New Features of oo2c v2

$Revision: 1.7 $ covering oo2c-2.1.9

Michael van Acken mva@users.sf.net

Table of Contents

1 Introduction 1

2 Programs 1

3 Library Modules 2

4 Doc Comments 2
- 4.1 Inline Commands 2
- 4.2 Block Commands 4
- 4.3 Glyphs 4

5 Built-in Type STRING 5

6 Exceptions 5
- 6.1 `TRY` and `RAISE` 5
- 6.2 Checked vs Unchecked Exceptions 6
- 6.3 Implementation Notes 7

7 Parametric Types 7

8 Initialization Functions 9

Appendix A Example Module 9

Appendix B Document History 11

1 Introduction

`oo2c` v2 is a complete rewrite of the compiler and associated tools. Some highlights are

- Reduced internal complexity, at the expense of more computational overhead. This goes hand in hand with the ability to add experimental language features to the compiler.
- A simplified internal SSA code representation that eliminates the need to keep track the block structure of nested statements. This reduces the complexity of most code transformations. On the other hand, producing the target code becomes more difficult.
- A broader range of code transformations, including loop rewriting and partial redundancy elimination on top of the existing ones (common subexpression elimination, loop invariant code motion, constant propagation, algebraic transformations, and dead code elimination).
- A large set of regression tests, increasing the reliability and stability of compiler releases.
- More built-in support for building and installing third party packages.
- Writing `FOREIGN` modules to interface with external libraries has become easier. A simple `#include` pulls in all the run-time type and module meta data a module needs to provide to the run-time system.

The v2 compiler implements most, but not all, of the features of its predecessor. Most programs should compile without changes, although some of the more esoteric features and library modules have been dropped. The following sections summarize the omissions and additions. For the most part, the language extensions are experimental in nature and should not be considered final.

2 Programs

The biggest change to the behavior of the programs `oo2c`, `oob`, `ooef`, and `oowhereis` is the introduction of so called repositories to structure source code, intermediate, and executable files. A *repository* is a path to a directory *r* with ‘`r/src/`’ holding the source code, ‘`r/pkginfo.xml`’ any repository meta data, ‘`r/sym/`’ the symbol files, and so on. When looking for a particular module, all configured repositories are searched. Output files for a particular module (symbol file, C files, objects files, and so on) are placed into the repository that holds the module's source code.

The tool `oocn` is gone. Part of its functionality have been moved into the compiler `oo2c`: converting a module's public interface and its documentation strings to HTML and listing all uses of a particular object. Some of the command line options of `oo2c` have been replaced, and it has a whole new set of commands dealing with packages: build, install, uninstall, etc. Packages allow to install a set of library modules, module documentation, executables, and auxiliary files from the meta data of a ‘`pkginfo.xml`’ file.

Please refer to the man page of `oo2c` for the details.

3 Library Modules

With the switch from oo2c v1 to v2 a number of highly specialized and rarely used library modules were dropped, while at the same time whole families of modules were included that were distributed separately in the past.

Replaced by other modules: `Filenames` (use `OS:Path` instead), `Integers` (use `Object:BigInt`), `Kernel` and `Types` (functionality covered by `RT0`), and `Rts` (see `OS:ProcessManagement`).

Removed without replacement: `ComplexMath`, `JulianDay`, `LComplexMath`, `LocNumConv`, `LocNumStr`, `LocStrings`, `LocText`, `LocTextRider`, `Locales`, `LongInts`, `LowLReal`, `LowReal`, `OakFiles`, `OakIn`, `OakMath`, `OakMathL`, `OakStrings`, `Reals`, `Signal`, and `Strings2`.

New modules:

`ADT:*` A set of abstract data types, most importantly `ArrayList` and `Dictionary`. Also provides a framework for serializing arbitrary graphs of objects.

`IO:*` IO modules for files and sockets, providing an interface that is closer to the I/O capabilities provided by `libc`. Also mappers based on the new classes, and an abstraction for the '`select()`' function.

`OS:*` Various low-level modules dealing with file systems, file names, and processes.

Preliminary modules (their interface might change significantly in the future, although their general functionality will continue to be part of the core distribution):

`URI:*` Data types representing Uniform Resource Identifiers (both hierarchical and opaque), plus an URI parser.

