Dynamic Caml v. 0.2
Run-Time Code Generation Library for Objective Caml

Dmitry Lomov, Anton Moscal
{dsl,msk}@tepkom.ru
St.-Petersburg State University
Faculty of Mathematics and Mechanics
Software Engineering Department

May 22, 2002

Contents

Chapter 1

Overview

Feathers, jewels
Work in crewel —
No expences spared.
— Anon., *"Mexican Legend"*

Dynamic Caml is a library for Objective Caml that provides *dynamic* (or *run-time*) code generation capabilities for Objective Caml programmer.

1.1 Features

Main features of Dynamic Caml (some propaganda `:-)`):

- it is *high-level*. You do not need to care about underlying machine architecture.
- it is intended to be *portable*. The library has the same interface for all supported platforms. To port your program to a different platform, you just link in version of a library for that platform.
- it is *type-safe*. All power of Objective Caml static type checking and type inference is in your hands when you use the library.
- it is *efficient*.
- it is *easy-to-use*, thanks to syntax extensions.
- it is a *library*. Dynamic Caml is *not* a new language — it is a set of modules and syntax extensions. All you need to use the library is to have Objective Caml and CamlP4 installed on your system. You can easily add dynamic code generation to your existing OCaml projects.
- it is *complete and consistent*.

1.2 What is implemented in current version

Current version is a very first alpha-release of the library. The only platform supported now is OCaml byte-code (so it runs everywhere and nowhere `:-)`).

The library now supports a consistent subset of Objective Caml that includes:

- function declaration ("`fun ... -> ...`") and application.
- "`let ... in ...`" constructs.
- "`if ... then ... else ...`" constructs.
- tuples
- full set of logical, integer and floating-point arithmetic operations.
- full set of imperative features.
- arrays and strings

Most notable exceptions from the above list are records, pattern-matching, labels and object-oriented features. All of these may be simulated through function wrappers and syntax extensions. See section 2.2.11 for more details.

This version of the library does not support mutually recursive functions.

1.3 Availability, Authors etc.

The library is developed in FP team of Software Engineering Department of the Faculty of Mathematics and Mechanics, St.-Petersburg State University, Russia (`http://oops.tepkom.ru/young`)

The project home page is `http://oops.tepkom.ru/dml`.

The library is distributed under the terms of a *GNU Library Public License*(see Appendix A for legal information).

Authors contacts:

⋆ Dmitri Lomov
e-mail: `dsl@tepkom.ru`
phone: +7-(812)-375-94-15

⋆ Anton Moskal
e-mail: `msk@tepkom.ru`
phone: +7-(812)-428-46-57

Chapter 2

User's Manual

Who or why, or which, or what
Is the Acond of Swat?
— Edward Lear, *"The Acond of Swat"*

2.1 Introduction

2.1.1 What is Dynamic Code Generation

The most widely used example of dynamic code generation (DCG) is Unix `grep` program. This program accepts a regular expression and a set of files, then it prints out those lines in these files that match given regular expression. Matching in `grep` is implemented as follows:

- at program start it parses regular expression given in command line and produces a *code of the function* that checks whether given string matches that regular expression.

- then it applies that function to every line in input file, printing the line if function returns TRUE.

So `grep` *adapts its code to its arguments*, thus achieving very high performance. This is the main idea behind DCG — adapting program's code to given input.

Let us look at some more examples.

Consider a function tabulation program. This program accepts a string with a mathematical notation of function (like "$(x + 1) * 2$"), and then tabulates it.
Conventionally implemented, this program will parse the definition of a function, obtaining some form of parse tree. It will then iterate through values of x, for each value scanning the parse tree and computing value of a function accordingly.
Applying DCG to this program may improve its performance by an order of magnitude. We can parse the definition of the function, then *generate code that computes it*, and then apply this code to each value of x. Thus we abandon scanning parse tree every time exploiting the same idea — adapt program's code to given input.

In general, suppose we have a function of, say, two arguments, $f(x, y), x \in \mathcal{X}, y \in \mathcal{Y}$. If we need for some (unknown at compile time) x_0 to calculate $f(x_0, y)$ for all $y \in \mathcal{Y}$, dynamically generating code for partially applied function $g(y) = f(x_0, y)$ may give us, as we saw in the examples above, significant performance benefits.

But generating code at run-time seems to require low-level assembler programming, which is tedious, error-prone and non-portable.

Dynamic Caml library is targeted to address these issues, providing means to generate code at run-time in simple, safe and portable way.

2.1.2 DCG in Dynamic Caml

This section is a brief introduction into programming with Dynamic Caml. We will write an implementation of a function tabulation program described in section 2.1.1.

