Nickel – version 1.4

(`ocamlwrap` command-line tool)

http://nickel.x9c.fr

February 6, 2010

Contents

Chapter 1

Introduction

Classically, the Objective Caml[1]programmer that needed to interface its program with Java[2] had to use O'Jacare. O'Jacare[3] is a code generator that bridges Java and Objective Caml objects. It does so by using a low level C interface and is built upon the camljava library[4].

By using Cadmium (`http://cadmium.x9c.fr`) whether alone or with Cafesterol (`http://cafesterol.x9c.fr`), it is possible to run an Objective Caml program on a Java Virtual Machine. It is then particularly interesting to access Java classes from the Objective Caml program. This can be done in plain Cadmium by using the Cadmium runtime library that allows access to Java reflection mechanism through modules `Cadmium`, and `CadmiumObj`. However, when numerous Java elements need to be used, it is far more convenient to use Nickel to generate bindings.

Since version 1.1, Nickel also allows to generate bindings to a database. Such bindings allow to evaluate SQL statements by simply calling an Objective Caml function, the statement execution being done through a JDBC[5] driver. The generated bindings are type-safe, meaning that the function signature is directly determined from the statement parameters and result. This generator was inspired by the PGOCaml[6] tool. Both tools share the same principle: the signature of the functions is determined by connecting to the database at *code-generation-time* in order to query the database for metadata.

Nickel is a bridge generator that produces Java, C and Objective Caml files. The Objective Caml file defines the module providing access to Java elements and the Java file defines the primitive provider with the supporting primitives. The C file provides a fake implementation of these primitives. Such a fake implementation is needed to allow compilation by the `ocamlc` compiler (by fake, we mean that every primitive raises an exception upon invocation).

Nickel, in its 1.4 version, is designed to work with at least version 1.4 of Cadmium. Naturally,

[1]The official Caml website can be reached at `http://caml.inria.fr` and contains the full development suite (compilers, tools, virtual machine, *etc.*) as well as links to third-party contributions.

[2]The official website for the Java Technology can be reached at `http://java.sun.com`.

[3]`http://www.pps.jussieu.fr/~henry/ojacare`

[4]The camljava library allows JNI access from Objective Caml and is available at `http://pauillac.inria.fr/~xleroy/software.html`.

[5]`http://java.sun.com/javase/technologies/database/`

[6]`http://developer.berlios.de/projects/pgocaml/`

the use of Cadmium entails the use of Objective Caml (version 3.11.2 and above).

Chapter 2

Building Nickel

Nickel can be built from sources using Ant[1] (at least version 1.7.1) under Java 1.6. Before invoking Ant, one is advised to edit `build.properties` to parametrize Ant targets; properties are not described here as they are self-explanatory. The target related to tests need both JUnit[2] and Cobertura[3] to be installed in the `lib` directory.
The following targets are available from the `build.xml` file:

`all-tests` performs all tests and generates coverage report

`clean` cleans 'classes' and 'javadoc' directories

`clean-classes` cleans 'classes' directory

`clean-javadoc` cleans 'javadoc' directory

`clean-reports` cleans 'reports' directory

`clean-tests` cleans 'tests' directory

`compile` compiles all files

`compile-tests` compiles and instruments all files

`coverage` generates coverage report

`deploy` compiles files then creates jar files

`func-tests` performs functional tests

`javadoc` generates javadoc

`style` generates style report

`unit-tests` performs unit tests

`veryclean` cleans all directories

[1] Apache Ant is a build tool, available at http://ant.apache.org.
[2] Java framework for unit tests, available at http://www.junit.org (tested with version 4.4).
[3] Java code coverage tool, available at http://cobertura.sourceforge.net (tested with version 1.9.1).

Chapter 3

Running Nickel

There are three ways Nickel can be used: as a command-line utility, as a GUI application, or as an Ant task. The main class of the first version is `fr.x9c.nickel.Main` whereas the main class of the second one is `fr.x9c.nickel.MainGUI`. It is also possible to launch the first version by executing `ocamlwrap.jar` and the second one by executing `ocamlwrap-gui.jar`.

3.1 Command-line version

The command-line version accepts a list of xml files to process as well as the following switches:

`--verbose` or `-verbose`
print progress information during process

`--debug` or `-debug`
print debug information during process

`--module-kind=<classes|database>` or `-module-kind <classes|database>`
kind of module to produce (defaulting to "classes")

`--version` or `-version`
show program version and exit

`--help` or `-help`
show this message and exit

`--java-dir=<directory>` or `-java-dir <directory>`
directory for generated Java files (defaulting to ".")

`--java-package=<package-name>` or `-java-package <package-name>`
package of generated Java files (defaulting to "pack")

`--ocaml-dir=<directory>` or `-ocaml-dir <directory>`
directory for generated OCaml files (defaulting to ".")

`--c-dir=<directory>` or `-c-dir <directory>`
directory for generated C files (defaulting to ".")

`--primitive-prefix=<string>` or `-primitive-prefix <string>`
prefix to primitive names (defaulting to "")

`--no-c-file` or `-no-c-file`
disable C file generation

`--generics` or `-generics`
enable support for generics (experimental)

The command-line version returns an exit code of 0 when successful, and an exit code of 1 otherwise.

