Pascal-F Verifier Internal Design Document

"Jcode" Interface Control Document

Scott Johnson

FACC / Palo Alto

1. Introduction

This document has two parts. The first part precisely defines the syntax of the language processed by the Pascal-F verification condition generator. The second part relates J-code to Pascal-F. It describes the J-code that is to be generated for each of the Pascal-F constructs.

Comments in this document that are marked with asterisks in the right margin do not concern the definition of J-code, but rather the state of the implementation of the verification condition generator, the program that processes the J-code. If a feature has an I in the right margin, that feature is not implemented. * *

2. Syntax of J-code

What follows is a context free grammar that describes the syntax of j-code. The meaning of the productions should be reasonably clear to anyone familiar with the Backus-Naur notation first used to define Algol 60. A description of the notation is given here for the sake of completeness.

The (printable) ascii characters are divided into two classes: "plain" characters, and "metacharacters." The metacharacters are space, newline, |, <, >, {, }, *, #, ~, and $. The plain characters are all the ascii characters that are not metacharacters. A metavariable is defined to be either string of characters that is enclosed in angular brackets (but does not contain any other brackets nested within the outer pair) or one of the metacharacters #, ~, or $. A symbol is defined to be a metavariable or plain character.

A production has three parts, which are written in sequence. The parts of a production are:

1. a metavariable
2. the string ::=
3. a series of alternatives separated by |.

There are two kinds of alternatives, basic and extended. A basic alternative is a series of symbols. An extended alternative may also contain one of the following two repetition constructs:

1. a symbol followed by *
2. a list of symbols preceded by { and followed by }*.

Within productions, spaces and newlines are insignificant, except that spaces are significant within metavariables. When two productions are written one after the other, the beginning of the second production is identified by the convention that any line containing the string ::= starts a new production.

A valid j-code string is one that can be derived by the following procedure. Begin with the metavariable <j-code>; call that the "current string." Then repeat the following steps until the current string contains no metavariables. Select a metavariable in the current string, and make a copy of one of the alternatives for that metavariable. If the alternative contains any repetition constructs, replace each repetition construct with zero or more copies of the symbol before the * (for the first kind of repetition) or zero or more copies of the string of symbols enclosed in brackets (for the second kind). After all the repetition constructs have been so removed, delete the selected metavariable from the current string, and replace it with the modified alternative.

2.1 Basic punctuation

<empty> ::=
<separator> ::= <space> | <tab> | <line end>

::= <separator>* <space>
$ ::= <separator>* <line end>
˜ ::= # | <empty>

<line end> ::= <optional comment> <newline>
<optional comment> ::= <empty> | <separator> -- <character>*

Unlike most language descriptions, this one specifies the language exactly, down to the placement of spaces and newlines. The metavariables ˜, #, and $ have been made short so that they do not obscure the overall structure of the rest of the syntax. The symbol # indicates a required separation, and ˜ indicates an optional separation. A J-statement can be broken across lines anywhere # or ˜ appears in the syntax, but the continuation line must begin with a space. The symbol $ indicates the end of line; any symbol appearing after $ in the syntax must appear at the beginning of a line.

Anywhere the syntax allows a newline, it also allows a comment, which begins with -- and ends with the end of the line.

2.2 Overall syntax of j-code

```
<j-code> ::= <j-unit>*
<j-unit> ::= BEGIN # <unit name> $
          <declaration part> <statement part> END $
<unit name> ::= <identifier>
<declaration part> ::= <declaration>*
<declaration> ::= <variable> ˜ : ˜ <form> $
<form> ::= ( ˜ <class> ˜ <type> ˜ )
<class> ::= variable | function | rulefunction

<statement part> ::= <break statement> <A bal B>

<A bal A> ::= <empty> |
   <A bal A> <simple statement> |
   <A bal B> <catch statement> |
   <A bal A> <rein statement> <A bal A> <renew statement>
          <A bal A> <reout statement> |
   <A bal B> <rein statement> <B bal A> <renew statement>
          <A bal A> <reout statement>

<B bal A> ::= <B bal A> <simple statement> |
   <B bal B> <catch statement> |
   <B bal A> <rein statement> <A bal A> <renew statement>
          <A bal A> <reout statement> |
   <B bal B> <rein statement> <B bal A> <renew statement>
          <A bal A> <reout statement>

<A bal B> ::= <A bal A> <throw statement> |
   <A bal A> <rein statement> <A bal A> <renew statement>
          <A bal B> <reout statement> |
   <A bal B> <rein statement> <B bal A> <renew statement>
          <A bal B> <reout statement>

<B bal B> ::= <empty> |
   <B bal A> <throw statement> |
   <B bal A> <rein statement> <A bal A> <renew statement>
          <A bal B> <reout statement> |
   <B bal B> <rein statement> <B bal A> <renew statement>
          <A bal B> <reout statement>

<throw statement> ::= <split statement>
              | <branch statement>
              | <hang statement>

<catch statement> ::= <when statement>
              | <join statement>
```

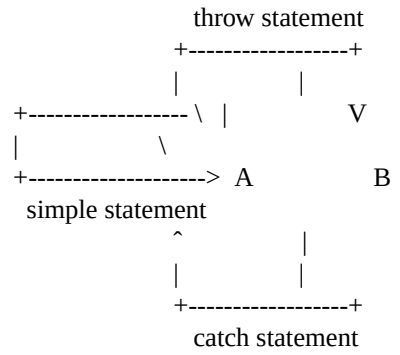
A file of j-code consists of a sequence of units. The j-code generator produces a unit for each routine. It is recommended that the name of the unit be the name of the routine, but the unit name is used for identification only. It does not affect the verification conditions produced from the unit.

A J-unit has two parts: one part containing declarations, and one containing J-statements. No variable may appear more than once in the declaration part, and every variable appearing in the statement part must also appear in the declaration part.