Basics

Dynamic Caml consists of two modules, `Rtcg` and `Eval`. Both of those modules operate with abstract polymorphic type `code`, declared in module `Rtcg`:

```
type 'a code
```

The semantics is following: "the value of type `'a code`, having been evaluated, produces a value of type `'a`".

To evaluate code, the following function from `Eval` module can be used:

```
val eval : ('a -> 'b) code -> ('a -> 'b)
```

Note that code of function only can be evaluated.

Values of a type `'a code` can be constructed using code construction primitives declared in module `Rtcg`. We will need the following of them in this section:

```
val quote :  'a -> 'a code
val abstr :  ('a code -> 'b code) -> ('a -> 'b) code
type int_bin_op =
  IOpSUB | IOpADD | IOpMUL | IOpDIV | ...
val int_bin :  int code -> int_bin_op -> int code -> int code
```

Description of these functions follows:

`quote` This is a simplest DCG function. It return a code which evaluates to its argument. Thus `quote 2` returns the code that evaluates to 2, and

```
quote (fun x -> x*x)
```

returns the code that evaluates to `(fun x -> x*x)`

`abstr` This function "raises" given function over `code` type. For example,

```
abstr (fun x -> (quote 2))
```

is the code of a function which returns 2 for any given argument.

`int_bin` This function creates code which evaluates two `int` codes given and applies specified operation to results. Thus

```
abstr(fun x -> int_bin x IOpADD (quote 2))
```

is the code of a function which accepts integer argument `x` and returns `x + 2`.

Now scenario of run-time code generation is clear: construct code of a function using primitives declared in module `Rtcg`, then use `Eval.eval` to produce "true" OCaml function.

Our first DCG program

Let us now implement the function tabulation program.

First, we declare a expression tree:

```
type iexpr =
    Var
|   Const of int
|   Add of iexpr * iexpr
|   Sub of iexpr * iexpr
|   Mul of iexpr * iexpr
|   Div of iexpr * iexpr
```

Now we are ready to generate code:

```
let fexpr e =
  let gen x = function
    Const i -> quote i
  | Var -> x
  | Add(l,r) -> int_bin (gen x l) IOpADD (gen x r)
  | Sub(l,r) -> int_bin (gen x l) IOpSUB (gen x r)
  | Mul(l,r) -> int_bin (gen x l) IOpMUL (gen x r)
  | Div(l,r) -> int_bin (gen x l) IOpDIV (gen x r) in
  eval (abstr (fun x -> gen x e))
```

The type of function `gen` is `int code -> iexpr -> int code`. Its first parameter is the code to obtain the value of the variable, `x`. `gen` returns the code that calculates an expression given as its second parameter.

Now tabulation is simple:

```
let tabulate table expr start start stop =
  let f = fexpr expr in
  for i = start to stop do
     table.(i) <- f i;
  done
```

Extended syntax

Dynamic Caml provides CamlP4 syntax extensions to assist in constructing complex `code` expressions. The basic syntax is "`{<` *`code expression`* `>}`". Syntax for code expressions resembles that of OCaml expressions, with several restrictions and extensions.

This is a definition of `fexpr` using extended syntax, just to give a "taste" of it:

```
let fexpr e =
  let gen x = function
    Var -> x
  | Const i -> {< `i >}
  | Add(l,r) -> {< @(gen x l) + @(gen x r) >}
  | Sub(l,r) -> {< @(gen x l) - @(gen x r) >}
  | Mul(l,r) -> {< @(gen x l) * @(gen x r) >}
```

```
| Div(l,r) -> {< @(gen x l) / @(gen x r) >}in
eval {< fun x -> @(gen x e)  >}
```

Both definitions of fexpr are completely equivalent. Here, "‘" is "quoting character" and "@" is "insertion character", which allows embedding code expressions written as conventional OCaml expressions into `{< ... >}` construct.

2.1.3 Real Life: Benchmarking Results

Code for two examples described in section 2.1.1, function tabulation and sparse matrix multiplication, can be found in `/tests` subdirectory of Dynamic Caml distribution. The following is the results of our benchmarking, with comments.

Benchmarking was performed on AMD K6-2 500MHz with 64M of RAM under Red Hat Linux 6.2 (kernel 2.2.14-5.0) in single-user mode with Objective Caml v.3.0.

Function tabulation

File `tests/tab.dml` in distribution.

Tabulating function with floating point arithmetic shows $2.5 \div 3$ times speed-up of DCG version over conventional version. Speed-up for integer arithmetic is as much as $4.5 \div 5$ times.

The cause of the difference is the part of total time for calculating function value spent in scanning parse tree. As floating-point operations are more time consuming than integer ones, in former case scanning parse tree takes less part in the total time.