3.2 GUI version

The GUI version accepts the same command-line switches as the command-line version and then opens a dialog to allow the user to modify the following elements:

- the directory for generated Java files;
- the package of generated Java files;
- the directory for generated Objective Caml files;
- the directory for generated C files;
- the prefix to primitive names;
- the kind of module to generate;
- the XML file to process.

3.3 Ant task

To be used in an Ant build file, the task should first be declared using a `<taskdef` element as shown by code sample 1. The Ant task can then be used as shown by code sample 2: it waits for an embedded `<fileset` element describing the XML files to process and supports the following attributes:

`verbose` whether execution should be verbose (boolean defaulting to *false*);

`debug` whether debug information should be printed (boolean defaulting to *false*);

`javadir` directory for generated Java files (directory defaulting to project base directory);

`javapackage` package for generated Java files (mandatory, no default value);

`ocamldir` directory for generated Objective Caml files (directory defaulting to project base directory);

`cdir` directory for generated C files (directory defaulting to project base directory);

`prefix` prefix to primitive names (string defaulting to the empty string);

`modulekind` kind of module to generate (string defaulting to *"classes"*).

Code sample 1 Ant file (task definition).

```
<path id="cp">
  <pathelement location="/path/to/ocamlwrap.jar"/>
</path>

<taskdef name="nickel" classname="fr.x9c.nickel.AntTask" classpathref="cp"/>
```

Code sample 2 Ant file (task use).

```
<target name="mytarget">
  <nickel javapackage="mypackage">
    <fileset dir=".">
      <include name="mymodule.xml"/>
    </fileset>
  </nickel>
</target>
```

Chapter 4

Compiling generated files

Upon successful invocation, Nickel produces for each passed XML file:

- an Objective Caml module providing functions and classes to access Java elements;
- a Java class that is a primitive provider;
- a C file that contains fake implementations (*i. e.* always raising an exception) for primitives.

Any C compiler can be used to compile the C file. One has only to pass the compiler the path of Objective Caml C header files (`/usr/local/lib/ocaml/caml` under usual Unix-like configuration). This means that the gcc[1] command-line to compile a generated `file.c` is:

```
gcc -I /usr/local/lib/ocaml/caml -c file.c
```

However, it is simpler to just invoke `ocamlc` to compile the C file. It will automatically called the C compiler chosen at Objective Caml configuration. It nevertheless remains necessary to pass the path of Objective Caml header files:

```
ocamlc -ccopt -I -ccopt /usr/local/lib/ocaml/caml -c file.c
```

Any Java 1.6 compiler can be used to compile the Java file. One has only to pass the compiler the path of the Cadmium jar file (as well as the classpath of any wrapped class outside the JDK, if any). This means that, using javac[2], the command-line to compile a generated `File.java` is:

```
javac -target 1.6 -cp /path/to/ocamlrun.jar File.java
```

To use the generated Objective Caml module, one first needs to generate its interface. It can be done using the following command-line (it needs Cadmium to be installed in the Objective Caml distribution):

```
ocamlc -I +cadmium -i file.ml > file.mli
```

Then, to be used with Cadmium as an interpreter, both (mli and ml) files can be compiled using the following commands:

```
ocamlc -I +cadmium -c file.mli
ocamlc -I +cadmium -c file.ml
```

[1] GNU Compiler Collection – http://gcc.gnu.org.
[2] Java compiler, bundled with Sun JDK – http://java.sun.com.

Finally, the main program can be compiled and linked by the following command (where `source.ml` contains the source of the main program, and `file.o` has been compiled from `file.c`):

```
ocamlc -I +cadmium -o prog -custom cadmiumLibrary.cma file.o file.cmo source.ml
```

Alternatively, if one wants to use the Cafesterol (*i.e.* `ocamljava`) compiler, the commands become (where *cp* is the classpath to access to the Java class compiled from compiled `File.java`, and *fqcn* its fully-qualified classname):

```
ocamljava -I +cadmium -c file.mli
ocamljava -classpath <cp> -provider <fqcn> -I +cadmium -c file.ml
ocamljava -o prog.jar -I +cadmium cadmiumLibrary.cmja file.cmj source.ml
```

Chapter 5

Generating bindings to classes

This chapter presents the generator allowing to produce bindings to Java classes. The full DTD of input files is given by code sample 3. The associated semantics is given below. Each XML file represents an Objective Caml module and acts as a collection of class bindings between Java and Objective Caml. For each class in the XML file, the Java class name is given as well as the name of the corresponding Objective Caml class to generate.

5.1 Meta elements

Meta tags are key-value pairs that give information on how the contents of the XML file should be interpreted. In the current version, only one meta key is recognized: `CLASSPATH`. It can be used to specify the classpath to be used when trying to load Java classes in order to produce bindings. Java classes will be found if and only if they appear either in the classpath of the Nickel process, or in an element pointed by a meta `CLASSPATH` element.

5.2 Class declarations

A Nickel XML files begins by optional `external` definitions. Such definitions are useful if some bindings defined in another XML file should be used in the current file. Each `external` declaration defines such a binding between a Java class (fully qualified name given by `java-name` attribute) and an Objective Caml class (name with module prefix given by `ocaml-name`). The definition of external bindings is useful because of the way Nickel produces bindings. When creating a binding for a method, constructor or field, Nickel must determine its related types (for parameters, return values, *etc.*); for each element, the chosen type is the most specific type Nickel knows. This means that if an element type is a class that is not defined in the current XML file, Nickel will use its closest parent, possibly `java.lang.Object`[1]. It is thus important to provide external definitions (if available) to ensure that bindings are as close as possible to the Java underlying hierarchy. Table 5.1 gives the mapping of *primitive* types (conversion between Objective Caml char and int resulting from Java char are done by `char_of_java_char` and `java_char_of_char` in either `Cadmium` or `CadmiumObj` module).

