User Defined Operators for occam2.x

James Moores (jm40@ukc.ac.uk)

Computing Laboratory, University of Kent at Canterbury, CT2 7NF

Abstract. This document is intended as a supplement to the occam2.1 extension documentation available from the IPCA (The Internet Parallel Computing Archive[1]). It describes the first in a series of new additions to the SGS-Thompson occam compiler. These extensions are only available as part of the KRoC compiler package, and not for the original toolset compiler.

1 Introduction

This paper documents the extension to the occam2.1 multi-processing language made in the KRoC 0.9beta release[2].

This extension provides user-defined operators over any occam2.1 primitive or user-defined types. It allows further overloading of the existing set of operators:

binary	*unary*
`+  -  *  /  \`	`-`
`PLUS  MINUS`	`MINUS`
`TIMES  AFTER`	
`/\  \/  ><`	`~`
`AND  OR`	`NOT`
`=  <>  <=  >=`	

Figure 1: Existing occam2.1 operators

and introduces some new ones:

binary or *unary*
`??  @@  $$  %  %%  &&  <%  %>  <&  &>`
`<]  [>  <@  @>  @  ++  !!  ==  ^`

Figure 2: New occam2.x operators

[1] http://www.hensa.ac.uk/parallel/occam/documentation/

[2] http://www.hensa.ac.uk/parallel/occam/projects/occam-for-all/kroc/

2 Syntax and semantics

The syntax for declaring a new operator is identical to a FUNCTION except that:

- instead of a user-defined name, there is a user-defined string containing the sequences of characters making up the desired operator. Note that this sequence must be one of those listed in Figures 1 or 2 – we are not allowed to make up our own symbols. Note also that since * is an escape character in occam strings, that operator has to be quoted as "**".
- there must be either two arguments (for a *binary* operator) or one argument (for a *unary* operator). Note that, as for FUNCTIONs, these arguments must be VAL data types. Henceforth, we shall use the term *operands* for these arguments.
- there may only be one return type (not a list) from these operators.

All the semantic rules for FUNCTIONs are inherited for operators. So, operator bodies may cause no side-effects, either through modifying global data-structures or through communication. This is checked by the compiler.

Operators may be INLINEd. Their bodies may be *one-liners*, using the IS syntax for FUNCTIONs, or they may be VALOF expressions.

Operators may be defined for any occam2.1 types, including array types. The operand types (for *binary* operators) do not have to be the same. Formal operand arrays may be *open* (i.e. unsized) or closed (i.e. sized). However, because of the normal rules for occam FUNCTIONs, any array result type must be sized.

3 A simple example

Suppose we define:

```
DATA TYPE COMPLEX64 IS
  RECORD
    REAL64 real, imag:
:

COMPLEX64 INLINE FUNCTION "+" (VAL COMPLEX64 a, b) IS
  [a[real] + b[real], a[imag] + b[imag]]:
```

We may then start using them like:

```
COMPLEX64 a, b, c:
SEQ
  a := [42.0, 99.7]
  b := [-123.456, 78.0]
  c := a + b
  ...  etc
```

4 Operator resolution and overriding

When resolving which operator function is intended for any particular use of an operator symbol, only the operand types are used. We considered making the result type significant, but that leads very quickly to nested expressions whose operators cannot be resolved (because they have equally valid alternative interpretations).

So, operators may be uniquely overloaded up to the types of their operands. However, this means that if we define the following two operators:

```
THING FUNCTION "+" (VAL FOO a, VAL BAR b) IS ...

THONG FUNCTION "+" (VAL FOO a, VAL BAR b) IS ...
```

the later definition will *override* the first one. Thus, if `x` is of type `FOO` and `y` is of type `BAR`, the expression `x + y` will be evaluated using the later definition and will yield a value of type `THONG`.

Note also that, because the compiler generates operator names made up from the operator symbol and its operand types, having the above two definitions compiled into some library (or even separate libraries) will cause a UNIX linking error.