Sparse matrix multiplication

File `tests/matrix.dml` in distribution.

In this test 100×100 matrices are multiplied, first of them having no more that 5 non-zero elements in a row. DCG gives about 20 times speed-up versus non-sparse-optimized matrix multiplication.

For optimized sparse matrix multiplication algorithm, DCG speedup begins when generated code is reused 2 times. Speed-up on second-time reuse is about 5%, and on 10th reuse speed-up is about 30%.

2.2 Programming with Dynamic Caml

This section provides a guide for programming in Dynamic Caml. All primitives from `Rtcg` module are described, along with description of syntax extensions that can be used to simplify coding.

See chapter 3 for information on installation of and compiling with Dynamic Caml. Chapter 4 provides a complete reference guide to both set of primitives for code manipulation and syntax extensions.

2.2.1 Introduction

The core of Dynamic Caml is abstract polymorphic data type `code`:

```
type 'a code
```

Its semantics is simple: value of type `'a code`, being evaluated, produces a value of type `'a`.

$\mathcal{NB}$: Objective Caml imposes a restriction on values of polymorphic types: value of a polymorphic type may not be polymorphic. Thus it is impossible to create a `code` of a polymorphic function in Dynamic Caml. More about this in section 2.2.4.

Module `Rtcg` contains a set of functions for creating `code` values. Module `Eval` contains a function `eval`, which evaluates given `code` value. Only functional `code` values can be evaluated.

Implementation of `Rtcg` module is the same across all supported platforms, whereas implementation of `Eval` module differ from platform to platform[1]. Nevertheless, the interface of both these modules stays the same regardless of target platform.

Scenario of dynamic code generation is following: create a `code` value using functions from `Rtcg` module (syntax extensions gives a great aid at this stage); then evaluate this value to obtain dynamically generated function adapted to your programs input.

2.2.2 Syntax extensions

Syntax extensions provide a simple, powerful and reliable way for creating `code` values. Basic syntax is:

{< *code-expression* >}

With syntax extensions, this is a valid OCaml expression of `'a code` type for some `'a`. Syntax for *code-expression*s will be described below in incremental manner.

In what follows, term "*code expression*" means something that may appear inside {< ... >}; term "*OCaml expression of `code` type*" means conventional OCaml expression of type `'a code` for some `'a`.

Now we shall proceed to details of run-time code generation of various OCaml constructs.

2.2.3 Quoting

Quoting "raises" arbitrary OCaml value over `code` domain; that is, quoting any value of type `'a` results in a value of type `'a code`, which, being evaluated, produces the source value of type `'a`.

Function `quote` is used for quoting:

```
val quote : 'a -> 'a code
```

For example,

```
quote (Printf.printf ''%d'')
```

is the quoted partially applied `printf` that can be used to print integers, as in

```
apply (quote (Printf.printf ''%d'') 27)
```

Above example is a code that prints 27 when evaluated.

[1]As of version 0.2, only one target platform is supported: OCaml byte-code. However, the statement is intended to be true in the future

Syntax extensions

A back-quote, “`” can be used in code expressions to quote particular OCaml expression. Thus,

```
{< `expr >}
```

is completely equivalent to

```
quote expr
```

Moreover, integer, floating point, string and unit (namely “`()`”) constants are quoted automatically, without the need of “`”, so `{< 2 >}` is equivalent to `` {< `2 >} `` and `` {< `''string'' >} `` to `{< ''string'' >}`.

“`@`” in code expressions can used to embed OCaml expressions of `code` type into code expressions. For example, result of

```
let i1 = {< 27 >}in
{< fun x -> x + @i1 >}
```

is a code of function that adds 27 to its argument.

2.2.4 Functions

Abstraction

Non-recursive function code can be created using `abstr` function:

```
val abstr : ('a code -> 'b code) -> ('a -> 'b) code
```

For example,

```
abstr (fun x -> x)
```

is the code of the identity function. Note the fundamental difference with

```
quote (fun x -> x)
```

Latter expression just quotes identity function already created at compile time; the former code is the actual *caml-code* of identity function, which will be created when it is evaluated.

Arguquoting

An important matter is semantics of function argument `x`. It is a *caml-code* of a function argument, that is, a code which is needed to obtain function argument when function is applied. Thus the following OCaml program is meaningless:

```
let f = eval (abstr (fun x -> quote x)) 27 in
let g = eval (abstr (fun y -> f)) 15 in ...
```

This expressions will pass type checks, because `f` has a type `int code`, and `g` will have the same type. But what does `f` return? As it quotes its argument it returns a code that obtains its own argument. This code is meaningless outside of `f` body, as in `g`. Evaluating code in the expression for `g` will result in an exception at run-time. Such problem is called *arguquoting*. Arguquoting occurs whenever an argument of “abstracted” function occurs inside a quotation in a function body and lifetime of that quotation extends function application. For example,

```
let f = eval (abstr (fun x ->
        quote (int_bin x IOpADD (quote 1))) in ...
```

will also cause arguquoting.