The statement part consists of a sequence of J-statements. A J-unit must begin with a break statement. This initial break ensures that all verification conditions terminate; it also delimits the end of the declaration

part.

The productions using metavariables of the form <x bal y> are somewhat confusing because they simultaneously describe two different ordering constraints. The first constraint is a finite state machine, which is used to ensure that a J-unit consists of a series of blocks that begin with a <catch statement> and end in a <throw statement>. The machine has two states: A and B, and has the following state transitions:



Every J-unit begins with a break statement, which puts the machine into state A. When the unit ends, it must be in state B.

The other order constraint is that the REIN, RENEW, and REOUT statements must be balanced in the same fashion as if-then-fi in Algol. What makes all this tricky is that although RENEW statements can only occur in state A, rein and reout statements can occur in any state, and do not change the state.

These are expressed in the grammar using the four variables of the form <x bal y>, where x and y may each be A or B. The meaning of <x bal y> is a sequence of statements that are balanced with respect to REIN, RENEW, and REOUT, and change the finite state machine from state x to state y.

2.3 J-statements

```
<j-statement> ::= <simple statement> | <throw statement>
            | <catch statement> | <rein statement>
            | <renew statement> | <reout statement>

<simple statement> ::= <require statement>
                | <new statement>
                | <assign statement>
                | <proclaim statement>
                | <break statement>

<require statement> ::= REQUIRE # <expression>
                        ~ (/ <string> /) $
<new statement> ::= NEW # <variable list> ~
                <expression> ~ (/ <string> /) $

<assign statement> ::= ASSIGN # <variable list> ~
    <selector expression> ~ <expression> ~ <expression> $

<proclaim statement> ::= PROCLAIM # <expression> $
<split statement> ::= SPLIT # <label> $
<when statement> ::= WHEN # <expression> ~ <label> $

<join statement> ::= JOIN # <label> $
<branch statement> ::= BRANCH # (/ <string> /)
                    ~ <label> $
```

```
<label> ::= <nonzero digit>
        | <nonzero digit> <digit>
        | <nonzero digit> <digit> <digit>
        | <nonzero digit> <digit> <digit> <digit>

<hang statement> ::= HANG $
<break statement> ::= BREAK ˜ (/ <string> /) $

<variable list> ::= ( ˜ <variable sequence> ˜ )

<variable sequence> ::=
   <empty> | <variable sequence> # <variable item>

<variable item> ::=
    <variable>  | <variable> ˜ : ˜ <form>

<rein statement> ::= REIN $
<renew statement> ::= RENEW <expression> $
<reout statement> ::= REOUT $
```

The syntax (though certainly not the semantics) of the J-statements should in most cases be clear from the productions. The syntax requires every statement to begin at the beginning of a line; continuation lines must begin with at least one blank.

There are syntactic restrictions (which are not enforced by the production rules) on the labels of the SPLIT, JOIN, WHEN, and BRANCH statements. A label appearing on a split or join statement cannot appear on any other split or join statement in the same J-unit. That is, if a label appears on a join statement, it cannot be used on any other SPLIT or JOIN statement, and the same rule applies to labels on split statements. The label on each when statement must match the label on some split statement in the same J-unit, and every split statement must have at least two matching when statements. Similarly, the label on each branch statement must match the label on some join statement in the same J-unit, and every branch statement must have at least one matching when statement.

Each J-unit must free of circularities, which are defined as follows. Every J-statement is defined to have zero or more successors. A HANG statement has no successors; all other J-statements have at least one successor. The successors of a split statement are the WHEN statements in the same J-unit with matching labels. The successor of a branch statement is the JOIN statement in the same J-unit with a matching label. In all other cases, the successor of a statement is the first immediately following it in the J-unit that is not a REIN or REOUT statement. A circularity is defined as a sequence of statements S1, S2, ... Sn such that S1 is a successor of Sn, and for all i in {1..n-1}, Si+1 is a successor of Si. The verification condition generator * infinitely loops when given J-code that contains a circularity.

2.4 Types

```
<type> ::= <enumerated type> | <other type>
<enumerated type> ::= <subrange type> | <boolean type>
<subrange type> ::= ( ˜ subrange # <low bound>
                        # <high bound> ˜ )
<boolean type> ::= ( ˜ boolean ˜ )
<other type> ::= ( ˜ integer ˜ )
                    | ( ˜ universal ˜ )
                    | ( ˜ module ˜ )
   | ( ˜ fixed # <low bound>                                          I
          # <high bound> # <precision> ˜ )                           I
   | ( ˜ set # <enumerated type> ˜ )                                  I
   | ( ˜ array # <index type> # <result type> ˜ )
   | ( ˜ record # <record name> # <field list> ˜ )

   <low bound> ::= <integer>
   <high bound> ::= <integer>
   <precision> ::= <integer>                                          I
   <index type> ::= <enumerated type>
   <result type> ::= <type>
   <type list> ::= <type> { # <type> }*
   <field list> ::= <field> { # <field> }*
   <field> ::= ( ˜ <field name> # <type> ˜)
   <record name> ::= <identifier>
   <field name> ::= <identifier>
```

The given types are a subset of the types found in Pascal-F. The subset is supposed to be the universe of types that make their way into i-code. Types can be written in declarations only.

2.5 Expressions

```
<expression list> ::= ( { ˜ <expression> }* ˜ )
<expression> ::= ( ˜ <operator> ˜ )
    | ( ˜ <operator> # <expression>
              { ˜ <expression> }* ˜ )
    | ( ˜ consti! # <integer> ˜ )
    | ( ˜ constf! # <integer> # <precision> ˜ )                        I
    | ( ˜ selectr! # <expression>
             # <field name> ˜ )
    | ( ˜ storer! # <expression>
            # <field name>
            # <expression> ˜ )

<operator> ::= <modifier> <variable> | <builtin>
<builtin> ::= <identifier> !
<variable> ::= <identifier>
<modifier> ::= <empty> | defined! # | new! #
        | defined! # new! #
```

Field names must be unique over all records.