`XML:*` A call-back based XML 1.0 parser. Also includes support for XML namespaces and validation.

4 Doc Comments

Documentation describing the public interface of a module can embedded into the source code using *doc comments*. Such comments start with a special delimiter '`(**`', like '`(**Some explanation. *)`', and refer to the declaration preceding them. Within doc comments, a subset of the GNU Texinfo command set can be used to mark up the text.

The following sections summarise the commands implemented by the OOC parser. Please refer to the Texinfo manual for a more thorough description of the their syntax and intended use.

4.1 Inline Commands

Inline commands take a single argument in curly braces like '`@code{ABS()}`'.

Font and style commands:

`@asis`	used with `@table` for entries without added highlighting
`@cite`	name of a book (with no cross reference link available)
`@code`	syntactic tokens
`@command`	command names
`@dfn`	introductory or defining use of a technical term
`@emph`	emphasis; produces *italics* in printout
`@file`	file name
`@kbd`	input to be typed by users
`@samp`	literal example or sequence of characters
`@strong`	stronger emphasis than `@emph`; produces **bold** in printout
`@var`	meta-syntactic variables (for example, formal procedure parameters)

References:

`@email`	an email address
`@url`	indicate a uniform resource locator (URL)
`@uref`	reference to a uniform resource locator (URL)

Referencing Oberon declarations:

`@omodule`	module name
`@oconst`	name of a constant
`@ofield`	designator referring to a record field
`@oparam`	parameter name
`@oproc`	designator referring to a procedure (normal or type-bound)
`@otype`	type name
`@ovar`	variable name

Designators can contain arbitrary '`.member`' parts, referring to members of modules, procedures, or records. For example, '`@oparam{MyModule.MyPointer.MyProc.param}`' would refer to the parameter '`param`' of the type-bound procedure '`MyProc`' from module '`MyModule`' that has '`MyPointer`' as its receiver type. Unless prefixed by a '`*`', names and designators must refer to existing declarations and the class of the indicated object must match the command name. If the designator is prefixed with a '`*`', then the designator is not checked by the compiler. In this case, it must begin with a fully qualified module name and use the canonical designator for the indicated object. This variant should only be used to refer to an object from another module if the module is not part of the `IMPORT` list.

4.2 Block Commands

Block commands start with the command '`@cmd`' on a line of its own, often followed by one or more arguments, and end with '`@end cmd`', also on a line of its own.

Lists and tables:

`@enumerate`	enumerated lists, using numbers or letters
`@itemize`	itemised lists with and without bullets
`@table`	two-column tables with highlighting
`@item, @itemx`	used with the above lists and tables for each entry

Paragraph formatting:

`@example`	example that is not part of the running text (fixed font)
`@noindent`	prevents paragraph indentation

Pre- and post-conditions:

`@precond`	pre-conditions of a procedure
`@postcond`	post-conditions of a procedure

4.3 Glyphs

`@@`	the character '`@`'
`@{`	the character '`{`'
`@}`	the character '`}`'
`@dots{}`	ellipsis '. . .'
`@bullet{}`	a '•', typically used with `@itemize`
`@minus{}`	a minus sign, '−'
`@result{}`	result of evaluating an expression, '⇒'
`---`	an em-dash for text '—'

5 Built-in Type STRING

The predefined identifier `STRING` is an alias for the type '`Object.String`', which implements Unicode strings. The semantics of this type are defined for the most part by the regular module '`Object`'[1], with two exceptions: the compiler converts string constants to instances of `STRING` automatically, and the operator '`+`' performs string concatenation.

String constants are assignment compatible with variables of type '`Object.Object`'. Such an assignment automatically converts the constant into an instance of `STRING`. That is, a string constant can be used instead of a `STRING` in an assignment, for a procedure argument passed to a value parameter, and as a function result. The string object is created once, as part of the module's initialization code, *not* each time its surrounding code is evaluated.

The operator '`+`' is defined for string operands and returns the concatenation of its operands. The result is of type `STRING`.