Unfortunately, type check fails to discover arguquoting. However, the chances of accidental arguquote are small; code generator can detect arguquoting, and this bug should turn out very fast if it ever occur in your program.

Limitation: no polymorphic functions

OCaml imposes one restriction on values of polymorphic type: these values itself cannot be polymorphic. This imposes a sever restriction on dynamically generated functions: they, in their turn, cannot be polymorphic too.

For example, in:

```
let id_code = abstr (fun x -> x) in
let id = eval id_code in
printf ''%d %s'' (id 27) (id ''27'')
```

the third argument to `printf` causes compilation error.

By all appearance, `id_code` should have type `('a -> 'a) code`, so `id` should have type `'a -> 'a`. However, when OCaml compiler encounters `id 27` it deduces that, as `id_code` is a value of polymorphic type and can not be polymorphic, that `id_code` has type `(int -> int) code`, thus `id` has type `int -> int`, and all goes wrong.

Nevertheless, the need for dynamically generated polymorphic function seldom arises in practice[2].

Recursive functions

Use `fix` function to create a recursive function:

```
val fix : (('a -> 'b) code -> ('a -> 'b) code) ->
                        ('a -> 'b) code
```

This is a familiar fix-point operator. For example,

```
fix (fun f -> abstr(fun x -> apply f x))
```

is a code of ever-looping function (not really useful). Look at a code of ever-popular factorial:

```
let fact =
  eval (fix (fun f ->
    (abstr (fun x ->
      (if' (eq x (quote 1)) (quote 1)
        (int_bin x IOpMUL (apply f
            (int_bin x IOpSUB (quote 1)))))))))
```

Of course, syntax extensions provide much better way to express function definitions:

```
let fact =
  eval {< fix f -> fun x ->
    if x == 1
    then 1
    else x * (f (x-1)) >}
```

Mutually recursive functions are not supported in current version of Dynamic Caml.

[2] Code generated for polymorphic functions is really polymorphic; the code presented above does work but for restriction imposed in OCaml compiler to the values of polymorphic types. So, when absolutely necessary, type checker can be bypassed with help of `Obj.magic`:

```
let id_code = abstr (fun x -> x) in let id : 'a -> 'a = Obj.magic (eval
id_code) in printf ''%d %s'' (id 27) (id ''27'')
```

However, using such "dirty trick" is strongly discouraged. Improper use of `Obj.magic` may lead to disastrous results.

Application

Use `apply` function to create code of an application:

```
val apply : ('a -> 'b) code -> 'a code -> 'b code
```

This function constructs a code which applies functions resulting from first argument evaluation to the result of second argument evaluation.

Example:

```
apply (quote print_int) 27)
```

— code that prints "27".

Syntax extensions

Application has the same syntax as in OCaml: just a space:

```
{< `(Printf.printf ''%d'') 27 >}
```

Abstractions can be created using familiar `fun` construct:

```
{< fun x -> x >}
```

is a code of identity function.

More than one argument can appear after <u>`fun`</u>:

```
{< fun x y -> ... >}
```

is the same as

```
{< fun x -> fun y -> ... >}
```

which, in turn, is the same as

```
abstr( fun x -> (abstr ( fun y -> ...)))
```

Arguments after <u>`fun`</u> can be tuple or universal patterns ("_"). See section 2.2.6 for more details.

Syntactic sugar in form of `let`-constructs is also supported; see section 2.2.5 for more details.

Fixpoint operator can be expressed with a construct similar to `fun` — a `fix` construct. Ever looping function is either:

```
{< fix f -> fun x -> f x >}
```

or

```
{< fix f x -> f x >}
```

`let rec` syntax is also supported, see section 2.2.5.

2.2.5 `let` constructs

`let`-constructs can be created using function `let'`:

```
val let' : 'a code -> ('a code -> 'b code) -> 'b code
```

Thus, a traditional way of expressing `let` as a function is employed. For example:

```
let' (apply (quote read_int) `())
   (fun i -> apply (quote print_int) i)
```

Note that in code generation time functional value for the second argument of `let'` is *not* created.

Syntax extensions

Conventional syntax for <u>`let`</u> `...` <u>`in`</u> `...` is supported:

```
{< let i = `read_int () in
   `print_int i >}
```

is equivalent to the above example. Functions can be declared in conventional way, too:

```
{< let f x y = x + y in
  f 2 >}
```

is the same as:

```
{< let f = fun x y -> x + y in
  f 2 >}
```

Recursive functions may be declared using familiar `let rec` constructs:

```
{< let rec fact x =
  if x == 1 then 1
  else x*(fact (x-1)) in
  fact >}
```

This is a substitute for `fix` definition of `fact` given in section 2.2.4.