The syntax for expressions is similar to that used by Boyer and Moore. Expressions always begin and end with parentheses. The value of a variable is denoted by enclosing the variable in parentheses. A variable appearing outside parentheses (in NEW statements, for example) denotes itself.

Builtin operators are distinguished from user variables, functions, and procedures by the convention that builtins always end with an exclamation mark. Every builtin operator takes a fixed number of arguments. The builtin operators are listed in the next section.

Some of the builtin operators have a special syntax. The operators consti! and constf! are used to build integer and fixed point constants. Note that the integers indicating the value of the constant are not enclosed in parenthesis.

2.6 Builtin operators

The following is a list of the builtin operators, organized by the type of their operands. Each operator takes a fixed number of operands. The following codes are used to indicate the types of operands for which the operator yields meaningful results:

I - integer
F - fixed point
B - Boolean
S - set
A - array
R - record
E - any of the above
C - integer constant
V - variable
T - type
N - field or record name

The list of operators should be considered tentative; it will be finalized once we have a firmer understanding of what i-code is.

2.6.1 Integer operators

(consti! C) integer constant
(addi! I1 I2) addition
(subi! I1 I2) subtraction
(negi! I) negation
(muli! I1 I2) multiplication
(divi! I1 I2) integer division
(mod! I1 I2) remainder
(odd! I1) odd
(gei! I1 I2) greater or equal
(lei! I1 I2) less or equal
(gti! I1 I2) greater than
(lti! I1 I2) less than
(mini! I1 I2) minimum
(maxi! I1 I2) maximum

2.6.2 Fixed point operators

(constf! C1 C2) scaled constant
(scale! F1 C1) rescaling operator
(addf! F1 F2) scaled addition
(subf! F1 F2) scaled subtraction
(negf! F) scaled negation
(mulf! F1 F2) scaled multiplication
(divf! F1 F2) scaled division
(gef! F1 F2) greater or equal
(lef! F1 F2) less or equal
(gtf! F1 F2) greater than
(ltf! F1 F2) less than
(minf! F1 F2) minimum
(maxf! F1 F2) maximum

2.6.3 Boolean operators

(true!) Boolean constant
(false!) Boolean constant
(and! B1 B2) conjunction
(or! B1 B2) disjunction
(not! B1) negation
(implies! B1 B2) implication
(impliedby! B1 B2) (implies! B2 B1)
(notimplies! B1 B2) (not! (implies! B1 B2))
(notimpliedby! B1 B2) (notimplies! B2 B1)

2.6.4 Array and record operators

(selecta! A I) subscripting: A[I]
(selectr! R N1) record selection: R.N1,
(storea! A I E) replace A[I] with E
(storer! R N1 E) replace R.N1 with E,
(assign! V E1 E2) used internally by system;
does not appear in J-code

2.6.5 Set operators

(empty!) set constant I
(range! I1 I2) the set of integers I
in the range I1 .. I2 I
(union! S1 S2) set union I
(diff! S1 S2) set difference I
(intersect! S1 S2) set intersection I
(subset! S1 S2) subset I
(superset! S1 S2) superset I
(in! I S) test if I is an element of S I

2.6.6 Universal operators

(equal! E1 E2) equality
(notequal! E1 E2) inequality

2.6.7 Other operators

(if! B E1 E2) conditional expression
(defined! V) test if V has a meaningful value
(new! V) instance of V after new
(defined! new! V) test if next instance of
V is meaningful
(arraytrue! A I J) array is all true from I to J I
(alltrue! A) object is entirely composed of true I
(arrayconstruct! V I J) construct array of repeats of V from I to J I
(emptyobject!) dummy object for storea and storer I

2.7 Strings

```
<string> ::= <ends in /> | <does not end in />

<does not end in /> ::= <empty> | <does not end in /> )
                      | <string> <safe character>
                      | <does not end in /> <string break>

<ends in /> ::= <string> / <string break>*

<character> ::= / | ) | <safe character>

<string break> ::= <newline> ~ /

<safe character> ::= <letter> | <digit>
                   | <metacharacter> | <other ascii character>

<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l
           | m | n | o | p | q | r | s | t | u | v | w | x
           | y | z | A | B | C | D | E | F | G | H | I | J
           | K | L | M | N | O | P | Q | R | S | T | U | V
           | W | X | Y | Z

<digit> ::= 0 | <nonzero digit>
<nonzero digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<metacharacter> ::= <greater than>
                  | <less than>
                  | <vertical bar>
                  | <asterisk>
                  | <left brace>
                  | <right brace>
                  | <number sign>
                  | <tilde>
                  | <dollar sign>
                  | <space>

<other ascii character> ::= ! | " | % | & | ' | ( | + | ,
                          | - | . | : | ; | = | ? | @ | [
                          | \ | ] | ^ | _ | `
```

Strings appearing in j-code are used to construct messages when verification fails. Wherever a string appears in a statement, it is preceded by (/ and followed by /). String breaks allow long strings to span multiple lines. A string break is a newline, followed by white space (spaces, newlines, and comments), followed by a slash. The meaning of a string is not changed by the insertion of a string break. Strings cannot contain newline characters, nor can they contain the substring /), even if the / and) are separated by a string break.

For each of the characters >, <, |, *, {, }, #, ~, $, space and newline, there is a corresponding variable <greater than>, <less than>, etc. Although no productions have been provided to do so, the each of the special characters can be derived from the metavariable bearing its name. The reader is asked to assume the metanotation has been encumbered with some escape sequences, and that these escape sequences have been used to write a production for each of the troublesome characters listed above.

