Pascal-F Verifier Internal Design Document

CPCI #6 -- Rule Builder

John Nagle

FACC / Palo Alto

1. Introduction

The Rule Builder is the part of the Verifier which is used to create rules. Rules are expressions which are known to be true in a general sense, and provide information to the Verifier about functions for which the simplifier does not have a built-in decision procedure. The job of the Rule Builder is to allow a trained user to add to the knowledge base of the verifier in a safe way. All rules must be proven true to the satisfaction of the Rule Builder before they are added to the database of rules used by the simplifier. The Rule Builder incorporates a powerful and elaborate theorem prover to assist the user in this difficult task.

2. The Interface between the Rule Builder and the Verifier

The Rule Builder is a separate program from the remainder of the Verifier. It is a large Interlisp program, the Boyer-Moore Theorem Prover, augmented by some special routines for manipulating the Verifier's rule databases. The Rule Builder is never invoked automatically.

Each program being verified has a working directory whose name is of the form "<programname>.d". One file in this directory is the rule database being used to verify that program. This file is read by the Verifier's simplifier, and is both read and written by the Rule Builder. The Rule Builder is invoked manually by users to interactively prove new theorems and add rules to the database.

2.1 Rules

Rules are expressions involving functions. Rules are expressed in S-expression notation. A number of functions are built into the Verifier and the Rule Builder, and are always available to the user for constructing rules through the Rule Builder.

2.1.1 Built-In Primitive Functions

2.1.1.1 Type Predicates

(INTEGERP x)	True if ‘‘x’’, which may be an expression or a variable, is of a numeric integer type. This includes enumeration types, but does not include fixed-point numbers.
(NUMBERP x)	True if ‘‘x’’, which may be an expression or a variable, is of a non-negative integer type.
(BOOLEANP x)	True if ‘‘x’’, which may be an expression or a variable, is Boolean-valued.
(ARRAYP x)	True if ‘‘x’’, which may be a variable or a selector-valued expression, is an array.

2.1.1.2 Arithmetic Operators

(PLUS x y)	Integer addition, without bounds. (NUMBERP x) must be true for both operands.
(MINUS x y)	Integer subtraction, without bounds. (NUMBERP x) must be true for both operands.

(TIMES x y)	Integer multiplication, without bounds. (NUMBERP x) must be true for both operands.
(DIVIDE x y)	Integer division, without bounds. (NUMBERP x) must be true for both operands, and (NOT (EQUAL y 0)) must be true.
(MOD x y)	Integer remainder, without bounds. (NUMBERP x) must be true for both operands, and (GREATERP y 0) must be true.

2.1.2 Comparison operators

(To be supplied)

2.1.3 Selector-valued operators

(SELECTA a i)	Array element extraction. Requires (ARRAYP a) and (NUMBERP i).
(STOREA a i j)	Array element replacement. Requires (ARRAYP a) and (NUMBERP i). Result is an array equal to ‘‘a’’ except that ‘‘a[i]’’ is replaced by ‘‘j’’.
(SELECTR a f)	(to be supplied)
(STORER a f)	(to be supplied)

2.2 User-defined functions

All of the above functions have definitions known to the Rule Builder. The user may add new functions in the Rule Builder through the use of the DEFINE command of the Boyer-Moore theorem prover. New functions must be made known to the Verifier by defining them as EXTRA functions in the source program. Such functions must not have ENTRY, EXIT, or EFFECT clauses, nor may they have any statements in the body or local variables. Functions of this type are called ‘‘definition functions’’ and may appear only in assertions in the Pascal-F source.

2.3 Pattern Matching in Rules

The application of rules is in a stylized, predicatable way. This limited approach has the advantage that the rule handler will behave in an understandable manner. Rules often will need to be created to handle specific situations; in many causes, rules will be usable only at one place in a given program. Rules are only a mechanism for making information derived in the Rule Builder available to the simplifier, not an attempt to make the simplifier smarter.

Rules are always of the form ‘‘A implies B’’. The B term cannot contain the operators AND or OR. Rules are applied only to expressions containing the implication operator. Rules are applied by generating a subgoal; i.e. if ‘‘A implies B’’ is applied to ‘‘X implies Y’’, a subgoal of ‘‘X implies B’’ is generated. A rule of the form ‘‘A implies B’’ will be applied to the expression ‘‘X implies Y’’ if and only if B matches Y. All rules which match must be applied in all possible ways of match, unless it can be determined that applying the rule is unnecessary or futile.