[1] `java.lang.Object` is always used for *generic* types, and type parameters should not be given (which means that one should write `java.util.List` rather than `java.util.List<T>`).

Code sample 3 DTD for XML files.

```
<!ELEMENT module (meta*,external*,(class|interface|enum)*)>
<!ATTLIST module name CDATA #REQUIRED>

<!ELEMENT meta EMPTY>
<!ATTLIST meta name CDATA #REQUIRED>
<!ATTLIST meta value CDATA #REQUIRED>

<!ELEMENT external EMPTY>
<!ATTLIST external java-name CDATA #REQUIRED>
<!ATTLIST external ocaml-name CDATA #REQUIRED>

<!ELEMENT class ((field|fields)*,(constructor|constructors)*,(method|methods)*)>
<!ATTLIST class java-name CDATA #REQUIRED>
<!ATTLIST class ocaml-name CDATA #REQUIRED>
<!ATTLIST class wrapper (yes|no) "no">

<!ELEMENT interface ((field|fields)*,(method|methods)*)>
<!ATTLIST interface java-name CDATA #REQUIRED>
<!ATTLIST interface ocaml-name CDATA #REQUIRED>
<!ATTLIST interface wrapper (yes|no) "no">

<!ELEMENT enum ((field|fields)*,(method|methods)*)>
<!ATTLIST enum java-name CDATA #REQUIRED>
<!ATTLIST enum ocaml-name CDATA #REQUIRED>

<!ELEMENT field EMPTY>
<!ATTLIST field name CDATA #REQUIRED>

<!ELEMENT fields EMPTY>
<!ATTLIST fields pattern CDATA #REQUIRED>

<!ELEMENT constructor EMPTY>
<!ATTLIST constructor signature CDATA #REQUIRED>

<!ELEMENT constructors EMPTY>
<!ATTLIST constructors pattern CDATA #REQUIRED>

<!ELEMENT method EMPTY>
<!ATTLIST method signature CDATA #REQUIRED>

<!ELEMENT methods EMPTY>
<!ATTLIST methods pattern CDATA #REQUIRED>
```

Java type	Nickel-mapped Objective Caml type
boolean	bool
java.lang.Boolean	bool
byte	int
java.lang.Byte	int
char	int
java.lang.Character	int
double	float
java.lang.Double	float
float	float
java.lang.Float	float
int	int32
java.lang.Integer	int32
long	int64
java.lang.Long	int64
short	int
java.lang.Short	int
java.lang.String	string
void	unit
java.lang.Void	unit

Table 5.1: Mapping of Java *primitive* types.

After `external` elements, the XML file consists of a list of `class`, `interface` and `enum` elements. Each of theses elements defines a mapping from a Java class (whose fully qualified name is given by attribute `java-name`) to an Objective Caml class (whose name is given by attribute `ocaml-name`). `class` elements may contain field, constructor and methods elements while `interface` and `enum` elements may only contain field and method elements.

5.3 Field declarations

Field elements can be declared either by a `field` or a `fields` tag. The first tag accepts a `name` attribute while the second one accepts a `pattern` attribute. A `field` tag is used to map the single field whose name is given; a `fields` tag is used to map a set of fields whose name pattern is given. Table 5.2 presents the meta-characters that can be used in a pattern.

Meta-character	Semantics
`*`	matches 0 or more characters
`+`	matches 1 or more characters
`?`	matches 0 or more characters (excluding `,`)
`!`	matches 1 or more characters (excluding `,`)

Table 5.2: Meta-characters to be used in patterns.

As an example, if *myField* is a field of type *t* in the concerned class, `<field name="myField"/>` generates the following:

- `method field'myField'get : t` to read the field value;
- `method field'myField'set : t -> unit` to change the field value, generated only if *myField* is not *final*.

These methods are generated independently of the *static* nature of the field. When related to a *static* field, methods can be called on any instance, even a *null* one.

5.4 Constructor declarations

Constructor elements can be declared either by a `constructor` or a `constructors` tag. The first tag accepts a `signature` attribute while the second one accepts a `pattern` attribute. A `constructor` tag is used to map the single constructor whose signature is given; a `constructors` tag is used to map a set of constructors whose signature pattern is given. A constructor signature has the following format: `(params)` where *params* is a comma-separated list of parameters (no whitespace is allowed). This means that the `(*)` pattern matches all constructors. Parameters are either fully qualified class names or primitives names, a trailing `[]` bracket couple being used for each array dimension.

As an example, if the concerned class contains a constructor taking a string and an integer as parameters, `<constructor signature="(java.lang.String,int)"/>` generates the following:

- `` `StringInt of string * int32 `` polymorphic variant, name being the concatenation of the Java types

Afterwards, an instance can be created by: `new myClass (` `` `StringInt ("a", 5l)) `` (warning: the integer value is 5 with an ending *ell*, as it is an `int32` value).

A Java enum class cannot specify any constructor but for each enum value *V* of the enum, a polymorphic variant `` `V `` is generated.