2.8 Numbers

<integer> ::= <whole number> | - <counting number>
<counting number> ::= <nonzero digit> <digit>*
<whole number> ::= 0 | <counting number>

Numbers are not used directly in expressions. Instead they are used as arguments to the builtin functions consti! and constf!. Note that leading zeroes are not permitted.

2.9 Identifiers

<identifier> ::= <letter> <identifier piece>*
<identifier piece> ::= <letter>
| <digit> | <break character>
<break character> ::= . | _ | <tilde>

Note that there are no reserved identifiers.

3. The Semantics of J-code

A J-unit can be looked upon as being either a formula or a program. A J-unit is formula in that it is either "true" or "false," and there is a rigorous definition that can be used to determine which. However, the process that is used to determine the truth or falsity of a J-unit looks a lot like program execution, and the structure of the J-unit closely parallels the structure of the program from which it is generated. In this section we will describe this dual nature of J-code, and define what it means for a J-unit to be true.

3.1 Variables

Associated with every J-unit is a state, which consists of a finite set of variables. J-code variables are essentially the same as variables in programming languages; they have a value, which can be changed by NEW and ASSIGN statements.

Every variable in the state must be declared. Variables may be declared in the declaration section of a J-unit, or in the variable lists of NEW or ASSIGN statements. No variable may be declared more than once in the same J-unit, and forward references to variables to be declared a NEW or ASSIGN statement are not permitted.

When a variable is declared, it is given a type. A type declaration is an invariant about its value. Later we will discuss constraints on what constitutes valid J-code; one of thes constraints ensures that each time a variable is changed, its type invariant is preserved.

Actually, each declaration serves to declare two variables, the "real" variable and a "shadow" variable. If V is a variable, its shadow variable is accessed by (defined! V). If V is a simple variable, its shadow is of type boolean. If V is a structured variable, its shadow variable has a similar structure, except that all of its leaves are Booleans. For example, if V is an array of integers, (defined! V) is an array of Booleans.

Shadow variables are intended to be used to check for accesses to undefined variables, though the verification condition generator and theorem prover do not have a notion of "defined" J-code should be generated so that initially, all the shadow variables are false, and when assignments are made to actual variables, the corresponding shadow variables are set to true. Whenever an access is made to a variable, its shadow variable must be checked.

3.2 Statements

One way of viewing a J-unit is as a non-deterministic program, in which statements are executed, causing the state to change.

3.2.1 BREAK statement

The BREAK statement is a starting point for the execution of a J-unit. Every J-unit begins with a BREAK statement, but it the unit may contain others. Since execution of the unit is non-deterministic, any BREAK statement may be chosen as the first statement to execute. After the selected BREAK, simple statements (NEW, RENEW, ASSIGN, PROCLAIM, and REQUIRE) statements are executed in sequence, until a control flow statement is reached.

3.2.2 BRANCH and JOIN statements

The BRANCH statement is the simplest kind of control flow statement. When control reaches a BRANCH statement with a label N, the next statement to be executed is the JOIN statement whose label is N. The syntax rules for J-code ensure that there will be exactly one such JOIN statement. The rules also ensure that it is impossible to "flow into" a JOIN statement; the only way to get there is via a matching BRANCH.

3.2.3 SPLIT and WHEN statements

The SPLIT statements is the second kind of control flow statement. It is similar to the BRANCH, except that it is nondeterministic. When a SPLIT statement with label N is encountered, the next statement to be executed will be a WHEN statement satisfying the following two constraints:

a. The label on the WHEN statement is N.

b. The expression on the WHEN statement evaluates to true (given the state when the SPLIT/WHEN pair is executed).

The syntax of J-code ensures that there will be at least two WHEN statements with a label N. Another constraint (to be discussed later) ensures that at least one of these WHEN will have an expression that evaluates to true. Because execution of J-units is nondeterministic, It is completely acceptable for more than WHEN statement to qualify as the "next" statement executed after a SPLIT.

3.2.4 HANG statement

The execution of a HANG statement terminates the execution of of a J-unit. Any complete execution path begins at a BREAK, possibly passes through a series of SPLIT/JOIN and BRANCH/JOIN pairs, and terminates at a HANG. Because loops are not allowed in J-units, all of these execution paths are finite in length, and never pass through the same statement more than once.

3.2.5 NEW and ASSIGN statements

The NEW statement is a nondeterministic assignment statement. When a NEW statement is executed, all the variables in the variable list, and all their shadow variables are given new values. The values chosen are nondeterministic; any elements of the appropriate types that satisfy the expression in the NEW statement may be chosen. A constraint on J-code is that it must be possible to find appropriate values for the variables.

The operator new! can be used in the expression in a NEW statement to relate the new values given to variables by the statement to the values those variables had before the NEW. If V is a variable being changed in a NEW, the syntax (V) is used to denote the value before the NEW, and the syntax (new! V) is used to denote its value after the NEW. Likewise, (defined! V) denotes the value of the shadow variable before the NEW, and (defined! new! V) denotes the value of the shadow variable after the NEW.

The ASSIGN statement is a deterministic version of NEW. The syntax of ASSIGN is:

```
ASSIGN (V) (S) (D) (E)
```

where (V) is a list of one variable or declaration, (S) is a selector expression specifying all or part of (V), (E) is an expression with the same type as (S), and (D) is a boolean expression.

In the case where (S) is simply (V), the ASSIGN above is equivalent to the statement:

```
NEW (V) (and! (equal! (new! V) (E))
        (equal! (defined! new! V) (D))) .
```

In the case where V is a structured variable and (S) specifies a selector expression, the ASSIGN statement is equivalent to:

```
NEW (V) (and! (equal! (new! V) (V'))
        (equal! (defined! new! V) (V'')))
```

where V' is everywhere equal to V, except at the portion specified by (S); at that point V' has the value E. Likewise, V'' is everywhere equal to (defined! V), except for the shadow component corresponding to (S);

that component is equal to (D).*

3.2.6 REIN, RENEW, and REOUT statement

These three instructions are not absolutely necessary. They are only present to simplify J-code generation for loop constructs. The triple REIN−RENEW−REOUT nest in exactly the same fashion as IF−ELSE−FI in Algol 68. REIN and REOUT are not executable statements, they only serve to delimit a region of J-code. The matching RENEW is, however, executable. The statement RENEW P is equivalent to NEW (V) P, where (V) is a list of all variables that appear in the variable list in at least one NEW or ASSIGN statement that is between the REIN and the REOUT.

