There are J-instructions other than the ones described in this section. Some of these are simply abbreviations for lists of other J-instructions. Others are used to perform checks that do not require a theorem prover. The rest of the J-instructions are described in the following sections.

2.2 SAFE

The format of this J-instruction is:

SAFE *v1,*v2, ... vn,E1,E2, ... Em

where each vi is the name of a variable, and each Ei is an expression or procedure call. The SAFE instruction is used to check that it is safe to evaluate E1, E2, ... Em together in the same statement in which variables v1, v2, ... vn are modified. Because of the rules about conflicting side effects, it is necessary for the SAFE instruction to have as operands all the expressions in the statement.

Before verification conditions are generated, a pass must be performed over the J-code in which each SAFE is replaced by a sequence of REQUIREs. During this pass, certain SAFE instructions can be determined to indicate errors without consulting the theorem provers. The following processing is performed on each SAFE instruction.

1. For every variable reference v in every expression Ei (except for VAR parameters to routines), the instruction

   REQUIRE DEFINED(v)

   is produced. If a component of a structured variable is referenced, only the component is checked for definedness. However, if the entire structure is referenced without selection, the entire structure is checked for definedness. Here are two examples:

```
A[K].E.F > B[J]          DEFINED(A[K].E.F)
                         DEFINED(B[J])
                         DEFINED(K)
                         DEFINED(J)

A                        DEFINED(A)
```

2. For every subexpression of the form A DIV B or A MOD B in the expressions, REQUIRE B<>0 is generated.
3. For every subexpression of the form A[I] in the expressions, REQUIRE L<=I and REQUIRE I<=U are generated, where L and U are the array bounds. Note that a two dimensional array in Pascal is syntactic sugar for an array of arrays.
4. For every subexpression of the form R.F, where F is a variant field, a REQUIRE R.T=c is generated, where T is the tag field for the variant, and c is the case constant associated with the field F.
5. For every call to a user-defined routine, a REQUIRE E is generated, where E is the entry condition with the actual arguments substituted for the formals. REQUIREs are also generated to ensure that value parameters are both DEFINED and have a value appropriate to the type of the parameter if it is a subrange. In addition, if the routine call is recursive, a check is made to ensure termination of the recursion. See the section on routine definitions for details.
6. A check is made to ensure that the set of variables marked with asterisks on the SAFE J-instruction and the sets of variables modified by each procedure and function call are all pairwise disjoint. If a component of a structured variable is passed as a VAR argument, the entire variable is considered modified for the purposes of this check.
7. For each variable modified by a routine call, a check is made to ensure that the the only place in the J-instruction the variable appears is as an argument to that call. This is the only check for which one big SAFE is not equivalent to a number of small ones.
8. Each routine call is checked for aliasing. For every pair of VAR arguments to the call one of the following conditions must hold:
   - The arguments are both readonly.
   - The arguments are variable references with distinct roots. The root of a simple variable is defined to be the variable itself. The root of A[I] or A.F is defined to be the root of A.
   - Both arguments are variable references with the same root, and at least one is writable. In this case, a REQUIRE may need to be generated. Each variable reference is formed from a sequence of selectors. For example, the reference A[I].F[K] is formed from the sequence

I,F,K (note that F is a field name, whereas I and K are subscript expressions). The variable A has a null sequence of selectors. Get the selector sequence for each variable reference, and truncate the longer one so that both sequences have the same length. By the type rules of Pascal, subscripts must correspond to subscripts and field selectors must correspond to field selectors. If any two corresponding field selectors differ, no REQUIRE need be generated. Otherwise, delete the field selectors from the sequences, leaving two sequences S1,S2 ... Sn and T1,T2 ... Tn of subscripts. Generate:

REQUIRE S1<>T1 OR S2<>T2 ... Sn<>Tn

The special case of n=0 would result in REQUIRE FALSE. This case can be flagged as an error without consulting the theorem prover.

- An additional aliasing check must be made if variant records are involved. If the tag field of a variant record is passed as a writable VAR argument, none of the variants controlled by that tag (or components thereof) can be passed as a VAR argument, not even on a readonly basis.

2.3 SIDE

Just as each SAFE instruction must be replaced by REQUIREs before verification conditions can be generated, SIDE instructions must be replaced by SMASHes and PROCLAIMs. The format of the SIDE instruction is SIDE E, where E is an expression or procedure call. For each routine call in E, the following J-code must be generated.