5.5 Method declarations

Method elements can be declared either by a `method` or a `methods` tag. The first tag accepts a `signature` attribute while the second one accepts a `pattern` attribute. A `method` tag is used to map the single method whose signature is given; a `methods` tag is used to map a set of methods whose signature pattern is given. A method signature has the following format: `name(params)` where *params* is a comma-separated list of parameters. This means that the `*(*)` pattern matches all methods.

As an example, if the concerned class contains a method *meth* taking a string and an integer as paramaters and returning a float, `<method signature="meth(java.lang.String,int)"/>` generates the following:

- `method meth : string -> int32 -> float`

Afterwards, the method can be called on an instance named *inst* by: `inst#meth "a" 51`. This method is generated independently of the *static* nature of the Java method. When related to a *static* Java method, the Objective Caml method can be called on any instance, even a *null* one.

In case of name clash (due to either Java method overloading, or clash with an Objective Caml keyword), the method name is appended with first `'1`, then `'2`, and so on.

5.6 Wrappers

When binding a Java class or interface, the user may require Nickel to generate a wrapper by setting the `wrapper` property to `yes` (this attribute is optional and its default value is `no`).
A wrapper is a Java class that wraps an Objective Caml instance to allow this instance to be used by others Java classes. It is useful as it allows to implement in Objective Caml a listener to be registered with a Java instance. The last part of this document presents an example where a *java.awt.event.ActionListener* is written in Objective Caml and added to the listeners of a *javax.swing.JButton* instance.

When a wrapper is requested for a Java class or instance, Nickel generates the following:

- `‘Cd’wrap of < m1: p1 -> r1; m2: p2 -> p2’ -> r2; ..>` polymorphic variant where m_i are methods with their p_i parameter types and r_i return types.
  If wrapping is done around an interface the `‘Cd’wrap` variant is generated; if wrapping is done around a class, a variant is generated for each constructor matched by either a `constructor` or a `constructors` tag.
  The methods are all the methods matched by `method` and `methods` tags of the related class or interface.

Afterwards, a wrapped instance can be created by: `new myClass (‘Cd’wrap inst)` where *inst* is an Objective Caml instance providing m_i methods with compatible parameter and return types.

5.7 Exceptions

When a an Objective Caml program is ran under Cadmium, any Java exception can be caught on the Cadmium side by using the `Cadmium.java_exception` exception. Such an Objective Caml exception, takes 3 parameters that are:

- the fully qualified class name of the exception (as a `string`);
- the exception message (as a `string`);
- the actual exception instance (as a `Cadmium.java_object`).

To throw a Java exception from the Objective Caml side (useful in wrapped code), the programmer can use either `Cadmium.throw_exception` or `CadmiumObj.throw_exception` function that respectively take a `Cadmium.java_object` and a `CadmiumObj.jObject` and return `unit` (both function raise `Invalid_argument` when passed a Java instance that is not a *java.lang.Throwable* one). It is of course possible to bind a Java exception using a `class` tag to allow easy construction of such an exception from the Objective Caml side; then the exception is thrown by `Cadmium.throw_exception (inst#cd’this)` or `CadmiumObj.throw_exception (inst :> CadmiumObj.jObject)` where *inst* is an instance of a Java exception.

5.8 Complete example

This section shows how to use Nickel in conjunction with Cadmium/Cafesterol to write in Objective Caml a small Java Swing application. This application will have a text area and a button and will exit upon button click by printing the contents of the text area.

Code sample 4 shows the makefile used by this example when using Cadmium as an interpreter. The `prepare` target first invokes Nickel to generate C, Java and Objective Caml files and then produces the Objective Caml module interface from its implementation source. The `compile` target first compiles module interface and implementation, then compiles the generated C and Java files, finally the complete Objective Caml program is compiled and linked. The `run` target runs the compiled Objective Caml program under Cadmium (if not run under Cadmium, the program fails upon an Objective Caml exception exhibiting that Cadmium is missing and required).

Code sample 4 Makefile for example.

```
JAVA=java
JAVAC=javac
OCAMLC=ocamlc -I +cadmium
CADMIUM_JAR=../../../Cadmium/deploy/ocamlrun.jar
CADMIUM_MAIN=fr.x9c.cadmium.Main
NICKEL_JAR=../../deploy/ocamlwrap.jar


all: clean prepare compile run

prepare:
	mkdir pack
	$(JAVA) -jar $(NICKEL_JAR) --java-dir=pack source.xml
	$(OCAMLC) -i java.ml > java.mli

compile:
	$(OCAMLC) -c java.mli
	$(OCAMLC) -c java.ml
	$(OCAMLC) -ccopt -I -ccopt /usr/local/lib/ocaml/caml -c java.c
	$(JAVAC) -target 1.6 -cp $(CADMIUM_JAR) pack/Java.java
	$(OCAMLC) -o prog -custom cadmiumLibrary.cma java.o java.cmo source.ml

run:
	$(JAVA) -cp .:$(CADMIUM_JAR) $(CADMIUM_MAIN) --providers=pack.Java ./prog

clean:
	rm -f java.c java.ml java.mli pack/Java.java
	rm -f *.cm* *.o prog pack/*.class
	rm -fr pack
```

Code sample 5 shows the makefile used by this example when using Cadmium with the Cafesterol compiler (*i.e.* `ocamljava`). The `prepare` target first invokes Nickel to generates C, Java and Objective Caml files and then produces the Objective Caml module interface from its source; finally the generated Java file is compiled. The `compile` target first compiles module interface and implementation, then compiles the main module and links both modules. The `run` target runs the compiled Objective Caml program using compiled libraries.