3.2.7 REQUIRE statement

The REQUIRE statement is what makes it possible for a J-unit to be a viewed as a formula. The REQUIRE statement is to J-units as the ASSERT statement is to Pascal-F programs. Every REQUIRE statement either "succeeds" or "fails." A REQUIRE is defined to succeed if the boolean expression evaluates to "true" when the REQUIRE is executed, no matter what nondeterministic choices lead to the execution of the REQUIRE. That is, not matter what BREAK statement was chosen to begin the execution, what values were given to variables in NEW (or ASSIGN) statements, and no matter what path was taken when SPLIT statements are executed, the boolean must evaluate to "true" for the REQUIRE to succeed. Conversely, a REQUIRE fails if there is a series of nondeterministic decisions that will lead to the execution of the REQUIRE in a state in which the Boolean expression evaluates to "false." A J-unit is said to be "correct" whenever all of its REQUIRE statements succeed.

The verification condition generator examines each of the REQUIRE statements and attempts to prove that each succeeds. Ideally, it would either prove that a REQUIRE succeeds, or produce the sequence of nondeterministic choices that results in its failure. This ideal is extremely difficult (or impossible, depending on ones views concerning the power of formal systems) to achieve for all J-units. The current system checks the REQUIRE for every possible execution path (which is possible because loops are not allowed in J-code), and reports every path for which it could not prove the REQUIRE succeeds. The strings on the BREAK and BRANCH instructions are used to indicate the path to in error messages.

It occurs to me that J-code could be changed so that WHEN statements were annotated, rather then * BRANCH statements. It makes sense to report the path in terms of BRANCH statements when the path is * being traced backwards, as the verification condition generator does it. However, people seem to prefer * looking forward, for which WHEN statements make more sense.

3.2.8 PROCLAIM statements

The statement PROCLAIM P is similar to the statement REQUIRE P. In each statement, P is expected to be true whenever control reaches the statement. The difference is that the verification condition generator checks REQUIRE statements, whereas it simply believes PROCLAIM statements, and uses them to justify subsequent REQUIRE statements. It is the responsibility of the program generating the J-code to ensure that generated PROCLAIM statements are correct.

3.3 Semantic restrictions on J-units

In the previous sections, we have alluded to some semantic restrictions on J-code. The restrictions amount to the following rule: J-units must contain REQUIRE statements that are sufficient to guarantee that correct J-units (i. e., ones where all the REQUIRE statements succeed) never "get stuck." By "getting stuck," we mean that whenever the program reaches a point where a number of nondeterministic choices are available to it, that number cannot be zero. These restrictions are not checked by the verification condition generator; it is up to the program generating the J-code to obey them.

A J-unit can only get stuck at places where nondeterministic choices can be made: in SPLIT statements and in NEW statements. Let us first consider SPLIT statements, since they are easier case. Suppose we reach a SPLIT statement that has two matching WHEN statements, with Boolean expressions B1 and B2.

---

* This description is intended to be more intuitive than rigorous. A more rigorous definition of the semantics of ASSIGN is probably in order.

Now suppose there is a way to execute the J-unit so that when we reach the SPLIT statement, neither B1 nor B2 is true. What happens?

The answer is that we are stuck, and that is not allowed in correct J-units. To correct the situation, we must precede the SPLIT statement with a statement of the form:

REQUIRE (or! B1 B2) (/split check/)

Now if we get to the SPLIT with neither B1 or B2 being true, the REQUIRE fails, so that the J-unit is not correct. If the REQUIRE succeeds, then the SPLIT never gets stuck. Either way, we do not have a correct program that gets stuck at the SPLIT, so we have met the restriction.

In most cases it is not necessary to precede a SPLIT statement by a REQUIRE. For example, the J-code generated for an IF statement involve WHEN B and WHEN (not! B). There is little point in generating:

REQUIRE (or! B (not! B)) (/system error/)

since it would always succeed.

A similar situation occurs with NEW statements. Suppose we reach the statement:

NEW (V) P

in a state in which there is no way to change V to make P true. This situation could arise because P just cannot be true in the current state, or because the required to make P true are not members of the type of V. In either case, we would again stuck. As in the SPLIT problem, we could (in theory) solve this problem by inserting:

REQUIRE ("there exists V of type T such that P") (/NEW check/)

before the NEW. However, we cannot really do this in practice, since the theorem prover cannot cope with quantified formulas. Instead we must recast the assertion to an equivalent one that does not use a quantifier, or convince ourselves that it is not necessary at all.

Fortunately, this task is not as hard as it sounds. For example, the NEW statements that come from assignment statements are of the form:

NEW (V) (equal! V E)

and the statement "there is a V such that V equals E" is always true, as long as E is an element of the type of V. If E is obviously a member of the type (for example, if it is a constant), then no REQUIRE is necessary. But if V and E are both subrange variables, where V has a smaller range than E, then a REQUIRE will be necessary.

4. Generating J-code

To use J-code to verify Pascal-F programs, it is necessary to Pascal-F into J-code. The J-code generation process is much like the one a compiler performs to generate assembly language. In this section, the J-code that should be generated for various statements of Pascal-F is discussed.

4.1 Routine calls

There are two kinds of routines in Pascal-F: functions and procedures. The J-code generated for procedure or function calls is quite similar. Before a routine may be called, certain conditions must be shown to hold. To enforce these constraints, some ASSIGN and REQUIRE statements are generated. Next, the effect of the routine call must be modeled by generating a sequence of NEW and ASSIGN statements.

