Pascal-F Verifier Internal Design Document

CPCI #2 -- Preverification Checker

John Nagle

FACC / Palo Alto

1. Introduction

CPCI #2 has two primary functions; the translation of Icode to Jcode, and the performance of all static checks on program structure not performed by the compiler phase, CPCI #1.

2. Information Passed Between Pass 1 and Pass 2

Three files are passed from CPCI 1 to CPCI 2. These are the Icode file, which represents the program executable statements, the routine interface database, which describes the interfaces, including assertions, of all procedures and functions, and the dictionary file, which contains the definitions of all data objects.

2.1 Icode

The Icode notation is borrowed from the Pascal-F compiler. Icode is a representation of the program in reverse polish notation. The notation is quite low-level, and it is in fact possible to implement an interpreter for Icode programs. The Icode used by the verifier is an extended form of that used by the compiler.

The detailed definition of Icode is found in the Icode Interface Control Document. A brief summary of the additions to the compiler's form of Icode appears below.

ASSERT, STATE, SUMMARY, and MEASURE statements are treated as procedure calls. The ENTRY, EXIT and EFFECT clauses in procedure headers are treated as operators with one boolean argument and no result and are found only at the beginning of procedure definitions. PROOF is an operator which takes statements, and returns nothing.

2.2 Routine Interface Database

The routine interface database ultimately contains all the information required when verifying a call to a routine. This includes the definition of the calling sequence, the ENTRY and EXIT assertions, and the global variables referenced by the routine.

All procedures are entered into the routine interface database by CPCI #1.

2.3 Dictionary

This is the compiler's dictionary. The form of this file is not yet defined.

3. Information Passed From Pass 2 to Pass 3

3.1 Jcode

The primary output of CPCI 2 is a representation of the program in ‘‘Jcode’’. Jcode is defined in the ‘‘Jcode Interface Control Document’’.

3.2 Dictionary

See above.

3.3 Routine Interface Database

The form of this database has not yet been defined.

4. Jcode Generation

Jcode generation is a combination of decompilation of Icode expressions, generation of ‘‘object code’’ in the form of Jcode, and detection of violations of Pascal-F rules.

4.1 Variable Handling

In the Icode representation, variables are referred to by triples of the form

    <level>, <size>, <address>

which is too machine-oriented for verification purposes. Addresses are converted back into variable names by dictionary lookup. This requries a dictionary organized by address. of this process... The dictionary file contains sufficient information to allow conversion of addresses back into variable names and subscripts where required.

5. Internal design

5.1 Expression decompiler

```
-------          ------------
|   |-------------| OFFSET   |     (record references)
| E |            ------------
| X |
| P |            ------------
| R |-------------| INDEX    |     (array  references)
| E |            ------------
| S |
| S |            ------------
| I |-------------| FIELD    |     (packed item references)
| O |            ------------
| N |
|   |            ------------   -------------
| P |-------------| VARBL    |-----|VAR lookup |____
| R |            ------------   -------------  |  |----------|
| O |                                          |---|Dictionary|
| C |            ------------   -------------  |  |----------|
| E |-------------| call     |-----|Function   |___|
| S |            |        |   |Lookup     |
| S |            ------------   -------------
| O |
| R |            ------------   -------------
|   |-------------| operators|-----|Op tables  |
|   |            ------------   -------------
-------
   |
-------
|Typed|
|Stack|
-------
```

Figure 1. Decompiler for Icode expressions

The expression decompiler translates expressions found within Icode. Expressions are present in Icode whenever the BNF for Pascal-F calls for an ‘‘<expression>’’. Various Icode statement operators take one or more expressions as arguments. When statement processing reaches a point where an expression is expected, the expression decompiler is called. The expression decompiler takes in Icode, starting at the current position in the Icode string, and returns a Jcode expression (see Jcode interface control document).

Although the names and types of variables appear to have been lost by the time Icode is processed, the decompiler is able to recover them. The various operators associated with names are processed as follows.

VARBL	The VARBL operator indicates a reference to a named variable. Using the address and class of the variable, the name and type of the variable are retrieved from the dictionary. The name and type are then pushed on the typed stack used by the expression decompiler. An internal error is detected if an address is not the starting address of a variable.
OFFSET	The OFFSET operator indicates a reference to a named subfield of a record. The typed stack is popped, and the name and type examined. The type must be a record type, and the offset, which must be a constant, must correspond to a field in the record definition. Variant records are handled by requiring that CPCI #1 generate unique offsets for the fields in each variant when running as part of the Verifier, allowing OFFSET processing to uniquely identify the record components. The result of OFFSET is the type of the component and the input expression with the ??? Jcode operator and the field name added.
INDEX	The INDEX operator indicates a reference to an array. The typed stack is popped twice, this being a binary operator. The two operands are the variable being indexed, which of course must be an array, and the index of the array. The result of INDEX is the input expression with the SUBSCRIPT operator and the name of the array applied; this result and the type of the array component are pushed on the stack.
FIELD	The FIELD operator indicates a reference to a field of a packed structure. In theory, the structure can be an array or a record. Inputs are the field size in bits and the displacement in bits from the first bit of the item. The FIELD operator is handled as INDEX or OFFSET as appropriate, and a suitable variable-valued expression is returned.

5.1.1 Internal representation of Jcode expressions

In Jcode, expressions are represented as S-expressions. For example, the expression

A + (C.D * B)

is represented in Jcode by

(addi A (multi (select C D) B))

and is passed to CPCI #3 as a character string of this form. The internal representation within CPCI #2 is the usual tree. We refer to the directions of edges as ‘‘right’’ and ‘‘down’’, rather than ‘‘left’’ and ‘‘right’’, purely so that we can draw more linear pictures, such as

```
-------
| addi|
-------
   |
-------   -------
|  A  |------|multi|
-------   -------
                |
             --------   -------
             |select|-----|  B  |
             --------   -------
                |
             -------   -------
             |  C  |------|  D  |
             -------   -------
```

where the argument list always starts below the relevant operator.

5.1.2 Decompiling of operators

As an example of the technique used for decompiling expressions, the sequence for decompiling a binary operator is given. This is not necessarily the code of the implementation.

```
{
        binaryoperator --  decompile a binary operator
}
procedure binaryoperator(op: opcode);   { operator being decompiled }
var
  expr1, expr2: ^jitem;              { pointers to Jcode expressions }
  type1, type2, newtype: ^titem;      { pointers to type expressions }
  class1, class2: valueclass;         { variablekind or valuekind }
begin
  pop(expr1,type1,class1);            { get info about 1st operand }
  pop(expr2,type2,class2);            { ditto for 2nd operand }
  binoptypecheck(op,type1,type2,newtype); { get result type }
  require(arg1req[op],type1,expr1);   { generate REQUIRE for domain
                                        enforcement for 1st operand }
  require(arg2req[op],type2,expr2);   { ditto for operand two }
                                      { construct and push Jcode
                                        representation of result }
  push(newjcode2(jcodeop[op],expr1,expr2),newtype,valuekind);
end {binary operator};
```