Please note that comparison of string values is done by means of the type-bound procedure '`String.Equals`'. The definition of the operators '`=`' and '`#`' has *not* been changed. That is, they test for object identity by comparing the strings' pointer values.

6 Exceptions

There are four user visible parts to exceptions:

- The module '`Exception`', defining the types '`Exception`', '`Checked`', and '`Unchecked`' and implementing the required run-time support.
- A new statement, `TRY`, to transfer control to exception handlers if a statement sequence raises an exception.
- A new predefined procedure, `RAISE`, to raise an exception.
- An extended syntax for procedures and procedure types, to declare which checked (or unchecked) exception can be raised by a procedure.

6.1 TRY and RAISE

Exceptions behave pretty much like their counterparts in Python or Java. The statement

```
TRY
  S
CATCH T1(t1):
  C1
CATCH T2(t2):
  C2
END;
```

is roughly equivalent to

1. Push exception handler for this `TRY` block ('`Exception.PushContext`').

[1] The module '`Object`' does not need to be imported to use `STRING`. It is part of the run-time system and as such included into every program.

2. Evaluate *S*, followed by '`Exception.PopContext`'. If there are any `RETURN` or `EXIT` statements within *S* that would cause control flow to leave the `TRY` statement, then they also do an implicit '`PopContext`' as part of the non-local exit.
3. If an exception is raised during *S*, then do

```
     Exception.PopContext;
     temp := Exception.Current();
     WITH temp: T1 DO
       t1 := temp;
       C1;
     | temp: T2 DO
       t2 := temp;
       C2;
     ELSE
       Exception.ActivateContext;
     END;
     Exception.Clear;
```

An exception is raised by calling the predefined procedure `RAISE` with an instance of '`Exception`'. This passes control to the nearest `CATCH` clause whose type is an extension of the raised exception, if such a clause exist. Otherwise, the exception is written to *stderr* and the program is aborted.

Within a `CATCH`, the optional name given in parenthesis refers to the current exception that triggered the `CATCH`. Its type is the one from the `CATCH` clause. The variable is read-only.

Within the module body no exceptions can be passed up, because there is no caller. As a consequence, any exception that is not caught explicitly is written to *stderr* and aborts the program. For checked exceptions, the compiler emits a warning if they are not caught in the module body.

6.2 Checked vs Unchecked Exceptions

There are two kinds of exceptions, checked and unchecked. The compiler enforces a stricter set of restrictions on the usage of checked exceptions. During program run-time, there is no difference between the two.

A *checked* exception *E* must either be caught within a procedure, or the procedure must declare that it may pass an exception of type *E* up to its caller. For example,

```
  PROCEDURE P() RAISES E;
```

declares that evaluation of '`P`' may raise an exception of type *E*, or an extension thereof.

An exception is of the "checked" variant if it is an extension of the class '`Exception.Checked`'. The base class '`Exception.Exception`' is also treated as "checked".

On the other hand, an *unchecked* exception can be raised anytime, without the need to declare or catch it. It is possible to add an unchecked exception to the '`RAISES`' list of a procedure declaration for documentation purposes. In this case, it is *not* a compile time error if the caller does not catch this exception, and does not declare to pass it on. An exception is "unchecked" if it is an extension of the class '`Exception.Unchecked`'.

Please note that the record type of the exception class, ‘Exception.ExceptionDesc’, is no longer exported. This means that it is not possible to extend it directly. Applications must use either ‘Exception.CheckedDesc’ or ‘Exception.UncheckedDesc’ to create specialized exception classes.

6.3 Implementation Notes

A TRY block is mapped to C’s ‘setjmp()’ function by oo2c. The amount of data stored by this function depends on the target architecture and may differ by a factor of ten or more. For example, on a ‘ix86’ processor, only 72 bytes are stored, while on a ‘PPC’ this is 768 bytes. As a consequence, the work done by the program within a TRY should be large enough to amortize the costs of the TRY for all possible targets.

On some systems[2], raising an exception also stores information about the top 20 activation frames on the call stack. This means that raising an exception can be moderately expensive as well. Therefore they should only be used to report exceptional conditions that are rarely triggered.