If E is a routine call, we will denote the sequence of ASSIGN and REQUIRE statements by writing "SAFE E", and we will denote the sequence of NEW and ASSIGN statement by writing "SIDE E." Later we will expand this notation so that E can be an arbitrary expression, and use the notation to describe how to generate J-code for other constructs.

4.1.1 Safety checks

Two kinds of checks are required to ensure that it is safe to perform a routine call. We must check that the ENTRY assertion for the routine is satisfied, and we must check for aliasing and side effects. The first check is simply a matter of generating a REQUIRE instruction by substituting into the ENTRY assertion the actual arguments appearing in the call.

The checks for side effects and aliasing are more sophisticated. There are subtleties associated with each form of checking that make them a delicate business. The tricky part of aliasing is what is done with records and arrays. The test performed for conflicting side effects is unique in that the test must be made on a statement as a whole. For example, it is possible for two statements of the form:

```
A[1] := E1;
A[2] := E2;
```

to be free of side effect problems, but to have the statement:

```
A[1] := E1 + E2
```

be disallowed. The two tests that must be performed are described in the next two sections.

To properly understand the checks, it is necessary to have the concept of the "root" of a variable reference. The root of a simple variable is defined to be the variable itself. The root of variable of the form A[I] or A.F is defined (recursively) to be the root of A. Syntactically the root of a variable is simply the first identifier that appears in the reference. For example, the root of X[J].F[K,L].M is X.

Every variable can be decomposed into its root and a sequence of selectors. For example, the reference X[J].F[K,L].M is formed from the selector sequence {J,F,K,L,M}. Note that F and M are field names, whereas J, K, and L are subscript expressions. The scalar variable A has a null sequence of selectors.

4.1.1.1 Aliasing

Every routine call can be checked for aliasing independently of what else is going on in the statement containing the routine call. To perform the check, it is necessary to examine every pair of VAR arguments to the (augmented) call. For each pair, if the arguments are both readonly or have different roots then they need not be considered further. If the pair have the same root and at least one is readwrite, a REQUIRE instruction may need to be generated.

Determining whether a REQUIRE needs to be generated, and if so, what the REQUIRE is, proceeds as follows. Get the selector sequence for each of the the two variables and truncate the longer sequence 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 in these two truncated sequences. If any two corresponding field selectors differ, no REQUIRE needs to 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) cannot be passed as a VAR argument, not even on a readonly basis.

4.1.1.2 Side Effects

Side effect checking is performed on a statement-by-statement basis. To perform the check, it is necessary to collect

a. all the routine calls made in the statement, and

b. all the variables that are otherwise modified in the statement.

Performing part (a) is straightforward. Note that if the statement is a procedure call, the statement itself

counts as one of the routines called in the statement. The list of variables obtained in part (b) depends of the statement type. In the case of an assignment statement, the list consists of the root of the variable to the left of the assignment operator. In the case of a FOR statement, the list consists of the root of the index variable. For all other statements the list is empty.

Next, a "modification list" is formed for every routine call collected in (a). This list consists of all the readwrite VAR arguments in the (augmented) call. The list collected in (b) also constitutes a modification list. A statement passes the side effects test if (1) every variable in the statement is in at most one of these modification lists, and (2) every variable in the modification list of a routine call appears (in that statement) only in the routine call.

4.1.1.3 ENTRY conditions

Most of the safety checks we have discussed so far have been syntactic in nature. The only REQUIRE statements generated are used to prove that subscripts are distinct. If the call being processed is recursive, a REQUIRE must be generated to ensure that the recursion will eventually terminate; see the section on routine bodies for more details. We next describe the other J-code statements that must be generated whenever "SAFE E" appears in a template.

When generating J-code for routine calls, it is necessary to have an unending supply of auxiliary variables that do not duplicate the names of program variables. Within the safety code, two sequences of these variables are declared and given values with ASSIGN statements. Every input argument in the augmented call is assigned to an auxiliary variable, using an ASSIGN statement. We will denote these variables R1, R2, ... Rm. These variables are used to simplify the processing required on the ENTRY conditions, and to save the "old" values of the parameters for later J-code statements.

Next, the root of every output argument is assigned to an auxiliary variable using an ASSIGN statement. Let us denote this new set of variables S1, S2, ... Sn. These variables will be referenced in the effects section of the J-code for the routine call.

Finally, a statement of the form REQUIRE P is generated, where P is obtained from the EXIT condition by substituting for every formal parameter and global variable the corresponding Ri.

4.1.2 Modeling effects

The next section of J-code that is generated for a routine call is the effects section. The effects section is a NEW instruction and a series of ASSIGN instructions, which reference the auxiliary variables declared and set in the safety section. These instructions update the program variables being modified by the routine call.

The first statement in the effects section is a NEW statement that declares and sets an auxiliary variable for each output parameter of the routine being called. Let us denote these variables as T1, T2, ... To. The assertion in the NEW instruction is generated by performing substitutions on the EXIT assertion of the routine. Every (augmented) parameter in the call R appearing in the EXIT assertion must be replaced with a variation on the corresponding actual parameter, as described below. Recall that R1, R2, ... Rm are values of the input parameters before the routine call, S1, S2, ... Sn are the roots of the output parameters, and T1, T2, ... To are the new values being returned by the routine.

Formal param. in EXIT	What is substituted in
Value-parameter.OLD	The corresponding Ri
VAR-parameter.OLD	The corresponding Ri
Output VAR parameter	The corresponding Ti
Readonly VAR parameter	The corresponding Ri
Function name	The actual function call, complete with arguments

Finally, a sequence of ASSIGN statements is is used to set each actual parameter to the corresponding Ti. Let A1, A2, ... Ao be the actual parameters. Recall that the actual parameters may be components of larger variables. Therefore, if Ai is an actual parameter, let RAi be the root of Ai, and let SAi be Ai with the

following substitutions applied: for every variable V that is modified by the routine call and appears in a subscript expression in Ai, replace V by the corresponding Sj, the auxiliary variable that was used to save the value of V in the safety section. The i'th ASSIGN statement is then:

ASSIGN (RAi) (Ai) (true!) (SAi) .