The current compiler is also blind to the *size* (or *un-size*) of operand arrays. Thus, the following two definitions will suffer the same fate as the above:

```
THING FUNCTION "[>" (VAL [16]INT a, VAL [4]INT b) IS ...

THONG FUNCTION "[>" (VAL []INT a, b) IS ...
```

However, note that *unary* operators cannot get overridden by *binary* operators. So that the following two operators can be declared in any order and co-exist:

```
COMPLEX64 INLINE FUNCTION "-" (VAL COMPLEX64 a) IS
  [-a[real], -a[imag]]:

COMPLEX64 INLINE FUNCTION "-" (VAL COMPLEX64 a, b) IS
  [a[real] - b[real], a[imag] - b[imag]]:
```

Generally, we deprecate the overriding of previously defined operators. For example, a really twisted user may declare:

```
INT INLINE FUNCTION "+" (VAL INT a, b) IS a - b:
```

with somewhat opaque consequences for the remaining code!

5 Unary and binary operators

The standard operators may only be overloaded provided their *binary-ness* or *unary-ness* is maintained. Thus, `+` may only be used as a *binary* operator, `NOT` may only be used as a *unary* operator, but `-` may only be used as both.

All the new operator symbols may be used for both *binary* and *unary* operators – for example:

```
INT FUNCTION "$$" (VAL []INT a)  -- sum the array
  INT result:
  VALOF
    SEQ
      result := 0
      SEQ i = 0 FOR SIZE a
        result := result + a[i]
    RESULT result
:
```

And, then:

```
[42]INT X, Y:
[99]INT Z:
SEQ
  ...  set up arrays X, Y, Z
  out.string ("The sum of X, Y and Z is ", 0, screen)
  out.number (($$ X) + (($$ Y) + ($$ Z)), 0, screen)
  out.string ("*n", 0, screen)
```

6 Operator inheritance

occam2.1 has a feature of inheriting the use of the standard operators on user-defined types that are defined directly in terms of the basic types. For example, if we had:

```
DATA TYPE BLUE.INT IS INT:
```

then we could have:

```
BLUE.INT a, b, c:
SEQ
  ...  set up a and b
  c := a + b
```

This was presumably done to allow operator usage over these types. Whether this inheritance should be retained now that user-defined operators are available for such types is debatable. [Note that `INLINE`d operators defining the standard arithmetic operators for `BLUE.INT`s (via casts into `INT`s) generates exactly the same code as direct use of the standard operators – there is no run-time overhead.] Anyway, this operator inheritance is preserved in the current KRoC 0.9beta release.

However, there leads to an inconsistency. If we were to define:

```
DATA TYPE BLUE.COMPLEX64 IS COMPLEX64:
```

where COMPLEX64 is as defined earlier, BLUE.COMPLEX64 does *not* inherit any user-defined operators for COMPLEX64. Feedback on which way this incosistency should be resolved is welcome.

The inheritance behaviour does raise a problem for types that are defined directly from the basic ones. For instance, in the 0.9beta release, we provide an example string library that manages dynamically allocated strings in the C-world. The occam type just holds a pointer to space in that C-world and the obvious type declaration is:

```
DATA TYPE DSTRING IS INT:  -- dynamic string
```

We can then define an ++ operator for concatenation and so on. However, DSTRING automatically inherits all the standard arithmetic operaors on INTs which will enable arithmetic on pointers – *not* a good thing! To avoid this, we define DSTRING as a RECORD:

```
DATA TYPE DSTRING           -- dynamic string
  RECORD
    INT pointer:
:
```

and the problem goes away.

7 Scope rules

The compiler follows the normal scoping rules as far as user defined operators are concerned – their scope is just like any other declaration.

8 Separate compilation

Operator definitions can be separately compiled and linked in the same way as FUNCTIONs and PROCs. INLINEd operators, of course, cannot be separately compiled and we have to use the #INCLUDE mechanism, rather than #USE, for libraries that contain them.

9 Literal constants

occam2.1 untyped literals are resolved according to the data type to which they are being assigned (or passed). For example:

```
INT x:
BYTE y:
SEQ
  x := 6
  y := 6
```

In the first case, the literal 6 is resolved to be of type INT and, in the second case, it is resolved as type BYTE.

