A taste of **Carbon**

Luca Padovani

May 2008

Summary

- genesis
- language features
- language implementation
- status

Working mates

- helm.cs.unibo.it
- matita.cs.unibo.it

Bindings: the “meta” solution

- Luca Padovani, Claudio Sacerdoti Coen, Stefano Zacchiroli, “*A Generative Approach to the Implementation of Language Bindings for the Document Object Model*”, at Generative Programming and Component Engineering, 2004.

Calling hell from heaven...

```
value ml_gdome_di_hasFeature(value self, value feature,
                             value version) {
  CAMLparam3(self, feature, version);
  GdomeException exc_;
  GdomeBoolean res_;
  res_ = gdome_di_hasFeature(DOMImplementation_val(self),
    DOMString_val(feature), DOMString_val(version),
    &exc_);
  CAMLreturn(Val_bool(res_));
}
```

- beware of the number of arguments
- beware of dynamic library
- not to mention the intricacies of data representation (**31 bits** integers)

. . . and heaven from the shell

```
LIB_DEPS =
  $(patsubst %, basic/%, $(shell cat $(srcdir)/basic/.linkorder))
  $(patsubst %, core/%, $(shell cat $(srcdir)/core/.linkorder))
  $(patsubst %, events/%, $(shell cat $(srcdir)/events/.linkorder))
  $(shell cat $(srcdir)/.linkorder)
...
if HAVE_OCAMLOPT_COND
install-data-local: $(OPT_INST)
else
install-data-local: $(BYTE_INST)
endif
  $(mkinstalldirs) $(OCAMLINSTALLDIR) $(STUBSDIR)
  for i in $^; do
    if [ "$$i" != "$(DLL)" ]; then
      $(INSTALL_DATA) $$i $(OCAMLINSTALLDIR)/$$i;
    fi
  done
  if [ "x$(OCAMLFIND)" != "x" ]; then
    mv $(OCAMLINSTALLDIR) $(OCAMLINSTALLDIR).saved &&
    $(mkinstalldirs) $(DESTDIR)$(OCAML_LIB_PREFIX)/ &&
    $(OCAMLFIND) install -destdir $(DESTDIR)$(OCAML_LIB_PREFIX)/ $(PKGNAME) META $(DLL) &&
    $(INSTALL_DATA) $(OCAMLINSTALLDIR).saved/* $(OCAMLINSTALLDIR)/ &&
    rm -rf $(OCAMLINSTALLDIR).saved/;
  else
    $(INSTALL_DATA) $(DLL) $(STUBSDIR);
  fi
  rm $(STUBSDIR)/lib$(ARCHIVE).so
...
%.cmo : $(srcdir)/%.ml
  if test ! -e $(@:%.cmo=%.ml) -a "x$(srcdir)" != "x." ; then $(LN_S) $< . ; fi
  $(OCAMLC) -c $(@:%.cmo=%.ml)
```

Teaching “compilers”

A matter of style

- OCaml is sometimes embarrassing

A matter of time

- a compiler project is too much work for a single student
- it would be good to have a clean (but real) compiler to study and modify

Scary stuff

- target code generation
- multiple architectures
- garbage collector

Using the work of others

Code generation

- C−−
- .NET architecture
- JVM
- LLVM
- GCC backend

Runtime support

- Boehm garbage collector (conservative, easy to use, thread support, . . .)

The language

In a nutshell

- functional fragment of OCaml
- imperative features
- type classes (overloading)

Functions and patterns

```
let ack : Int -> Int -> Int
let ack
[ 0 n = n + 1
| m 0 = ack (m - 1) 1
| m n = ack (m - 1) (ack m (n - 1)) ]

let merge : List Int -> List Int -> List Int
let merge
[ [] l = l
| l [] = l
| (a :: l) ((b :: _) as m) when a < b = a :: merge l m
| l (b :: m) = b :: merge l m ]
```

Recursive definitions without 'rec' and 'and'

```
let skip : [a] List a -> List a
let skip
[ [] = []
| (_ :: l) = take l ]

let take : [a] List a -> List a
let take
[ [] = []
| (x :: l) = x :: skip l ]
```

- same syntax for global and local definitions

Operator currying and partial application

```
let prepend_all : [a] a -> List a -> List a
let prepend_all x = List.map (x ::)

let qsort : List Int -> List Int
let qsort
[ [] = []
| (hd :: tl) =
    let l1 = List.filter (< hd) tl
    let l2 = List.filter (>= hd) tl in
      qsort l1 ++ [hd] ++ qsort l2 ]
```