Code sample 5 Makefile.cafesterol for example.

```
JAVA=java
JAVAC=javac
JAR=jar
OCAMLJAVA=ocamljava -I +cadmium -java-package fr.x9c.nickel.example
CADMIUM_JAR=../../../Cadmium/deploy/ocamlrun.jar
NICKEL_JAR=../../deploy/ocamlwrap.jar
LIB_DIR=‘ocamljava -where‘
LIBS=$(LIB_DIR)/stdlib.jar:$(LIB_DIR)/cadmium/cadmiumLibrary.jar


all: clean prepare compile run

prepare:
        mkdir pack
        $(JAVA) -jar $(NICKEL_JAR) --java-dir=pack --java-package=pack source.xml
        $(OCAMLJAVA) -i java.ml > java.mli
        $(JAVAC) -target 1.6 -cp $(CADMIUM_JAR) pack/Java.java

compile:
        $(OCAMLJAVA) -c java.mli
        $(OCAMLJAVA) -classpath . -provider pack.Java -c java.ml
        $(OCAMLJAVA) -c source.ml
        $(OCAMLJAVA) -o prog.jar -standalone cadmiumLibrary.cmja java.cmj source.cmj
        $(JAR) uf prog.jar pack/*

run:
        $(JAVA) -jar prog.jar

clean:
        rm -f java.c java.ml java.mli pack/Java.java
        rm -f *.cm* *.j* prog.jar pack/*.class
        rm -fr pack
```

Code sample 6 shows the Objective Caml source of the program. The opened `Java` module is the module generated by Nickel. The `lock` object is used to handle the end of the program: the main code waits upon this object and the `actionPerformed` method notifies this instance. The main code constructs the frame, text area (wrapped in a scroll pane) and the button. Then an action listener is constructed by wrapping an instance of the `quit`

class. Finally, the user interface is created and made visible, a wait is issued on the `lock` object, and upon notification the final contents of the text area is printed on the standard output.

Code sample 6 Objective Caml source for example.

```
open CadmiumObj
open Java

let lock = new jObject ‘Void

class quit = object
  method actionPerformed (_ : jObject) = lock#notify
end

let () =
  let frame = new jFrame (‘String "Nickel test") in
  let text = new jTextArea (‘String ("This is a Nickel/Cadmium example.\n"
                                     ^ "One can input text in this area.")) in
  let view = new jScrollPane (‘Component (text :> jComponent)) in
  let button = new jButton (‘String "OK") in
  let listener = new jActionListener (‘Cd’wrap (new quit)) in
  button#addActionListener listener;
  ignore (frame#getContentPane#add "Center" (view :> jComponent));
  ignore (frame#getContentPane#add "South" (button :> jComponent));
  frame#setSize 320l 240l;
  frame#setVisible true;
  lock#wait;
  print_endline "*** final text:";
  print_endline text#getText;
  exit 0
```

Code sample 7 shows the XML file used to generate C, Java and Objective Caml files.

As a concluding remark, we want to draw the reader's attention on the fact that Swing user interfaces can be created more easily by using SwiXml and its associated Cadmium bindings. SwiXml is a powerful and very convenient library that allows one to render a GUI from an XML description (available at http://www.swixml.org). The corresponding Cadmium bindings are part of Cadmium since version 1.1.

Code sample 7 Nickel file for example.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!DOCTYPE module SYSTEM "dtds/module.dtd">

<module name="Java">
  <interface java-name="java.awt.event.ActionListener"
             ocaml-name="jActionListener"
             wrapper="yes">
    <methods pattern="*(*)"/>
  </interface>
  <class java-name="javax.swing.JButton" ocaml-name="jButton">
    <constructor signature="(java.lang.String)"/>
    <method signature="addActionListener(java.awt.event.ActionListener)"/>
  </class>
  <class java-name="javax.swing.JFrame" ocaml-name="jFrame">
    <constructor signature="(java.lang.String)"/>
    <method signature="getContentPane()"/>
    <method signature="setSize(int,int)"/>
    <method signature="setVisible(boolean)"/>
  </class>
  <class java-name="javax.swing.JTextArea" ocaml-name="jTextArea">
    <constructor signature="(java.lang.String)"/>
    <method signature="getText()"/>
  </class>
  <class java-name="javax.swing.JScrollPane" ocaml-name="jScrollPane">
    <constructor signature="(java.awt.Component)"/>
  </class>
  <class java-name="java.awt.Component" ocaml-name="jComponent"/>
  <class java-name="java.awt.Container" ocaml-name="jContainer">
    <method signature="add(java.lang.String,java.awt.Component)"/>
  </class>
</module>
```

Chapter 6

Generating bindings to a database

This chapter presents the generator allowing to produce bindings to databases through JDBC drivers. The full DTD of input files is given by code sample 8. The associated semantics is given below. Each XML file represents an Objective Caml module and acts as a collection of SQL statements that can be executed to either query or modify the database. There are indeed two kinds of SQL statements: commands are SQL statements that modifiy the database and return no result, while queries are SQL statements that return a result set.

Code sample 8 DTD for XML files.