First a PROCLAIM is generated from the EXIT condition of the routine. The following substitutions are performed (after the call has been augmented).

```
Value parameter           The corresponding actual
OLD(VAR parameter)        The corresponding actual
Writable simple VAR       The actual, with (+) appended
Writable VAR component    A newly created variable
Readonly VAR parameter    The corresponding actual
Function name             The actual function call
```

Next a sequence of PROCLAIMs must be generated for each component of a structured variable that is passed as a writable VAR argument. Let A[I1], A[I2], ... A[In] be all the components of A passed as writable VAR arguments. (Each Ii may correspond to a sequence of selections.) Let R1, R2, ... Rn be the new variables created for these parameters, as indicated above. Generate the following sequence of PROCLAIMs:

PROCLAIM T1 = UPDATE(A, I1, R1)
PROCLAIM T2 = UPDATE(T1, I2, R2)
...
PROCLAIM Tn-1 = UPDATE(Tn-2, In-1, Rn-1)
PROCLAIM A(+) = UPDATE(Tn-1, In, Rn)

where T1,T2, ... Tn-1 are also created variables. The UPDATE function is described under assignment statements. A sequence such as the one above must be generated for each structured variable that has one or more components passed as writable VAR args.

Finally, a SMASH instruction is generated for each variable that is modified by the call. If several components of the same structured variable is passed to more than one argument, only one SMASH instruction is generated for the variable.

Though it may look like we are spending a lot of effort in order to allow statements to have side effects, we would need the same mechanism to handle procedure calls even if side effects were prohibited. The J-code generated for a procedure call consists of simply a SAFE and a SIDE.

2.4 REIN, REOUT, and RESMASH

These three instructions always come as a matched set. Their formats are:

```
REIN     name
RESMASH  name
REOUT    name
```

The purpose of the name is to identify the three instructions that are part of each matched set. Each triple must have a unique name. These names have nothing to do with variable names.

The REIN must precede its corresponding REOUT. These two instructions delimit a region of the J-code. Before verification conditions are generated, the RESMASH is replaced by a sequence of SMASHes. The sequence consists of an enumeration of the set of SMASH instruction appearing in the region delimited by the REIN and REOUT. Note that no matter how many times a variable is SMASHed in the region, only one SMASH of that variable replaces the RESMASH.

The current design only uses REIN, REOUT, and RESMASH in a restricted fashion. First, REINs and REOUTs are properly nested. That is, for any two regions, either one completely contains the other, or the regions are disjoint. Second, the RESMASH associated with a region is contained within that region and no others.

If these constraints are made mandatory, the name can be eliminated from the three instructions. This change may simplify the design of the algorithm used to replace RESMASHes with SMASHes.

2.5 FREEZE and THAW

These two instructions do not effect verification condition generation; they are used to enforce syntactic checks. The formats are:

```
FREEZE v
THAW   v
```

where v is a variable. It is an error for a SMASH v to appear in the region between a FREEZE v and its corresponding THAW v. More precisely, every var is assumed to have a "temperature" that begins at zero. A sequential pass is made through the J-code in which a FREEZE lowers the temperature of the variable by a degree, and a THAW raises it by a degree. It is illegal to SMASH a variable that has a negative temperature.

Careful attention must be paid to the order in which the various J-instructions are processed. SMASHes that are inserted as the result of SIDEs are flagged if they refer to frozen variables. However, frozen variables *may* be referenced by SMASHes that are inserted as the result of RESMASHes.

3. J-code Generation

The process of generating J-code from program text is much like the one a compiler might perform to generate assembly language. In this section, all the statements of Pascal-F are listed, and the J-code to be generated is outlined.

3.1 Assignment Statements

The assignment statement is of the form:

V := E

Where V is a variable and E an expression of suitable types. The most general form of assignment statement is:

A[I1][I2] ... [In] := E

where n>=0. The rules of Pascal-F, of course, constrain the type of A when n>0. Assignment to fields of records fits into this general scheme. A record is treated as an array whose index type is the set of field names.

The J-code generated for the assignment statement is:

```
SAFE      *A, A[I1][I2] ... [In], E
REQUIRE   inrange(E, type(A[I1][I2] ... [In]))
PROCLAIM  DEFINED(A(+)[I1][I2][I3] ... [In])
PROCLAIM  A(+) = store(A,I1,
                  store(A[I1],I2,
                  store(A[I1][I2],I3,
                  store(...
                  store(A[I1][I2] ... [In-1],In,E) ... )))
SIDE      A[I1][I2] ... [In]
SIDE      E
SMASH     A
```

In the special case of n=0, the awful PROCLAIM simplifies to:

PROCLAIM A(+) = E

The REQUIRE concerning "inrange" is not necessary if E is a structured variable, because in this case the type rules of Pascal demand that E has the same type as the variable begin assigned.