4.1.3 Multiple routine calls

Let us summarize the J-code that is generated for each routine call. The J-code has two parts: the safety section and the effects section. The following takes place in the safety section:

a. REQUIRE statements necessary to rule out aliasing of array elements are generated.

b. REQUIRE statements necessary to rule out infinite recursion are generated.

c. The variables R1, R2, ... Rm are used to save the values of the input arguments.

d. The variables S1, S2, ... Sm are used to save the roots of the output arguments.

e. REQUIRE instructions are generated by substituting R1, R2, ... Rm into each of the the ENTRY assertions.

In the effects section, the following takes place:

a. A NEW statement is generated using the variables T1, T2, ... To. The assertion is obtained by substituting R1, R2, ... Rm and T1, T2, ... To into the ENTRY conditions of the routine.

b. A series of ASSIGN statements is used to update each of the actual parameters of the call to the appropriate Ti. S1, S2, ... Sm are substituted into subscript expressions in the actual parameters.

If E is a single routine call, we abbreviate the safety section as SAFE E, and the effect section as SIDE E. However, if a statement contains several function calls, and we must pay careful attention to the order in which the J-code instructions are generated.

Within expressions in J-code generated for a Pascal-F statement, an unembellished variable reference denotes the value of the variable before the statement began, and a reference of the form (new! v) refers to the value of the variable v after the statement finishes. It is important that no variable ever appears in the variable lists of two NEW instructions generated for any one statement. The purpose of the aliasing restrictions is to ensure us that this restriction will be observed.

Let us consider other ways two NEW instructions generated for the same Pascal-F statement could interfere with one another. If we have the two instructions:

```
NEW (V1) P1
...
NEW (V2) P2
```

The first NEW instruction has the effect of altering the meaning of any variables that are in both V1 and P2. It is therefore important that V1 and P2 do not have any variables in common, since the variables in P2 must denote their values before the statement began execution.

The way we avoid this kind of interference is to make sure that all the safety sections precede all the effect sections that are generated in any one statement. The assertions in the NEW instructions never mention actual program variables by name, only auxiliary variables that are set in the safety section. For this to work, it is necessary that whenever a variable V is used in a statement, the value accessed is the value V had before the statement began execution, and not some new value obtained through a side effect. This condition will always hold, by our restrictions on the use of side effects.

Thus, if E is an arbitrary expression, when we write SAFE E, we mean the concatenation of all the safety sections for all the routine calls in E. When write SIDE E, we mean the concatenation of all the effect sections for all the routine calls in E. The restrictions on aliasing and side effects ensure that the order of the sections within SAFE E and SIDE E is immaterial.

4.1.4 Variant records

Another feature of the language requiring special consideration is variant records. The verifier 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. The tag field is given a value, but all other fields controlled by the tag are made undefined. A variable v is made undefined with the J-code statement

NEW (v) (true!)

The checking performed by the SAFE operation must ensure that a particular variant field will be accessed only when the associated tag field has the correct value.

4.2 Assignment statements

Assignment statements can be considered to be a special case of procedure calls. Assume the existence of a generic procedure of the form:

```
PROCEDURE ASSIGN(VAR V: type1; E: type2);
ENTRY {value of E appropriate for type of V};
EXIT V = E, DEFINED(V);
...
```

Then treat the statement A := E as the call ASSIGN(A, E). Note that since ASSIGN has only one VAR parameter and no global variables, it cannot have aliasing problems. A statement such as A := F(A), where F modifies its argument, is considered to have an erroneous side effect.

4.3 J-code templates

The J-code generated for control structures is presented using templates. Some of the notation used in the templates requires explanation. The dotted brackets [. and .] indicate that the enclosed expression must be translated to J-code notation. For example, [.NOT (x > 0).] stands for

(not! (gti! (x) (consti! 0))).

CODE <construct> is notation meaning the J-code recursively generated for <construct>.

We have seen before that the J-code generated for a routine call has two parts: the safety part and the effects part. If E is an expression to be evaluated, then J-code will have to be generated for every function call in E. We will denote by "SAFE E" all the concatenation of the safety sections for all the function calls in E. We will denote by "SIDE E" the concatenation of all the effect sections for the function calls in E.

4.4 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.]
SPLIT    1
WHEN     [.B.] 1
SIDE     [.B.]
CODE     <then part>
BRANCH   2 (/then branch/)
WHEN     [.NOT B.] 1
SIDE     [.B.]
CODE     <else part>
BRANCH   2 (/else branch/)
JOIN     2
```

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 and the "else branch" string is changed to "skip then branch."

4.5 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:

```
SAFE     [.x.]
REQUIRE  [.x=c1 OR x=c2 OR ... x=cn.]
SPLIT    1
WHEN     [.x=c1.]  1
SIDE     [.x.]
CODE     <statement 1>
BRANCH   (/CASE c1/) 2
WHEN     [.x=c2.] 1
SIDE     [.x.]
CODE     <statement 2>
BRANCH   (/CASE c2/) 2
...
WHEN     [.x=cn.]
SIDE     [.x.]
CODE     <statement n>
BRANCH   (/CASE cn/) 2
JOIN     2
```

The ISO 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.

4.6 The LOOP statement

While loops and until loops can each be written in terms of LOOP statements. LOOP statements have the following syntax:

LOOP

<pre-body>

STATE P
MEASURE M

<post-body>

ENDLOOP

The semantics executing ENDLOOP is to transfer control to the matching LOOP. The STATE and MEASURE statements must be at the "top" nesting level of the loop; they cannot be nested within other compound statements. This restriction is enforced to ensure that all paths from LOOP to ENDLOOP pass through the STATE and MEASURE.

Within the loop there may be statements of the form:

EXIT IF B

and

EXIT IF B THEN S.

Execution of these statements causes B to be evaluated, and if it is true, control is transferred to the ENDLOOP. In the second form, the statement S is executed before the loop is exited. There is no reason to make restrictions on the nesting level of EXIT statements, except for those needed to properly match the EXIT with the proper ENDLOOP.

The J-code for IF and CASE statements exactly mirrored the structure of the Pascal-F code. The J-code for loops is not as simple, since J-code is not allowed to have loops. As soon as a loop is entered, the J-code branches to the MEASURE statement, where a RENEW statement updates all the loop variables. If the loop executes for N iterations, this RENEW can be thought of as an abstraction of the first N-1 iterations. A HANG statement is inserted just before the RENEW to prevent loops in the J-code.

The following J-code is generated for a loop with n EXIT statements, each of the form.

EXIT IF Bi THEN Si (for i=0, 1, ... n-1).

Two temporary variables must be declared as loop counters, which are used to prove that the loop terminates.

```
ASSIGN    (temp1: (integer)) (temp1) (true!) (temp1)
ASSIGN    (temp2: (boolean)) (temp1) (true!) (true!)
REIN
SPLIT                                                    0
WHEN TRUE 0
BRANCH 1  (/advance to loop state/)
WHEN TRUE 0
BRANCH 2  (/first iteration of loop/)
JOIN      2

CODE      <pre-body>

REQUIRE   P  (/loop state/)
REQUIRE   [. M >= 0 .] (/measure non-negative/)
REQUIRE   [. (M < temp1) or temp2 .] (/measure decreased by loop body/)
HANG
JOIN      1
RENEW     P
ASSIGN    (temp1) (temp1) (true!) M
ASSIGN    (temp2) (temp2) (true!) (false!)

CODE      <post-body>

BRANCH    2 (/loop back/)
REOUT

WHEN      B0 4
SIDE      B0
CODE      S0
BRANCH    3 (/loop exit/)

WHEN      B1 5
SIDE      B1
CODE      S1
BRANCH    3 (/loop exit/)
...

WHEN      Bn-1 4+n-1
SIDE      Bn-1
CODE      Sn-1
BRANCH    3 (/loop exit/)

JOIN      3
```

If the i'th exit statement does not have a then part, then the CODE Si is simply omitted from the above template. For each statement of the form:

```
EXIT IF Bi [THEN Si]
```

The following J-code is generated:

```
SAFE B0
SPLIT 4+i
WHEN [. not B0 .] 4+i
SIDE B0
```

The generated J-code is the same whether or not the THEN part is present.

4.7 FOR loops

Pascal-F FOR loops have the following syntax:

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

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

One approach to FOR loops would be to compile them into LOOP statements and generate code for the LOOP statement. Although this would work, it would require the user to write more complicated loop invariants. Instead, the following J-code should be generated. The J-code generated for both loops is the same except for four instructions. For these four instructions, the version for downward loops is shown in angle brackets on the next line.

```
SAFE      lo
SAFE      hi
ASSIGN    (lot: (integer)) (lot) (true!) (lo)
ASSIGN    (hit: (integer)) (hit) (true!) (hi)
SIDE      lo
SIDE      hi
ASSIGN    (i) (i) (true!) (lot)
<ASSIGN   (i) (i) (true!) (hit)>
SPLIT     1
WHEN      [.lot <=  hit.] 1
BRANCH    (/nnn: ENTER/) 2
JOIN      2
REIN
CODE      <prebody>
REQUIRE   [.P.]
HANG
WHEN      [.TRUE.] 1
RENEW     [.P and (lot <= i) and (i <= hit) and defined(i).]
CODE      <postbody>
SPLIT     3
WHEN      [.i < hit.]  3
<WHEN     [.i > lot.]  3>
ASSIGN    (i) (i) (true!) [.i+1.]
<ASSIGN   (i) (i) (true!) [.i-1.]>
REOUT
BRANCH    (/nnn: LOOP/) 2
WHEN      [.i = hit.] 3
<WHEN     [.i = lot.] 3>
BRANCH    (/nnn: EXIT/) 4
WHEN      [.lo > hi.] 1
BRANCH    (/nnn: SKIP/) 4
JOIN      4
NEW       (i) [.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.

4.8 ASSERT statements

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

ASSERT(P)

is:

REQUIRE P

4.9 The SUMMARY statement

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

```
REQUIRE   P
BREAK
PROCLAIM  P
```

4.10 Routine definitions

Associated with each routine, are ENTRY and EXIT assertions. If the routine is recursive, a DEPTH expression is also associated with the routine. After call augmentation has been performed, the only variables contained in these assertions and the expression ought to be parameters to the routine. It is not strictly speaking an error for the ENTRY assertion of a routine to contain a variable that is not part of its augmented parameter list, but such an occurrence should probably flagged anyway, since it is bad form.

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

```
NEW (parameters and local variables)
  [. <entry condition> and
    <definedness of value parameters> and
    (new(d0) = <depth expression>) and
    (new(v1) = new(p1)) and
    (new(v2) = new(p2)) and
    ...
    (new(vn) = new(pn)) .]

REQUIRE    [. d0 >= 0 .]  (/depth non-negative/)
CODE       <routine body>
REQUIRE    <exit condition>
REQUIRE    (defined! f)   (/function name is defined/)
```

The last REQUIRE only appears in functions. Its role is to make sure the function is defined, which in turn ensures that it truly is a function. Any assumptions about the definedness of VAR parameters must appear explicitly in the entry assertion.

The variables v1, v2, ... vn are used to interpret the pseudofield OLD. Whenever pi.OLD 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.