```
<!ELEMENT dbmodule (meta*,(command|query)*)>
<!ATTLIST dbmodule name CDATA #REQUIRED>

<!ELEMENT meta EMPTY>
<!ATTLIST meta name CDATA #REQUIRED>
<!ATTLIST meta value CDATA #REQUIRED>

<!ELEMENT command EMPTY>
<!ATTLIST command name CDATA #REQUIRED>
<!ATTLIST command code CDATA #REQUIRED>
<!ATTLIST command prepare (yes|no) "yes">

<!ELEMENT query EMPTY>
<!ATTLIST query name CDATA #REQUIRED>
<!ATTLIST query code CDATA #REQUIRED>
<!ATTLIST query prepare (yes|no) "yes">
<!ATTLIST query updatable (yes|no) "no">
```

6.1 Meta elements

Meta tags are key-value pairs that give information on how the XML file should be interpreted. In the current version, the following meta keys are recognized: `CLASSPATH`, `DRIVER`, `URL`, `USER`, and `PASSWORD`.

The `CLASSPATH` meta has the same meaning than in the classes generator. It allows to specify the classpath to be used when trying to connect to the database. It is hence used mainly to ensure that the JDBC driver to be used can be reached by Nickel. The `DRIVER` meta is used to give the fully-qualified classname of the JDBC driver.

The `URL`, `USER`, and `PASSWORD` meta elements are used to set respectively the database URL, as well as the login/password to be used for database connection. At the opposite of the `CLASSPATH` and `DRIVER` elements, the `URL`, `USER`, and `PASSSWORD` elements cannot be defined more than once. Finally, the `USER` and `PASSWORD` elements are optional as there could be no authentification needed for a given database.

It is important to keep in mind that all these meta elements are used at *compile-time* and are **not** recorded in any way to be used at *run-time*. It is hence regular to use different urls or users to connect at these different times. However, the database should obviously share the same schema to ensure that metadata gathered at *compile-time* is coherent with the *run-time* environment.

6.2 Prepared and unprepared statements

Both commands and queries accept an attribute `prepare` indicating whether the statement can be used in *prepared* mode. The *prepared* mode allows to leverage the power of JDBC prepared statements. For prepared statements to be used, it is necessary to define a connection to the database. To this end, each generated module will provide the three following functions:

- `connect : string -> string option -> string option -> unit` connects to the database, the parameters being database url and login/password (both being optional)
- `disconnect : unit -> unit` disconnect from the database
- `is_connected : unit -> bool` tests whether the connection to the database has been established

These functions define the connection to be used by the *prepared* mode for the given module. This means that multiple database Nickel-generated modules may coexist happily in the same application but will not share connection information.

The functions and classes generated to execute a *prepared* statement all accept as their first parameter a `CadmiumJDBC.Connection.t option` value that defines how to connect to the database. If `None` is passed, the module connection depicted above is used. Otherwise, the specified connection is used. This scheme allows to leverage the power of prepared statement while still allowing to use a specific connection for the same statement.

The functions and classes generated to execute an *unprepared* statement accept a `CadmiumJDBC.Connection.t` value, which means that they cannot take advantage of the connection handled at the module level.

6.3 Type mapping

The first parameter accepted by generated functions and classes has been defined in the previous section. The other parameters are determined by the actual SQL statement to be executed. As defined by the JDBC standard, the SQL statement can contain some `?` characters that are used

as parameter placeholders. For each of these question marks, the user will have to provide a value at runtime in order to actually execute the statement. As a consequence, each question mark will result in an additional parameter to the function that will be generated for the execution of the statement. If the parameter can be a SQL *null* value then it will be represented by an Objective Caml `option` value. Table 6.1 gives the mapping from SQL types to Objective Caml types.

SQL type	**Nickel-mapped Objective Caml type**
ARRAY	Cadmium.java_object
BIGINT	int64
BINARY	string
BIT	bool
BLOB	CadmiumJDBC.Blob.t
BOOLEAN	bool
CHAR	string
CLOB	CadmiumJDBC.Clob.t
DATALINK	string
DATE	CadmiumJDBC.Date.t
DECIMAL	CadmiumMath.BigDecimal.t
DOUBLE	float
FLOAT	float
INTEGER	int32
LONGNVARCHAR	string
LONGVARBINARY	string
LONGVARCHAR	string
NCHAR	string
NCLOB	CadmiumJDBC.NClob.t
NUMERIC	CadmiumMath.BigDecimal.t
NVARCHAR	string
REAL	float
REF	CadmiumJDBC.Ref.t
ROWID	CadmiumJDBC.RowId.t
SMALLINT	int
SQLXML	CadmiumJDBC.SQLXML.t
STRUCT	Cadmium.java_object
TIME	CadmiumJDBC.Time.t
TIMESTAMP	CadmiumJDBC.Timestamp.t
TINYINT	int
VARBINARY	string
VARCHAR	string

Table 6.1: Mapping of SQL types.

6.4 Command declarations

Command tags accept three attributes: `name`, `code`, and `prepare`. The `name` attribute gives the name of the function to be generated while the `code` attribute contains the actual SQL code to be executed. Finally, `prepare` can be either `yes` or `no` to indicate whether the command may be used as a prepared statement. The functions generated for commands have a return type equal to `int32`. The returned value will indicate the number of rows impacted by the command execution.

6.5 Query declarations

Query tags accept four attributes: `name`, `code`, `prepare`, and `updatable`. The three first attributes have the very same meaning as for command tags. The functions generated for queries have a return type equal to `CadmiumJDBC.ResultSet.t`. Additionally, for each generated function, a class with the same name and parameters is also defined. This class defines an *iterator* that can be used to iterate over the values of the result set. The following methods are provided:

- `next -> t` where `t` is a tuple type corresponding to the columns of the result set (each *nullable* column being translated into an `option` type), raises `Not_found` when all tuples have already been returned
- `close : unit` closes the underlying result set

The `updatable` attribute indicates whether the aforementioned class should also provide methods allowing to modify the current row of the result set. When created, the iterator is positioned before its first element, which means that `next` should be called before any update could be made.

The `CadmiumJDBC.Iterators` module contains functions designed to work with the generated classes: various flavours of map, iter, and fold are provided. It is also possible to convert an iterator instance into a bare list.

6.6 Complete example

This section shows how to use Nickel in conjunction with Cadmium/Cafesterol to write in Objective Caml a small JDBC-based application. This application will iterate over the elements of a given table, and modify some of them. The example is based on Derby[1].

Code sample 9 shows the makefile used by this example when using Cadmium as an interpreter. The `startup` target starts the database and creates a table with some values. The contents of both `start-database.sh` and `execute-database.sh` are specific to the database actually used; the first one should launch the database service while the second one should only log to this service in order to evaluate the passed SQL script (which is represented by code sample 10). At the other end, the `shutdown` target first drops the table contents (by executing the `drop.sql` script reproduced by code sample 11), and then stops the database service by executing the `shutdown-database.sh` shell script.

[1] Java database, available at http://db.apache.org/derby/ (tested with version 10.4.1.3).

Besides these enclosing targets, the `prepare` target first invokes Nickel to generate C, Java, and Objective Caml files and then produces the Objective Caml module interface from its implementation source. The `compile` target first compiles module interface and implementation, then compiles the generated C and Java files, finally the complete Objective Caml program is compiled and linked. The `run` target runs the compiled Objective Caml under Cadmium (if not run under Cadmium, the program fails upon an Objective Caml exception exhibiting that Cadmium is missing and required).

Code sample 12 shows the makefile used by this example when using Cadmium with the Cafesterol compiler (*i.e.* `ocamljava`). The `prepare` target first invokes Nickel to generates C, Java and Objective Caml files and then produces the Objective Caml module interface from its source; finally the generated Java file is compiled. The `compile` target first compiles module interface and implementation, then compiles the main module and links both modules. The `run` target runs the compiled Objective Caml program using compiled libraries. The other targets are identical to their counterpart from the Cadmium-only makefile.

Code sample 7 shows the XML file used to generate C, Java and Objective Caml files. The `<dbmodule` tag first defines the name of the module to generate. Then `<meta` tags are used to specify the classpath, the JDBC driver, and the database URL to be used at module generation time. Finally, three SQL statements are defined:

- a command named `insert` that takes fives parameters and inserts the associated values into the `PEOPLE` table;
- a query named `get_people` that takes no parameter and will iterate over the whole table;
- a query named `get_people_with_mail` that takes no parameter and will iterate over the table element whose `MAIL` column is not `NULL`.

All three statements are prepared, in order to be able to call them without explicitly providing a connection instance. The first of the two queries is also made updatable to be able to modify the table contents while iterating over it.

Code sample 14 shows the Objective Caml source of the program. The `driver` and `url` global variables respectively define the JDBC driver to be used and the URL of the database to log to. The `print_people` uses the `iter_close` function of the `CadmiumJDBC.Iterators` module to iterate over the passed object and closes the result set associated with the passed object. It is important to notice that the tuples that are iterated over have their type determined by the database schema. In this case the tuple is composed of five elements: three `string`s, an `int32 option`, and a `string option`. The first three elements are *simple types* while the last two are *option types* because the first three elements of the `PEOPLE` table are declared `NOT NULL` while the others are not (*cf.* code sample 10 for the details about the table declaration).

The main code first ensures that the driver is loaded (by calling `Cadmium.Class.for_name`), and creates the connection for statements execution (by calling `Database.connect`). Then `get_people` and `get_people_with_mail` classes from the `Database` module are executed to get objects instance to iterate over; `None` is passed for the creation of both objects in order to use the implicit connection set by the call to `Database.connect` function. After that, a new `get_people` instance is created to iterate over the table and change the `NULL` values of the `MAIL` column to some dummy values. Finally, a new element is added to the table through the `Database.insert`

Code sample 9 Makefile for example.