There is a question concerning just what PROCLAIMs are necessary so that other parts of the program can use the fact that variables, if DEFINED, are INRANGE. In the simple case of:

V := E

once inrange(E, typeof V) is established before the assignment, it is a simple deduction from V(+) = E that inrange(V, typeof V) after the assignment. The same kind of reasoning presumably works for arrays if Oppen's decision procedure really is complete.

For the type information to be remembered across loops it will be necessary to augment the loop invariants with type information. It may be simpler to just add a bunch of clauses of the form:

DEFINED(V) IMPLIES INRANGE(V, typeof V)

to the hypothesis of every verification condition.

Some questions must be answered concerning the semantics of records and arrays in Oppen's prover. For example, under what circumstances are two arrays or records considered equal? Must their values agree everywhere, including over illegal subscript ranges? What assumptions are by the prover about type checking?

Another aspect of the language that has been ignored above is that of variant records. The following approach ought to work. The theorem prover will view a variant record as a long record containing a tag field and all the variant fields, without any sharing of storage. An assignment to the tag field is then treated like a sequence of assignments. First the entire record, including the tag field and all the variants, is smashed and undefined. Then the tag field is set to the appropriate value. The checking performed by the SAFE operation will ensure that a particular variant field will be accessed only when the associated tag field has the correct value.

Given this approach, the verifier will take a conservative view of the semantics of variant records. If a tag field whose value is x is given a different value and then restored, the verifier will consider the final values of the other fields to be undefined. Likewise, the verifier will consider the assignment statement:

rec.tag := rec.tag

to leave all the variant fields undefined. These two restrictions seem consistent with the enforcement of good programming style and, as nearly as I can determine, the ISO Draft standard.

3.2 The IF statement

There are two forms of the IF statement: with and without an else clause. (The details concerning the use of the punctuation BEGIN and END are not relevant here.) The J-code generated for an IF statement of the form:

IF B
THEN <then part>
ELSE <else part>

is:

```
SAFE      B
LABEL     1
PROCLAIM  B
SIDE      B
CODE      <then part>
LABEL     2
PROCLAIM  NOT B
SIDE      B
FORK      1
SIDE      B
CODE      <else part>
LABEL     3
FORK      2,3
```

There is no CODE instruction. CODE <construct> is notation meaning the J-code recursively generated for <construct>. The J-code generated for an IF without a matching ELSE is the same as that given above, except that the line CODE <else part> is deleted.

3.3 The CASE statement

The CASE statement is quite similar to the IF statement. The J-code generated for a CASE statement of the form:

CASE x OF
c1: <statement 1>
c2: <statement 2>
...
cn: <statement n>
END

is:

```
LABEL     1
SAFE      x
REQUIRE   x=c1 OR x=c2 OR ... x=cn
LABEL     2
PROCLAIM  x=c1
SIDE      x
CODE      <statement 1>
LABEL     3
FORK      1
PROCLAIM  x=c2
SIDE      x
CODE      <statement 2>
LABEL     4
...
FORK      1
PROCLAIM  x=cn
SIDE      x
CODE      <statement n>
LABEL     n+1
FORK      2,3, ... n+1
```

The ISO draft standard specifies that the constants c1, c2, ... cn must be distinct, which is important because no ordering of cases is implied by the above J-code.

3.4 The WHILE statement

To generate J-code for the WHILE loop:

WHILE B DO BEGIN
<pre-body>
MEASURE M
INVARIANT P
<post-body>
END

an extra variable and label. The variable c is needed to serve as a loop counter, and the label loopname is used to delimit the body of the loop. Each time J-code is generated for a new loop, a new variable and label not used elsewhere are required. The J-code is:

```
PROCLAIM    c=INFINITY
LABEL       1
FORK        1,2
SAFE        B
PROCLAIM    B
SIDE        B
REIN        loopname
CODE        <pre-body>
REQUIRE     P
REQUIRE     M>=0
REQUIRE     M<c
FORK        1
RESMASH     loopname
SMASH       c
PROCLAIM    P
PROCLAIM    c=M
CODE        <post-body>
REOUT       loopname
LABEL       2
FORK        1,2
PROCLAIM    NOT B
SIDE        B
```

The initial PROCLAIM of c=INFINITY is a bit of a kludge. The purpose of the REQUIRE c<M is to make sure that the loop measure decreases on every iteration of the loop. However, the path tracer will also insist on establishing c<M the first time through the loop. Therefore, we tell it that c=INFINITY so that it will not complain. Formulas involving the constant INFINITY will not be passed to the theorem prover.