Tuples or universal patterns can be used in place of identifiers after `let` (though not in place of function identifiers). See section 2.2.6 for more details.

`let ... and ... in ...` syntax is not supported.

2.2.6 Tuples and patterns

It is impossible in OCaml type system to create general type safe code creation primitives for tuples, as there is no type to which tuples of all sizes belong. Module `Rtcg` provides unsafe functions for tuples, that make heavy usage of `Obj.magic`, and syntax extensions provide type safe interface to them via pattern-matching (see below). If you really need to use those functions directly, refer to ?.

However, module `Rtcg` provides full set of primitives for tuples of length 2 and 3. For example, for tuples of length 2:

```
val tuple2 : 'a code -> 'b code -> ('a * 'b) code
val slice20 : ('a * 'b) code -> 'a code
val slice21 : ('a * 'b) code -> 'b code
```

First function creates a code that creates tuple from to values, and other functions take components of tuples.

Functions for tuples of length 3 are analogous.

Syntax extensions

Syntax extensions employ pattern-matching to provide simple and type-safe access to tuples.

As for now, only tuple patterns and universal patterns are allowed. Patterns can appear as formal parameters to functions, in both `fun` and `let` constructs, similiary to conventional OCaml syntax.

Examples:

```
{< fun (x,_) -> x >}
{< let (s,n) = `frexp 0.5 in ... >}
{< let fst (x,_) = x in ... >}
```

2.2.7 Conditionals

Module `Rtcg` provides an `if'` function:

```
val if' : bool code -> 'a code -> 'a code -> 'a code
```

This function creates a code that evaluates its first argument (condition) and depending on whether it is `true` or `false`, evaluates its second or third argument. For example:

```
if' (apply (quote (=)) x (quote 1)) (quote 2) (quote 3)
```

Physical equality (similar to OCaml `==` and `!=`) can be checked with functions `eq` and `neq`:

```
val eq : 'a code -> 'a code -> bool code
val neq : 'a code -> 'a code -> bool code
```

and structural equality (similar to OCaml `=`) — with function `equal`:

```
val equal : 'a code -> 'a code -> bool code
```

General comparison on values is supported by function `compare`:

```
type comparison = LT | GT | LE | GE | EQ | NEQ
val compare : 'a code -> comparison -> 'a code -> bool code
```

Functions `bor`, `band` and `bnot` are for boolean "and", "or" and "not" operations. For example, `bor` prototype:

```
val bor : bool code -> bool code -> bool code
```

Syntax extensions

Conventional `if ... then ... else` can be used in code expressions:

```
{< if x == 1 then 2 else 3 >}
```

`=`, `==`, `!=`, `&&`, || and `not` can be used in the same manner as in conventional OCaml expressions.

2.2.8 Arithmetic operations

Full set of integer arithmetic operations is provided by `int_bin`, `int_un` and `int_cmp` functions (binary, unary and comparison operation, respectively). Floating-point operations are provided by `float_bin` and `float_neg` functions. See section ? for details.

Syntax extensions

All the familiar operations (`+`, `-`, `+.`, `-.` etc.) can be used in code expressions.

2.2.9 Arrays and strings

Functions `array_set` and `array_get` can be used to get or set an element of an array:

```
val array_get : 'a array code -> int code -> 'a code
val array_set : 'a array code -> int code -> 'a code
      unit code
```

Function `array_length` obtains array length:

```
val array_length : 'a array code -> int code
```

Similiar functions `string_get`, `string_set` and `string_length` do the same for strings.

Bounds checking

Bounds checking in array and string operations can be controlled using function `set_unsafe` in `Eval` module:

```
val set_unsafe : bool -> unit
```

Effect of `set_unsafe true` is similiar to OCaml compiler `-unsafe` parameter.

Default value is `false`, i.e. bound checking is performed.

Syntax extensions

Syntax for array and string operations in code expressions is the same as in conventional OCaml expressions: familiar `.[]` and `.()` operations can be used.

2.2.10 Imperative features

Function `seq` provides sequential execuction of the code:

```
val seq : unit code -> 'a code -> 'a code
```

This function constructs a code which evaluates its first argument, then its second argument, and the result is a value of the second argument. To sequence code whcih return some value, use function `lose`:

```
val lose : 'a code -> unit code
```

Examples:

```
seq (apply (quote print_int) x) x
seq (lose (quote 2)) x
```

The first is a code that prints x, then returns it. The second is meaningless — it is just an example of `lose` function.

