Durham University Oberon-2 Compiler
Proposed Language Extensions

James King & Pedro Gonnet

August 31, 2013

1 Introduction

The main goal for the Durham University Oberon-2 Compiler (DUO2C) is to provide an expandable compiler compatible with most modern systems through the use of the LLVM infrastructure. Additionally, we aim to add a clean method of interfacing with external procedures defined in system libraries, and to extend the language with support for explicit vectorization of primitive types, an intrinsic component of high performance computing.

This document will outline the proposed additions to the language and provide examples of the new syntax's usage. For both of the extensions, care has been taken to follow the design principles employed by Oberon where possible.

2 External Interface Extension

To declare the existence of an external symbol to be linked, an *EXTERNAL* keyword has been added. This keyword, followed by a string literal specifying the external symbol's identifier, may replace the body of a procedure. An optional *IMPORT* keyword may also be used to import symbols from Dynamic Link Libraries on Windows, and translates to the *dllimport* flag used by LLVM. When using the compiler, the libraries that contain definitions for external symbols can be specified in the same style as the GCC *-l* option.

```
ExternalDecl = PROCEDURE IdentDef [FormalPars] ";"
               EXTERNAL [IMPORT] string.
```

Figure 1: The formal syntax for an external declaration. The tokens *IdentDef*, *FormalPars*, and *string* are as defined in the Oberon-2 specification[1].

This declaration has the same header format as a normal procedure definition, but with the external reference in the place of a body. The following is an example of how this new syntax is used, to refer to a symbol from the OpenGL Utility Toolkit[2]. The command line arguments used to link with this library will contain *-lglut* (or alternatively *-l glut*, for POSIX compliance), just like with GCC.

```
PROCEDURE InitWindowSize* (width, height : INTEGER);
EXTERNAL IMPORT "glutInitWindowSize";
```

Figure 2: An example of an externally defined procedure declaration.

Arguments of primitive types such as numerics and booleans are passed directly to the called procedure, but more complex types such as arrays and records are implicitly marshalled. The compiler stores an array as a pair of the array size and a pointer to the first element, which is marshalled by only sending the element pointer. Records contain a pointer to their corresponding virtual table used when dynamically dispatching bound procedure calls, and this will be stripped when marshalling records for use as arguments in external procedures.

3 SIMD Vector Extension

This second language extension adds some support for declaring and using vectors of primitive types, for systems that support Single Instruction Multiple Data (SIMD) operations. High level languages rarely support this technique, let alone feature it as a part of their syntax. Central to this extension is the addition of a new *VECTOR* data type, which may substitute any primitive type.

```
VectorType    = VECTOR ConstExpr OF PrimitiveType.
PrimitiveType = LONGINT  | INTEGER | SHORTINT
              | LONGREAL | REAL
              | BOOLEAN  | SET     | CHAR.
```

Figure 3: The formal syntax for the *VECTOR* type.

Additionally, a vector literal may be used which is a list of expressions surrounded by angle braces.

```
Vector   = "<" ExprList ">".
ExprList = Expr {"," Expr}.
```

Figure 4: The formal syntax for a vector literal.

Any operators applied to vectors are implicitly applied to each element of the vector independently. For systems that support it, these operations are parallelised as SIMD instructions by LLVM. These expressions are checked at compile time to ensure the vector sizes match, and the element types are compatible. For operations applied to a vector and a literal, the literal is implicitly promoted to a vector with the same number of elements as the other operand, where each element is the literal's original value.

```
VAR a, b : VECTOR 4 OF INTEGER;
BEGIN
    a := 4;              (* set every element of a to 4  *)
    b := <1, 2, 3, 4>;   (* assign a vector literal to b *)
    a := a + b * 2;      (* a is now <6, 8, 10, 12>      *)
END;
```

Figure 5: An example of operations applied to vectors.

Individual elements of a vector may be accessed or assigned to with the same syntax as array indexing.

```
a[0] := 12;
a[1] := a[0] * a[2];
b    := <a[3], a[2], a[1], a[0]>;
```

Figure 6: An example of indexing elements of a vector.

Finally, two procedures to transfer elements of vectors to and from arrays are provided, *VECLOAD* and *VECSTORE*.

```
VECLOAD(a, c, 2);  (* fill vector a with values from
                    * array c starting at index 2    *)

VECSTORE(b, c, 0); (* store elements from vector b in
                    * array c starting at the first index *)
```

Figure 7: Examples of *VECLOAD* and *VECSTORE*.

References

[1] Hanspeter Mössenböck, Niklaus Wirth, *The Programming Language Oberon-2*. `http://cseweb.ucsd.edu/~wgg/CSE131B/oberon2.htm` [Retrieved August 31, 2013].

[2] Khronos Group, *GLUT - The OpenGL Utility Toolkit*. `http://www.opengl.org/resources/libraries/glut/` [Retrieved August 31, 2013].