Matching is semantic in nature. More material on matching will be supplied.

3. An Example

Let us suppose that we want to prove that all the elements in an array are zero after a loop has cleared the array. Consider the following program fragment.

```
VAR tab: array[1..10] of integer;
   i,n: 1..10;
...
FOR i := 1 TO 10 DO BEGIN
   tab[i] := 0;
   END;
ASSERT(a[n] = 0);
```

The proof of the assertion will require proving that the loop is exited with all elements of the array equal to 0. There is no way to describe the concept of all the elements of an array being zero with the primitives of the Pascal-F language alone, so we will introduce a function.

```
{
        allzero  -- part of array is zero predicate

        True if a is zero from lo to hi.
}
EXTRA FUNCTION allzero(a: tab; lo: integer; hi: integer);
        BEGIN END;
```

We are now able to state our loop invariant. We can also state that the entire array is zero after the clearing loop.

```
VAR tab: array[1..10] of integer;
   i,n: 1..10;
...
FOR i := 1 TO 10 DO BEGIN
   tab[i] := 0;
   INVARIANT (allzero(tab,1,i));
   END;
ASSERT(allzero(a,1,10));  { a[n] now zero from 1 to 10 }
ASSERT(a[n] = 0);
```

This completes our preparation of the source program. It is now necessary to work with the Rule Builder to build the necessary rules needed to deal with this program.

We begin by defining ‘‘allzero’’ informally. Note that the definition is recursive.

```
allzero(a,i,j) =
if not ARRAYP(a) then false
else if not NUMBERP(i) then false
else if not NUMBERP(j) then false
else if i > j then true
else if a[j] <> 0 then false
else allzero(a,i,j-1)
```

Note the type restrictions. All functions in the Rule Builder must generate some value for all inputs. We return false for invalid inputs.

We are now ready to define the function for the Boyer-Moore theorem prover in its language.

```
DEFN(ALLZERO
   (A I J)
   (IF (AND (ARRAYP A)
            (NUMBERP I)
            (NUMBERP J))
        (IF (LESSP J I)
          T
          (IF (NOT (EQUAL (SELECTA A J) 0))
               F
               (IF (ZEROP J)
                 T
                 (ALLZERO A I (SUB1 J)))))
        F))
```

We now have to prove a few things about ‘‘allzero’’, which we will use later as rules. Proof of the loop invariant, above, will require that we prove that if an array is zero up to i-1, and it is zero at i, then it is zero up to i. In the Boyer-Moore notation, this reads

```
PROVE.LEMMA(ALLZERO.EXTEND
          (REWRITE)
          (IMPLIES (AND (ALLZERO A I (SUB1 J))
                        (EQUAL (SELECTA A J) 0))
                  (ALLZERO A I J)))
```

We will call this Rule #1. If, after proving this with the Boyer-Moore prover and adding this as a rule to the database for the program being proven, we then attempt reverification of the program, we will be able to prove the loop invariant for the path around the loop. We should even be able to prove the assertion after the loop. The case at entry to the loop requires another rule, a trivial one, which we will call rule #2.

```
PROVE.LEMMA(ALLZERO.VOID
          (REWRITE)
          (IMPLIES (AND (NUMBERP I)
                        (NUMBERP J)
                        (ARRAYP A)
                        (LESSP J I))
                  (ALLZERO A I J)))
```

In this case, the case for which the low bound for ‘‘allzero’’ exceeds the high one, the truth of the proposition follows directly from the definition. However, the simplifier needs to know this explicitly.

Finally, we will need a third rule about ‘‘allzero’’ later in the program.

```
PROVE.LEMMA(ALLZERO.SELECT
          (REWRITE)
          (IMPLIES (AND (NUMBERP I)
                        (NUMBERP J)
                        (NUMBERP X)
                        (ARRAYP A)
                        (NOT (LESSP J X))
                        (NOT (LESSP X I))
                        (ALLZERO A I J))
                  (EQUAL (SELECTA A X) 0)))
```

When we want to prove that a[n] = 0, this rule #3 will be used by the simplifier.

The definition of ALLZERO given above, along with the three lemmas, have been proven to be sound by the Boyer-Moore prover. No additional lemmas about ALLZERO were needed to prove the lemmas given above, and all the proofs were short. This is an encouraging result, in that it indicates that proving rules similar to these will not be unduly demanding on the user.