Use function `for'` to create a code of a `for`-loop :

```
type direction = DTo | DDownTo
val for' : int code -> direction -> int code ->
  (int code -> unit code) -> unit code
```

For example:

```
for' (quote 1) DTo (quote 10)
    (fun i -> (apply (quote print_int) i))
```

is a code that prints numbers from 1 to 10. Function `while'` creates a `while`-loop:

```
val while' : bool code -> unit code -> unit code
```

References are also fully supported.

Syntax extensions

Again, everything is the same as in conventional OCaml expressions.

"`;`" is a sequencing operator:

```
{< `print_int x; x >}
```

Note, however, that if you sequence non-`unit` code, you cannot sequence it directly: the following will not typechecks:

```
{< 2; x >}
```

because it is equivalent to

```
seq (quote 2) x
```

and `quote 2` has type `int code`, not `unit code`. Use something like:

```
{< @(lose {< quote 2 >}); x >}
```

Loops has the same syntax as in OCaml expressions:

```
{< for i = 1 to 10 do
     `print_int i
   done >}
```

2.2.11 Functional wrappers

Some constructs of Objective Caml cannot be supported "natively" by Dynamic Caml. These are

- Record creation and access
- Creation of abstract data type values
- Pattern matching

Those constructs can be used in dynamically generated code indirectly, via "functional wrappers". In the following example, functional wrapper is used to access a field of a parameter of function:

```
type point = { px : int; py : int }
...
let f = eval (abstr (fun p ->
  int_bin
    (apply (quote (fun p -> p.px)) p) IOpADD (quote 1)))
```

This can be rewritten with syntax extensions as follows:

```
let f = eval {< fun p -> (`(fun p -> p.px) p) + 1 >}
```

which still looks pretty ugly, however.

Good new are that functional wrappers are automatically created by Dynamic Caml syntax extensions. So the above can be written as follows:

```
let f = eval {< fun p -> p.px + 1 >}
```

which is probably as good as you can get.

Functional wrappers are automatically generated for record creation, abstract data type values creation and pattern-matching constructs.

Dynamic Caml code generator inlines all functional wrappers described above (except wrappers for pattern-matching), so additional applications does not incur any overhead on performance of dynamically generated code.

2.2.12 Unsupported constructs

The following constructs are unsupported by Dynamic Caml at the moment:

- Object-oriented constructs (`new` and method calls)
- Labelled applications
- Variant types

Those contructs can still be used in Dynamic Caml programs via functional wrappers, but those wrappers are not automatically generated by syntax extensions.

Chapter 3

Technical Guide

They sought it with thimbles, they sought it with care,
They pursued it with forks and hope,
They threatened its life with a railway share,
They charmed it with smiles and soap.
— Lewis Caroll, *"The Hunting of the Snark"*

3.1 Installation

3.1.1 Prerequisites

To install Dynamic Caml library you should have installed on your computer:

- Objective Caml bytecode compiler.
- *(optional but highly recommended)* CamlP4.
- ANSI C compiler, preferably GNU C.
- GNU make.
- POSIX-complaint awk.

Library works with version 3.04 of Objective Caml system.

Library have been tested on Linux and on Windows with Cygwin port of OCaml.

3.1.2 Building the Library

Un-gzip the distribution, then proceed with the installation as follows:

1. Run `./configure`. Parameters to `configure` may be:

   `-oc` *`filename`*
   Filename of OCaml bytecode compiler. Default is from `PATH`.

   `-p4` *`filename`*
   Filename of CamlP4. Default is from `PATH`.

`-no-syntax`
Do not compile syntax extensions

`-cc` *`filename`*
Filename of C compiler. Default is `cc` from `PATH`.

`-lib` *`dir`*
Directory of OCaml runtime library. Default is extracted from information given by bytecode OCaml compiler with `-v` switch. Dynamic Caml installs to this directory.

`-p4lib` *`dir`*
Directory of CamlP4 runtime library. Default is extracted from information given by `camlp4` with `-where` switch. Dynamic Caml installs to this directory.

`-help`
Information on `configure` parameters.

2. Run `make`. This will build the whole Dynamic Caml library and test it on a set on a test suite.

3. You may want to benchmark your build. To do so, go to `tests` directory and type `make test`.

4. Now you may install Dynamic Caml. Become root if necessary and do `make install`. This will copy library modules to OCaml library dir and syntax extensions — to CamlP4 library dir.

Installation is complete. Enjoy!

3.2 Compiling With Dynamic Caml

3.2.1 Libraries

Every program that uses Dynamic Caml must be linked with `rtcg.cma` library. This library contains implementation of `Rtcg` module and should go *before* any target-dependent library. For different targets different libraries must be linked.