- partial applications on the "right" side

Extensible notation

```
(# Int.cmx #)
notation `~-` _ = neg at 30
notation left _ `+` _ = add at 60
notation left _ `-` _ = sub at 60

(# Prelude.cmx #)
type List a = [ Nil | Cons a (List a) ]

notation right _ `::` _ = Cons at 80
notation `otherwise` = True
```

- symbols as value aliases
- symbols as contructor aliases
- named constants within backticks

Type and value qualifiers

```
(# Lazy.cm #)
private type Content a = ...

abstract type T a = { mutable content : Content a }

(# Array.cmx #)
let private index_out_of_bounds a i =
  i < 0 || i >= length a
```

- no need to separate interface and implementation
- very simple module system
- no functors

Type classes and overloading

```
class Eq a {
  let eq : a -> a -> Bool
}

instance Eq Int {
  let eq : Int -> Int -> Bool
  let eq = Int.eq
}

let mem : [Eq a] a -> (List a) -> Bool
let mem
[ _ [] = False
| x (y :: _) when eq x y = True
| x (_ :: l) = mem x l ]
```

Language implementation

Digesting Carbon

Language	functions as values	algebraic datatypes	pattern matching	implicit typing	type classes
Carbon	✓	✓	✓	✓	✓
core	✓	✓	✓		
object	✓				

Carbon	⇒	core	type inference, overloading resolution
core	⇒	object	function flattening, pattern matching, boxing/unboxing
object	⇒	target	memory allocation, code flattening

List.filter (Carbon)

```
let filter : (a -> Bool) -> List a -> List a
let filter
[ f [] = []
| f (hd :: tl) when f hd = hd :: filter f tl
| f (_ :: tl) = filter f tl ]
```

List.filter (core)

```
let filter : [a].(a -> Bool) -> (List a) -> (List a) =
  fun [a] (f : a -> Bool) (l : (List a)) =
    match l with
      Nil[a]{} when True{} => Nil[a]{}
    | Cons[a]{head = hd : a; tail = tl : (List a)}
        when f hd =>
          Cons[a]{head = hd; tail = filter [a] f tl}
    | Cons[a]{head = hd : a; tail = tl : (List a)}
        when True{} => filter [a] f tl
    end match
  end fun
```

List.filter (object)

```
fun filter<a>(x : record taggedRecordT,
              y : record ctor_Cons<a>)
  : record ctor_Cons<a> =
if ‘VEQ >(y, null ctor_Cons<a>) then
  null ctor_Cons<a>
else
  let u : a = y.ctor_Cons<a> _0
  and v : record ctor_Cons<a> = y.ctor_Cons<a> _1 in
    if (applyF_1(x, (u <: a)) :> bool) then
      init_ctor_Cons<a>(new ctor_Cons<a>, u, filter<a>(x, v))
    else
      filter<a>(x, v)
    end if
  end let
end if
```

Carbon function = native function + dispatcher

```
(# Carbon #)
let add x y = x + y

(# object #)
fun add(a__0 : int, a__1 : int) : int =
  ‘IADD(a__0, a__1)

fun dispatcher_add(acl_add : record taggedRecordT,
                   a__0 : value, a__1 : value) : value =
  (add((a__0 :> int), (a__1 :> int)) <: int)
```

- most applications are to known functions
- most applications are saturated

Compiling algebraic data types

class	example
unitary	`type T = A`
boolean	`type T = A | B`
enumeration	`type T = A | B | C | ...`
record	`type T a b = T Int a b`
stripped	`type T = Age Int`
nullable	`type List a = Nil | Cons a (List a)`
	`type Maybe a = None | Some a`
general	`type T = A Int | B | C Float | ...`

Unitary parameters are useless

```
(# Carbon #)
let f () () () = 3

(# object #)
fun f() : int = 3

fun dispatcher_f(acl_f : record taggedRecordT,
                 a__0 : value, a__1 : value,
                 a__2 : value) : value =
  (f() <: int)
```

Unitary data may disappear

```
(# Carbon #)
let ignore _ = ()

(# object #)
fun ignore<a>(a__3 : a) : void = begin end

fun dispatcher_ignore(acl_ignore : record taggedRecordT,
                      a__3 : value) : value =
  begin
    ignore<value>(a__3);
    (0 <: int)
  end
```

Boolean types

```
(# Carbon #)
type T = [ A | B ]

let f [ A = 0 | B = 1 ]

(# object #)
value g_A : bool = false
value g_B : bool = true

fun f(a__2 : bool) : int =
  if a__2 then 1 else 0 end if

fun dispatcher_f(acl_f : record taggedRecordT,
                 a__2 : value) : value =
  (f((a__2 :> bool)) <: int)
```