```
JAVA=java
JAVAC=javac
OCAMLC=ocamlc -I +cadmium
CADMIUM_JAR=../../../Cadmium/deploy/ocamlrun.jar
CADMIUM_MAIN=fr.x9c.cadmium.Main
NICKEL_JAR=../../deploy/ocamlwrap.jar
DATABASE_JAR=path-to-derby/derbyclient.jar


all: clean startup prepare compile run shutdown

startup:
        ./start-database.sh
        ./execute-database.sh create.sql

prepare:
        mkdir pack
        $(JAVA) -jar $(NICKEL_JAR) --java-dir=pack --module-kind=database source.xml
        $(OCAMLC) -i database.ml > database.mli

compile:
        $(OCAMLC) -c database.mli
        $(OCAMLC) -c database.ml
        $(OCAMLC) -ccopt -I -ccopt /usr/local/lib/ocaml/caml -c database.c
        $(JAVAC) -target 1.6 -cp $(CADMIUM_JAR) pack/Database.java
        $(OCAMLC) -o prog -custom cadmiumLibrary.cma database.o database.cmo source.ml

run:
        $(JAVA) -cp .:$(CADMIUM_JAR):$(DATABASE_JAR) $(CADMIUM_MAIN) --providers=pack.Databa

shutdown:
        ./execute-database.sh drop.sql
        ./shutdown-database.sh

clean:
        rm -f database.c database.ml database.mli pack/Database.java
        rm -f *.cm* *.o prog pack/*.class
        rm -f *.log
        rm -fr nickel
        rm -fr pack
```

Code sample 10 Creation of database table (`create.sql` file).