OCaml bytecode. Caml library `rtcgbyte.cma` and C library `librtcg.a` must be linked. Your program must be linked in `-custom` mode. Example:

```
$ ocamlc -o t1 -cclib -lrtcg -custom rtcg.cma rtcgbyte.cma t1.ml
```

3.2.2 Syntax extensions

OCaml programs that use extended syntax must be passed through `camlp4` before compiling. Syntax extension module is called `dml.cmo`.

The following command may be used to compile Dynamic Caml program with syntax extensions:

```
$ ocamlc -c -o t2.cmo \
> -pp "camlp4 pa_o.cmo dml.cmo pr_dump.cmo -impl" \
> -impl t2.dml
```

If you do not need other syntax extensions in your files, you may use syntax extensions files bundlend in one library:

```
$ ocamlc -c -o t2.cmo -pp ‘‘camlp4 dml-syntax.cma -impl’’ \
> -impl t2.dml
```

It is not difficult to adapt the above for makefile entry:

```
.dml.cmo:
     ocamlc -c -o $@ \
        -pp "camlp4 pa_o.cmo dml.cmo pr_dump.cmo -impl" \
        -impl $<
```

We use suffix `.dml` to denote Dynamic Caml programs.

Chapter 4

Reference Manual

`Earth`: Mostly harmless
— Douglas Adams

4.1 Library Reference

4.1.1 Module Rtcg

Module `Rtcg` contains primitives for code creation.

```
type 'a code
```

Basic abstract type. Semantics is the following: value of type `'a code`, being evaluated, produces a value of type `'a`.

Core functions

```
val abstr : ('a code -> 'b code) -> ('a -> 'b) code
```

Function abstraction. "Raises" a function over `code` domain.

```
val apply : ('a -> 'b) code -> 'a code -> 'b code
```

Creates code of function application.

```
val quote : 'a -> 'a code
```

Quotes arbitrary OCaml value in code.

```
val fix : (('a -> 'b) code -> ('a -> 'b) code) ->
  ('a -> 'b) code
```

Fixpoint operator for recursive functions.

```
val let' : 'a code -> ('a code -> 'b code) -> 'b code
```

Functional representation of a `let` construct.

Booleans and conditionals

```
val if' : bool code -> 'a code -> 'a code -> 'a code
```

Equivalent for `if`-statement. Creates code that evaluates either second or third argument depending on whether result of evaluating first argument is `true` or `false`

```
val bor : bool code -> bool code -> bool code
```
Boolean "or".

```
val band : bool code -> bool code -> bool code
```
Boolean "and".

```
val bnot : bool code -> bool code
```
Boolean "not".

All boolean operations have McCarthy semantics.

Equality

```
val eq : 'a code -> 'a code -> bool code
```
Test for physical equality (like OCaml ==).

```
val neq : 'a code -> 'a code -> bool code
```
Test for physical inequality (like OCaml !=).

```
val equal : 'a code -> 'a code -> bool code
```
Test for structural equality (like OCaml =).

```
type comparison = LT | GT | LE | GE | EQ | NEQ
```
Comparison type

```
val compare : 'a code -> comparison -> 'a code -> bool code
```
Structural ordering of values

Tuples

```
val tuple2 : 'a1 code -> 'a2 code -> ('a1*'a2) code
val tuple3 : 'a1 code -> 'a2 code -> 'a3 code ->
  ('a1*'a2*'a3) code
```
Functions for creating tuples of length 2 and 3.

```
val slice20 : ('a*'b) code -> 'a code
val slice21 : ('a*'b) code -> 'b code
val slice30 : ('a*'b*'c) code -> 'a code
val slice31 : ('a*'b*'c) code -> 'b code
val slice32 : ('a*'b*'c) code -> 'c code
```
Accessing tuple elements — tuples of length 2 and 3.

```
val unsafe_tuple : Obj.t -> Obj.t
```
Unsafe function for creating code creating tuples.
An argument must be OCaml memory block containing codes of all elements of a tuple. Result is code creating tuple with that elements.
This function should never be used directly.

```
val unsafe_slice : Obj.t -> int -> Obj.t
```
Unsafe function for creating code for getting elements of tuples. Arguments are tuple code and element to get (first has index 0). Result is a code for obtaining an element of a tuple.
This function should never be used directly.

Imperative

```
val seq : unit code -> 'a code -> 'a code
```
Create `code` that sequence evaluation of two `code`s.

```
val lose : 'a code -> unit code
```
"Loses" result of `code` so that it can be sequenced.

```
type direction = DTo | DDownTo
```
Direction of `for` loop.

```
val for' : int code -> direction -> int code ->
  (int code -> unit code) -> unit code
```
For loop. 3rd argument is applied for values starting from value of first argument, to the value of third argument increasing or decreasing depending on second argument.

```
val while' : bool code -> unit code -> unit code
```
While loop.