7 Parametric Types

A parametric type can be seen as a type definition with a certain degree of freedom. The freedom comes in form of type parameters acting as placeholders for type arguments that are provided when the parametric type is used in a particular context. There are two restrictions on type parameters and type arguments: the parameter must be based on a record pointer, and the argument must be an extension of the parameter’s base type.

Take for example the type ‘ArrayList’. The element type of the list can be any type derived from ‘Object’, like ‘MyElementType’, which is provided when creating an ‘ArrayList’ variable:

```
TYPE
  ArrayList*(E: Object.Object) = POINTER TO ArrayListDesc(E);
  ArrayListDesc*(E: Object.Object) = RECORD
    ...
  END;
...
VAR myList: ArrayList(MyElementType);
```

The compiler statically detects any uses of ‘myList’ or of its methods that are incompatible with the declared element type ‘MyElementType’.

The implementation of parametric types extends the syntax in three places:

- A type declaration can have a list of type parameters.
- Usage of a parametric type can provide a list of type arguments.
- For a type-bound procedure of a parametric type, the receiver declaration must provide the names of local aliases for the type parameters of the base type. These names act as type variables within the procedure.

[2] At the time of writing, this means all systems running with GNU libc and implementing the ‘backtrace()’ function.

(For the details, please refer to the EBNF grammar at the end of this section.)

A type declared with a type parameter list like '`T(t1:B1,t2:B2,...,tn:Bn)`' is called a *parametric type*. The formal type *Bi* of a type parameter declaration is called its type bound. The type bound must be a record pointer. A type name *ti* is visible to the end of the type declaration.

For a qualified type expression of the form '`T(A1,A2,...,An)`',

a. the type *T* must be a parametric type,
b. it must have the same number of type arguments as there are type parameters, and
c. each actual type parameter *Ai* is either an extension of the corresponding type bound *Bi*, or *Ai* is a type variable whose bound is an extension of *Bi*.

If *T* is a parametric type as defined above, then the type expression '`T`' (without any type arguments) is equivalent to the qualified type "`T(B1,B2,...,Bn)`', where each type argument equals the corresponding type bound. *Bi* and '`Ai`' can be forward references to types that are defined later.

A type-bound procedure of a parametric type must define a list of type names after its receiver type, for example

```
PROCEDURE (r: P(T1,T2,...)) TBProc(...);
```

Each name is a type alias for the corresponding type parameter of the procedure's *base record* type. Within a type-bound procedure, a variable '`v: Ti`', with *Ti* declared with a type bound '`Ti: Bi`', can for the most part be used like it had been declared as '`v: Bi`'. The exceptions are that it can only be assigned values of type *Ti* (or `NIL`), and that `NEW` is not applicable to such a variable.

Two qualified types are considered to be the same type, if they have the same base type and if their corresponding type arguments are of the same type.

`NIL` is assignment compatible with a parametric type if it is assignment compatible with the type's base type.

'`NEW()`' is applicable to a variable of parametric type if its base type is a pointer.

Syntax:

```
TPSection  = ident {"," ident} ":" Qualident.
TypePars   = "(" [TPSection {";" TPSection}] ")".
TypeDecl   = IdentDef [TypePars] "=" Type ";".

QualType   = Qualident ["(" [QualType {"," QualType}] ")"].
Type       = QualType|ArrayType|RecordType|PointerType|ProcType.
RecordType = "RECORD" ["("QualType")"] ... "END".
FormalPars = ["(" [FPSection {";" FPSection}] ")" [":" QualType]].

AliasList = "(" [ident {"," ident}] ")".
Receiver = "(" ["VAR"] ident ":" ident  [AliasList] ")".
```

Note: Polymorphic procedures, where free type parameters are added to a formal parameter list to place an additional restriction on the acceptable argument lists of calls, are currently not supported.

8 Initialization Functions

The compiler provides a common notation to define an initialization procedure for an object, to redefine the initialization procedure within an extended type, and to call this procedure automatically when creating an object. For this, a class may provide a type-bound procedure `INIT`, like

```
PROCEDURE (l: List) INIT*(initialSize: LONGINT);
```

Such a procedure has the special property that a call to it always binds to the procedure bound to the *static* type of the receiver, *not* the dynamic one. In this it behaves like a call to a normal procedure, where the actual code to be evaluated is known at compile time.