The J-code given for the WHILE statement contains forward FORKs. These forks could conceivably be a problem when generating verification conditions. At the very least, they make it nonobvious that the FORK/LABEL structure is loop-free. These problems can be eliminated using the following equivalent J-code.

```
PROCLAIM    c=INFINITY
LABEL       1
RESMASH     loopname
SMASH       c
PROCLAIM    P
PROCLAIM    c=M
CODE        <post-body>
REOUT       loopname
LABEL       2
FORK        1,2
SAFE        B
PROCLAIM    B
SIDE        B
REIN        loopname
CODE        <pre-body>
REQUIRE     P
REQUIRE     M>=0
REQUIRE     M<c
FORK        1,2
PROCLAIM    NOT B
SIDE        B
```

The cost of producing this J-code is that the order of the code for the the pre- and post-body is reversed from the order in which it appears in the program source. The standard push-down transducer method cannot be used for generating this kind of code.

3.5 The REPEAT/UNTIL loop

The J-code generated for the REPEAT/UNTIL loop:

```
REPEAT
<prebody>
MEASURE   M
INVARIANT P
<postbody>
UNTIL B
```

is:

```
PROCLAIM  c=INFINITY
LABEL     1
FORK      3
PROCLAIM  NOT B
SIDE      B
LABEL     2
FORK      1,2
REIN      loopname
CODE      <prebody>
REQUIRE   M>=0
REQUIRE   c<M
REQUIRE   P
FORK      1
RESMASH   loopname
SMASH     c
PROCLAIM  P
PROCLAIM  c=M
CODE      <postbody>
REOUT     loopname
SAFE      B
LABEL     3
PROCLAIM  B
SIDE      B
```

As was the case with WHILE loops, an extra variable c is needed to prove termination. As with the WHILE loops, forward FORKs can be eliminated at the cost of reordering the prebody and postbody.

3.6 FOR loops

There are two forms of FOR loops, upward loops:

```
FOR i := lo TO hi DO
BEGIN
   <prebody>
   INVARIANT P
   <postbody>
END
```

and downward loops:

```
FOR i := hi DOWNTO lo DO
BEGIN
   <prebody>
   INVARIANT P
   <postbody>
END
```

To generate J-code for a FOR loop, two new variables whose names do not appear elsewhere must be created. In the following, lo' and hi' will be used to indicate these variables. The J-code generated for both loops is the same except for three PROCLAIMs. For these three PROCLAIMs, the version for downward loops is shown in brackets.

```
SAFE        lo,hi
PROCLAIM    lo'=lo
PROCLAIM    hi'=hi
SIDE        lo
SIDE        hi
LABEL       1
PROCLAIM    i(+)=lo'      {i(+)=hi'}
LABEL       2
FORK        2,4
REIN        loopname
SMASH       i
PROCLAIM    lo'<=i
PROCLAIM    i<=hi'
REQUIRE     inrange(i, typeof i)
FREEZE      i
CODE        <prebody>
REQUIRE     P
FORK        1
RESMASH     loopname
PROCLAIM    P
CODE        <postbody>
THAW        i
REOUT       loopname
LABEL       3
PROCLAIM    i(+)=i+1      {i(+)=i-1}
LABEL       4
FORK        3
PROCLAIM    i=hi'         {i=lo'}
LABEL       5
FORK        1
PROCLAIM    lo'>hi'
LABEL       6
FORK        5,6
SMASH       i
PROCLAIM    NOT DEFINED(I)
```

The verifier will not complain if the final "undefined" value of the the index variable is in reality inappropriate for its subrange type. We must be sure that these semantics are consistent those of the Pascal-F compiler. Likewise, a loop such as:

FOR i := 1 to N

where i is declared to be 1..10, is considered valid as long as N<=10. In particular, the case N=0 is allowed.

3.7 The WITH statement

This statement is handled by using a macro expansion approach. Given the construct:

WITH R DO <statement>

The following J-code is generated:

FREEZE <subscript variables in R>
CODE <macro processed statement>
THAW <subscript variables in R>

The subscript variables in R are all variables appearing in subscript expressions in R. The macro processing performed on a statement is replacing every occurrence of F by R.F, where F is a field name associated with the type of R.