Arrays

```
val array_get : 'a array code -> int code -> 'a code
```
Creates `code` to get an element from array

```
val array_set : 'a array code -> int code -> 'a code ->
  unit code
```
Creates `code` to set an element of an array

```
val array_length : 'a array code -> int code
```
Creates `code` that calculates array length

Strings

```
val string_get : string code -> int code -> char code
```
Creates `code` to get a character from a string

```
val string_set : string code -> int code -> char code ->
  unit code
```
Creates `code` to set a character of a string

References

```
val ref_create : 'a code -> 'a ref code
```
Creates `code` of a reference

```
val ref_value : 'a ref code -> 'a code
```
Gets value of reference

```
val ref_set : 'a ref code -> 'a code -> unit code
```
Assign a value to a reference

Integers

```
type int_bin_op =
  IOpSUB | IOpADD | IOpMUL | IOpDIV | IOpMOD |
  IOpAND | IOpOR | IOpXOR |
  IOpLSL | IOpLSR | IOpASR
```

Binary operations with integers

```
type int_un_op = IUnNEG | IUnNOT
```

Unary operations with integers

```
val int_bin : int code -> int_bin_op -> int code -> int code
```

Code of binary integer operation

```
val int_un : int_un_op -> int code -> int code
```

Code of unary integer operation

```
val int_cmp : int code -> comparison -> int code -> bool code
```

Code of integer comparison

Floats

```
type flop =
  FLOP_ADD | FLOP_SUB | FLOP_MUL | FLOP_DIV |
  FLOP_MOD | FLOP_POWER
```

Floating-point operations

```
val float_bin : float code -> flop -> float code -> float code
```

Code of floating-point binary operation

```
val float_neg : float code -> float code
```

Code of floating-point negation

4.1.2 Module Eval

Module `Eval` contains primitives for code evaluation.

```
val eval : ('a -> 'b) code -> ('a -> 'b)
```

Evaluate given code to obtain a function.

```
exception Arguquoting
```

Raised by `eval` if arguquoting occurs

```
val set_unsafe : bool -> unit
```

Set code generation mode to safe or unsafe. In safe mode, no bounds checking for arrays or strings is peformed. This is equivalent to `ocamlc` "`-unsafe`" parameter.

Default mode is safe.

4.2 Syntax Reference

The following are syntax definitions for code expressions.

4.2.1 OCaml expressions

Definition of OCaml expression is extended to include the following:

```
(1) expr :: {< code-expr >}
```

Here, *code-expr* is *code expression* as defined further in this section.

4.2.2 Code expressions

```
(2) code-expr :: constant | value-path
        | ( code-expr )
        | code-expr { , code-expr }+
        | ncconstr code-expr | code-expr :: code-expr
        | [ [ code-expr { , code-expr }* ] ]
        | code-expr code-expr
        | code-expr infix-op(8:24) code-expr
        | prefix-op(7:24) code-expr
        | code-expr . ( code-expr )
        | code-expr . ( code-expr ) <- code-expr
        | code-expr . [ code-expr ]
        | code-expr . [ code-expr ] <- code-expr
        | fun code-patt+ -> code-expr
        | fix identifier -> code-expr
        | match code-expr with
            [ | ]cd-match-case(6:24)
            [ | cd-match-case(6:24) ]*
        | let code-patt+ = code-expr in code-expr
        | let rec identifier code-patt+ = code-expr in
        | if code-expr then code-expr else code-expr
        | code-expr ; code-expr
        | for identifier = code-expr direction code-expr
          do code-expr done
        | while code-expr do code-expr
          do code-expr done
        | ‘ expr | @ expr
(3) direction :: to | downto
(4) simple-patt :: identifier | _
(5) code-patt :: simple-patt | ( simple-patt [ , simple-patt ]+)
(6) cd-match-case :: pattern [ when expr ] -> code-expr
(7) prefix-op :: - | -. | lnot | ! | not
(8) infix-op :: + | - | * | / | +. | -. | *. | /.
        | land | lor | lxor | mod | lsl | lsr | asr
        | || | && | = | <> | == | != | > | < | <= | >= | ::
```

For grammar entities undefined in the above (such as *constant* or *ncconstr*) please refer to “Objective Caml System” documentation.

The precedence and associativity rules for operators and non-closed contructs can also be found there.

Appendix A

Legal Notice

– *Hey look, there rides Joe-No-Catch!*
– *How come they can't catch him?*
– *Well, they can... But what for?*
— Old cowboy joke