`INIT` must not return a result, and it must be exported. Its formal parameters do not need to match the parameter list inherited from the base type, if one exists. If the base type provides an '`INIT`' procedure, but there is no super call like '`l.INIT^(...)`', then the compiler produces a warning.

As a shortcut to create an object is to use '`NEW()`' as a function, passing the type of the the new object as its first argument. '`v := NEW(T,a1,a2,...)`' is equivalent to

```
VAR temp: T;
...
NEW(temp);
temp.INIT(a1,a2,...);
v := temp;
```

The initialization call is omitted if '`T`' does not define an '`INIT`' procedure.

Note: The definition of '`NEW(v)`' when called as a procedure has not changed. That is, with this use of '`NEW`' the '`INIT()`' procedure is *not* called implicitly for the new object.

Possible changes: Right now, the type argument of '`NEW()`' must be a record pointer. This might be relaxed to any pointer type in the future.

Appendix A Example Module

The modules below exercise most of the features described above: strings, parametric types, exceptions, and initialization functions.

```
MODULE Example:Map;
(**A simple parametric map type using linked lists. *)

IMPORT Object, E := Exception;

TYPE
  Key* = Object.Object;
  Value* = Object.Object;

TYPE
  Entry(K: Key; V: Value) = POINTER TO EntryDesc(K, V);
  EntryDesc(K: Key; V: Value) = RECORD
    key: K;
    value: V;
    next: Entry(K, V);
  END;
```

```
TYPE
  Map*(K: Key; V: Value) = POINTER TO MapDesc(K, V);
  MapDesc(K: Key; V: Value) = RECORD
    name: STRING;
    entries: Entry(K, V);
  END;

PROCEDURE (map: Map(K,V)) Put*(key: K; value: V);
VAR
  entry: Entry(K, V);
BEGIN
  NEW(entry);
  entry.key := key;
  entry.value := value;
  entry.next := map.entries;
  map.entries := entry;
END Put;

PROCEDURE (map: Map(K, V)) Get*(key: K): V
RAISES E.Exception;
VAR
  entry: Entry(K, V);
BEGIN
  entry := map.entries;
  WHILE entry # NIL DO
    IF key.Equals(entry.key) THEN
      RETURN entry.value;
    END;
    entry := entry.next;
  END;
  RAISE(NEW(E.Exception,
            "Map '"+map.name +"' undefined for '"+key.ToString()+"'"));
END Get;

PROCEDURE (map: Map(K, V)) INIT*(name: STRING);
BEGIN
  map.name := name;
  map.entries := NIL;
END INIT;

END Example:Map.

MODULE TestMap;

IMPORT
  Example:Map, Object, Object:Boxed, E := Exception, Out;

TYPE
  Key = STRING;
  Value = Object.Object;

PROCEDURE ShowEntry(map: Map.Map(Key, Value); key: Key);
VAR
  value: Value;
BEGIN
  TRY
    value := map.Get(key);
    Out.Object("The value for "+key+" is "+value.ToString());
  CATCH E.Exception(e):
    Out.Object("Exception: "+e.GetMessage());
  END;
  Out.Ln
END ShowEntry;

PROCEDURE Test;
VAR
  map: Map.Map(Key, Value);
BEGIN
  map := NEW(Map.Map(Key,Value), "my map");

  map.Put("one",  Boxed.NewLongReal(1.0));
  map.Put("pi",   Boxed.NewLongReal(3.14159265358979));
  map.Put("true", Boxed.NewBoolean(TRUE));
  map.Put("a rose", "a rose");

  ShowEntry(map, "a rose");
  ShowEntry(map, "one");
  ShowEntry(map, "two");
END Test;

BEGIN
  Test;
END TestMap.
```

Appendix B Document History

1.7 A type variable can be used on the right hand side of a type test or type guard. Introduced with `oo2c-2.1.9`.

1.5 Adds information regarding "checked" vs "unchecked" exceptions, which were introduced with `oo2c-2.0.13`. Covers `oo2c-2.0.15`.

1.4 First release. Covers `oo2c-2.0.12`.