This presents a problem for user defined operators. Given the code:

```
INT FUNCTION "&&" (VAL INT x, y) IS x:
INT FUNCTION "&&" (VAL BYTE x, y) IS INT y:
INT z:
SEQ
  z := 3 && 4
```

which version do we use? The solution is to reduce untyped literals to a default state – but only when used within an operand for a user-defined operator (so as not to change the semantics of occam2.1). The rules we have adopted are that undecorated literals, appearing in operand expressions for user-defined operators, are interpreted as follows:

```
   6  ->  INT
  'b' ->  BYTE
  6.0 ->  REAL32
```

Thus, `z` will be assigned the value `3` in the above.

10 Array and record constructors

There is a similar problem with array and record literals (or *constructors*). In occam2.1, if the type of a constructor is not clear – the type of variable (or parameter) to which it is being assigned (or passed) is used to resolve the type. For example:

```
DATA TYPE BLUE.INT IS INT:
[3]BLUE.INT a:
a := [0, 1, 3]
```

If the type of `a` had not been known, then the `0`, `1` and `3` could not have been resolved to `BLUE.INT`. This resolution problem occurs normally in occam2 when we have untyped abbreviations. For example:

```
VAL a IS [1, 2, 3]:
```

in which case, `a` is resolved to the type `[3]INT`.

This resoution problem also occurs when using constructors with user defined operators. For example, consider:

```
DATA TYPE THING
  RECORD
    INT count:
    REAL32 value:
:

INT FUNCTION "^" (VAL THING x, y) IS x[count] + y[count]:

INT result:
result := [12, 1.2] ^ [1, 3.1]
```

This will not compile because there is no way to determine the types of the two operands (the compiler could search all the definitions of user defined operators, searching for a match for the types, but this raises many other complicating issues). The solution is to decorate the record constructors with the typename – replacing the last line as follows solves the problem and will compile correctly:

```
result := [12, 1.2](THING) ^ [1, 3.1](THING)
```

If no type is specified then the compiler will try to resolve the constructor as an array. It will first search the constructor for a known type and then attempt to make each other (possibly type not specified) elements in the array match this type. If sucessful, the constructor gets typed as an array. For example:

```
INT FUNCTION "@" (VAL [6]BLUE.INT x) IS (INT x[0]):

INT res:
res := @ [1, 2, 3, 4, 5(BLUE.INT)]
```

will resolve all elements in the constructor to type `BLUE.INT`, and then to `[6]BLUE.INT`. If the last line had been:

```
res := @ [1, 2, 3, 4, 5]
```

the constructor would have been resolved to `[6]INT` and no match for the `@` operator would have been found.

There is one other slight modification – if a real literal is used as an operand to a user defined operator then it is resolved to `REAL32` – normal occam rules in this situation would throw it out without any decoration (e.g. `3.2(REAL32)`).

Note that none of these new rules effect the normal rules of occam2.1. Much care was taken not to damage the current type resolution system, so these new rules are only applicable when used with user defined operators.

11 Example libraries

Some example libraries are included with the 0.9beta release – firstly a complex number library that supports the use of the standard operators on a complex type. Please see the documents by David Wood on this and other similar libraries, including `INT128`s, `REAL128`s, `REAL80`s, `REAL40`s, sets, and a rational numbers library.

There is also a new dynamic string library that actually allocates memory for strings in the C world, using occam types to hold pointers to them. Strings can be created and deallocated (no automatic garbage collection is provided), converted to and from occam strings, and compared both for pointer equality (two variables of type STRING pointing at the same string), and textual equality (two strings textually equivalent). All the other comparison operators are also available. Concatenation of two strings is also supported using the new `++` operator. Documentation is included with the release.

12 Using user defined operators in KRoC

By default user defined operators are disabled in the current release of KRoC, so a compiler flag must be set to enable their use:

```
kroc -X2 prog.occ
```

This will compile prog.occ with the experimental compiler additions enabled.

13 Bugs

Please send bug reports to `ofa-bugs@ukc.ac.uk`.