Records and fields with native type

```
(# Carbon #)
type R a = {
  a : Int;
  b : Char;
  c : a;
}

(# object #)
public record ctor__R<a> =
  a : int;
  b : char;
  c : a;
```

List.filter and nullable types

```
public record ctor_Cons<a> = _0 : a; _1 : record ctor_Cons<a>;

public fun filter<a>(x : record taggedRecordT,
                     y : record ctor_Cons<a>)
  : record ctor_Cons<a> =
if ‘VEQ<record ctor_Cons<a> >(y, null ctor_Cons<a>) then
  null ctor_Cons<a>
else
  let u : a = y.ctor_Cons<a> _0
  and v : record ctor_Cons<a> = y.ctor_Cons<a> _1 in
    if (applyF_1(x, (u <: a)) :> bool) then
      init_ctor_Cons<a>(new ctor_Cons<a>, u, filter<a>(x, v))
    else
      filter<a>(x, v)
    end if
  end let
end if
```

Binding object functions

```
(# Int.cmx #)
let neg : Int => Int
let add : Int -> Int => Int
let sub : Int -> Int => Int
let mul : Int -> Int => Int
let unsafe_div : Int -> Int => Int
let unsafe_rem : Int -> Int => Int

(# Int.cox #)
fun neg(a : int) : int = `INEG(a)
fun add(a : int, b : int) : int = `IADD(a, b)
fun sub(a : int, b : int) : int = `ISUB(a, b)
fun mul(a : int, b : int) : int = `IMUL(a, b)
fun unsafe_div(a : int, b : int) : int = `IDIV(a, b)
fun unsafe_rem(a : int, b : int) : int = `IREM(a, b)
```

Binding native functions

```
(# Int.cmx #)
let to_string : Int => String

(# Int.cox #)
external to_string : fun (int) : string =
  ["shell", "carbon_int_to_string"],
  ["C", "carbon_int_to_string"]

(# int_native.c #)
CARBON_STRING carbon_int_to_string(CARBON_INT v) {
  CARBON_STRING res;
  ...
  return res;
}
```

Binding native non-functions

```
(# Float.cmx #)
let epsilon : Float
let max_value : Float

(# Float.cox #)
external g_epsilon : float32 =
  ["shell", "carbon_float_epsilon"],
  ["C", "carbon_float_epsilon"]

external g_max_value : float32 =
  ["shell", "carbon_float_max_value"],
  ["C", "carbon_float_max_value"]

(# float_native.c #)
CARBON_FLOAT32 carbon_float_epsilon = G_MINFLOAT;
CARBON_FLOAT32 carbon_float_max_value = G_MAXFLOAT;
```

Binding native types

```
(# Prelude.cmx #)
type Array a

(# Array.cmx #)
let length : [a] Array a => Int

(# Array.cox #)
external length<a> : fun (value) : int

(# array_native.c #)
CARBON_INT carbon_array_length(CarbonArray* a)
{
  return ((CarbonArray*) a)->length;
}
```

Binding native types with finalizers

```
(# Timer.cmx #)
type T
let new : Unit => T

(# Timer.cox #)
external &new : fun () : value

(# timer_native.c #)
static void timer_finalizer(CARBON_VALUE, CARBON_VALUE);

CARBON_VALUE carbon_timer_new() {
  GTimer* timer = g_timer_new();
#ifdef CARBON_HAVE_LIBGC
  GC_REGISTER_FINALIZER(timer, timer_finalizer);
#endif
  return timer; }
```

Accessing Carbon data types

```
(# String.cmx #)
let concat : List String => String

(# String.cox #)
external concat : fun (record ctor_Cons<string>) : string =
  ["shell", "carbon_string_concat"],
  ["C", "carbon_string_concat"]

(# string_native.c #)
CARBON_STRING carbon_string_concat(CARBON_VALUE l) {
  for (CARBON_LIST l1 = (CARBON_LIST) l;
       l1 != NULL;
       l1 = CARBON_LIST_NEXT(l1)) {
    CARBON_STRING s = CARBON_LIST_HEAD(l1);
    ...
  }
```

Linking against Carbon code

live...

Conclusion

- Carbon tastes good

Urgent things to do

- type classes
- fix bugs
- write a yummy app

Then

- more backends (tail recursion)
- bootstrapping Carbon
- coercions
- ...