```
CONNECT 'jdbc:derby://localhost:1527/nickel;create=true';

CREATE TABLE PEOPLE (
  ID VARCHAR(32) NOT NULL PRIMARY KEY,
  FIRST_NAME VARCHAR(32) NOT NULL,
  LAST_NAME VARCHAR(32) NOT NULL,
  AGE INTEGER,
  MAIL VARCHAR(64)
);

INSERT INTO PEOPLE VALUES
  ('JS', 'John', 'Smith', 35, NULL),
  ('JoD', 'John', 'Doe', NULL, NULL),
  ('JaD', 'Jane', 'Doe', NULL, NULL),
  ('AS', 'Alan', 'Smithee', 40, 'alan.smithee@movies.org');

exit;
```

Code sample 11 Destruction of database table (`drop.sql` file).

```
CONNECT 'jdbc:derby://localhost:1527/nickel';

DROP TABLE PEOPLE;

exit;
```

Code sample 12 Makefile.cafesterol for example.

```
JAVA=java
JAVAC=javac
OCAMLJAVA=ocamljava -I +cadmium -java-package fr.x9c.nickel.dbexample
CADMIUM_JAR=../../../Cadmium/deploy/ocamlrun.jar
NICKEL_JAR=../../deploy/ocamlwrap.jar
DATABASE_JAR=path-to-derby/derbyclient.jar


all: clean startup prepare compile run shutdown

startup:
        ./start-database.sh
        ./execute-database.sh create.sql

prepare:
        mkdir pack
        $(JAVA) -jar $(NICKEL_JAR) --java-dir=pack --module-kind=database source.xml
        $(OCAMLJAVA) -i database.ml > database.mli
        $(JAVAC) -target 1.6 -cp $(CADMIUM_JAR) pack/Database.java

compile:
        $(OCAMLJAVA) -c database.mli
        $(OCAMLJAVA) -classpath . -provider pack.Database -c database.ml
        $(OCAMLJAVA) -o prog.jar -standalone -classpath . -provider pack.Database cadmiumLi

run:
        $(JAVA) -jar prog.jar

shutdown:
        ./execute-database.sh drop.sql
        ./shutdown-database.sh

clean:
        rm -f database.c database.ml database.mli pack/Database.java
        rm -f *.cm* *.j* prog.jar pack/*.class
        rm -f *.log
        rm -fr nickel
        rm -fr pack
```

Code sample 13 Nickel file for example.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!DOCTYPE dbmodule SYSTEM "dtds/dbmodule.dtd">

<dbmodule name="Database">

  <meta name="CLASSPATH" value="path-to-derby/derbyclient.jar"/>
  <meta name="DRIVER" value="org.apache.derby.jdbc.ClientDriver"/>
  <meta name="URL" value="jdbc:derby://localhost:1527/nickel"/>

  <command name="insert"
           code="INSERT INTO PEOPLE VALUES (?, ?, ?, ?, ?)"
           prepare="yes"/>

  <query name="get_people"
         code="SELECT * FROM PEOPLE"
         prepare="yes"
         updatable="yes"/>

  <query name="get_people_with_mail"
         code="SELECT * FROM PEOPLE WHERE MAIL IS NOT NULL"
         prepare="yes"
         updatable="no"/>

</dbmodule>
```

function. Before returning, the main code does some cleanup by calling `Database.disconnect` in order to close the implicit connection of the `Database` module.

Code sample 14 Objective Caml source for example.

```
let driver = "org.apache.derby.jdbc.ClientDriver"

let url = "jdbc:derby://localhost:1527/nickel"

type people = < next : string * string * string * int32 option * string option;
                close : unit >

let print_people (x : people) =
  CadmiumJDBC.Iterators.iter_close
    (fun (id, fname, lname, age, mail) ->
      Printf.printf "  %s -> %s %s (%s) %s\n"
        id
        fname
        lname
        (match age with Some x -> Int32.to_string x | None -> "-")
        (match mail with Some x -> x | None -> "-"))
    x

let () =
  ignore (Cadmium.Class.for_name driver);
  Database.connect url None None;
  print_endline "People:";
  print_people ((new Database.get_people None) :> people);
  print_endline "People with mail:";
  print_people ((new Database.get_people_with_mail None) :> people);
  let rs = new Database.get_people None in
  (try
    while true do
      match rs#next with
      | (_, fname, lname, _, None) ->
          rs#update_5 (Some (Printf.sprintf "%s.%s@unknown.org" fname lname))
      | _ -> ()
    done
  with Not_found -> ());
  ignore (Database.insert None (Some "BD") (Some "Baby") (Some "Doe") (Some 1l) None);
  print_endline "People (updated):";
  print_people ((new Database.get_people None) :> people);
  Database.disconnect ()
```