This approach to the WITH statement will ensure that a verified program will execute the same no matter whether the "static" or "dynamic" semantics are chosen for WITH.

3.8 ASSERT statements

The J-code generated by a statement of the form:

```
ASSERT P
```

is:

```
REQUIRE   P
PROCLAIM  P .
```

Note that if the theorem prover is a complete decision procedure, the use of this statement will not assist the verification effort any because it isn't telling the theorem prover anything it couldn't figure out on its own. However, by the time we finish this project I deem it unlikely that the theorem prover being used will be a complete decision procedure.

3.9 The STATE statement

The STATE statement is similar to the ASSERT statement, except that it causes a break. The J-code generated for STATE P is:

```
REQUIRE   P
BREAK
PROCLAIM  P
```

3.10 Procedure Calls

Procedure calls and function calls will be handled using the same mechanism. Thus, the J-code generated for a procedure call of the form P (the metavariable P includes the procedure parameters) is:

```
SAFE   P
SIDE   P
```

3.11 Routine definitions

Associated with every routine, there will be an ENTRY and EXIT assertion. In addition, if the routine is recursive, there will be a DEPTH expression associated with the routine. After call augmentation has been performed, the only variables contained in these assertions and the expression ought to be be parameters to the routine. A dependency on an irrelevant variable is not (strictly speaking) an error, but it will be flagged because it is bad form.

A separate file of J-code is produced for each routine definition, the main body of the program. These file can be combined by separating the J-code for each routine with BREAKs.

To generate J-code for a routine definition, a number of extra variables are required. An extra variable is required for each VAR parameter. Let p1, p2, ... pn be the VAR parameters and v1, v2, ... vn be the corresponding extra parameters. For recursive routines, an extra variable (d0) is required for the DEPTH expression. The J-code for a routine definition is:

```
PROCLAIM   <entry condition>
PROCLAIM   <range of value parameters>
PROCLAIM   <definedness of value parameters>
PROCLAIM   d0 = <depth expression>
REQUIRE    d0 >= 0
PROCLAIM   v1 = p1
PROCLAIM   v2 = p2
...
PROCLAIM   vn = pn
CODE       <routine body>
REQUIRE    <exit condition>
```

The range information of value parameters is taken from the subrange declarations for those parameters. Any assumptions about the definedness and range of value parameters must appear explicitly in the entry assertion.

The variables v1, v2, ... vn are used to interpret the pseudofunction OLD. Whenever OLD(pi) appears in an assertion, invariant, etc., it is replaced by vi. In particular, this substitution is performed in the final REQUIRE.

The purpose of the variable d0 is to prove the termination of recursion. Let Q be the routine whose body is being processed. As part of the processing of the SAFE instruction, for every recursive call to a routine P within the the body of Q, the following J-instruction:

REQUIRE D<d0

is generated. In the REQUIRE, D is the depth expression for P (not Q) with the actual arguments of the call substituted for the formal parameters of P appearing in D, and d0 is the extra variable created for Q.

4. Open issues

This section contains a list of issues that we still do not know what to do with, or were simply forgotten when the preceding was written.

4.1 Multiprogramming Statements

There are three multiprogramming statements: WAIT, SEND, and INIT. Something special will also have to be done about functions and procedures exported from monitors. It is not entirely clear to me at this time what to about these statements.

4.2 Conditional Expressions

When generating REQUIRES that check the legality of expression evaluation, special attention must be given to conditional expressions. For example, the expression:

IF X < 0 THEN A*X ELSE A/X

requires NOT(X > 0) IMPLIES X <> 0, and not X <> 0, as a precondition.

4.3 Definedness of functions

At the end of the body of any function F, the assertion DEFINED(f) must hold.

4.4 The dynamics of variant records

The J-code generated for constructs that change the value of the tag fields of variant records is not discussed.

4.5 Soundness of termination using scaled arithmetic

We must convince ourselves that it is.

4.6 Type information

The verification condition generator will make use of what Steve German calls the LESSDEF lemma, which says that any variable that is DEFINED has a value appropriate to its type. This lemma is proved valid by induction on the computation sequence. The VCG will output required instantiations of the LESSDEF lemma. To avoid circular reasoning, it is important that the REQUIRES that establish that a value being assigned to a variable not make use of the LESSDEF lemma. I believe it is safe to first require the variable to be in range, and then PROCLAIM that is DEFINED.