Relational data types

Pierre Weis

JFLA – January 28 01 2008

The idea

Enhance Caml data type definitions in order to

- handle *invariants* verified by values of a type,
- provide quotient data types, in the sense of m
  quotient structures,
- define automatic computation of canonical rep
  values.

Usual data type definition ki

There are three classical *kinds* of data type definitio

- *sum* type definitions (disjoint union of sets with t mands),
- *product* type definitions (anonymous cartesian pr sian products with named components)
- *abbreviation* type definitions (short hands to na pressions)

Visibility of data type definiti

There are two classical *visibility* of a data type defi

- *concrete* visibility: the implementation of the ty
- *abstract* visibility: the implementation of the typ

Consequence of visibility for progr

For concrete types:

- value inspection is allowed via pattern matching
- value construction is not restricited,

For abstract types:

- value inspection is not possible,
- value construction is carefully ruled.

Consequence of visibility for pro

For concrete types, the representation of values is r

- the compiler can perform type based optimizati
- the debugger (and the toplevel) can show (print

For abstract types, the representation of values is h

- the compiler cannot perform type based optimiz
- the debugger and the toplevel system just prir <abstr>.

Visibility management constr

Modules are used to define visibility of data type de

- the implementation defines the data type as co
- the interface exports the data type as concrete/

The interface exports the data type as concrete if it
data type with its definition (the associated constr
sum type, the labels for a record, or the defining typ
for an abbreviation).

Defining invariants

Usual (concrete) data types implement *free* data st

- sums: free (closed) algebra (the constructors de
  nature of the free algebra),
- products: free cartesian products for records,
- abbreviations: free type expressions.

By *free* we mean the usual mathematical meaning: n
on the construction of values of the set (type), p
signature constraints are fulfilled.

Examples

```
type expression =
   | Int of int
   | Add of expression * expression
   | Opp of expression

type id = {
  firstname : string;
  lastname : string;
  married : bool;
};;

type real = float;;
```

Counter examples

Sum and products:

```
type positive_int = Positive of int;;
```

```
type rat = { numerator : int; denominator : int; }
```

Despite the intended meaning:

- `Positive (-1)` is a valid `positive_int` value,
- `{numerator = 1; denominator = 0;}` is a valid `rat`

Counter examples

Abbreviations:

```
type km = float;;
type mile = float;;
```

Despite the intended meaning:

- `-1.0` is a valid `km` value,
- `((x : km) : mile)` is not an error (a `km` value is a

Non free data types

Many mathematical structures are not free.
(Cf. Generators & relations presentations of mathem
tures.)

Many data structures are not free having various
straints.

The usual feature of programming languages to de
free data structure is to provide abstract visibility a
data types (or ADT).

ADT as Non free data typ

Using an ADT, the constructors, labels, or type exp
onym of the type are no more accessible to build sp
sired values.
Construction of values is restricted to *construction f*
fined in the implementation module of the abstract

Advantage: non free data types invariants are prope
Drawback: inspection of values is no more a buil
Inspection functions should be provided explicitely b
mentation module.

There is no pattern matching facility for ADTs.

Example

```
type positive_int = Positive of int;;
let make_positive_int i =
  if i < 0 then failwith "negative int" else Posit
let int_of_positive_int p = p;;

type rat = { numerator : int; denominator : int; }
let make_rat n d =
  if d = 0 then failwith "null denominator" else
  { numerator = n; denominator = d; };;

let numerator r = r.numerator;;
let denominator r = r.denominator;;
```

Example

```
type km = float;;
let make_km k =
  if k <= 0.0 then failwith "negative distance" el

let float_of_km k = k;;

type mile = float;;
let make_mile m =
  if m <= 0.0 then failwith "negative distance" el
let float_of_mile m = m;;
```

Private visibility

To provide pattern matching for non free data ty
troduced a new visibility for data type definitions:
visibility.

As a concrete data type, a private data type (*PDT*)
ifest implementation. As an abstract data type, a
type limits the construction of values to provided
functions.

In short, private data type are:

- concrete data types that support *invariants* or
  tween their values,

- fully compatible with pattern matching.

Examples

All the quotient sets you need can be implemente
types.

For quotient types the corresponding invariant is:
any element in the private type is the canonical rep
its equivalence class.

Formulas, groups, . . .

Definition of private data ty

As abstract and concrete data types, private data t
plemented using modules:

- inside *implementation* of their defining module, re
types are regular concrete data types,

- in the *interface* of their defining module, private da
simply declared as *private*.

Usage of a private data typ

In client modules:

- a private data type does not provide labels nor
  to build its values,
- a private data type provides labels or constructor
  matching.

Consequences

The module that implements a private data type:

- must export *construction functions* to build the
- has not to provide *destruction functions* to acce
  values.

The pattern matching facility is available for private

Comparison with abstract data

Abstract data types also provide invariants, but:

- once defined, an ADT is *closed*: new functions
  are mere compositions of those provided by the
- once defined, a private data type is *still open*: a
  functions can be defined via pattern matching o
  sentation of values.

Consequences

- the implementation of an ADT is big (it basic
the set of functions available for the type),
- the implementation of a PDT is small (it only
set of functions that provides the invariants),
- proofs can be simpler for PDT (we must only pr
mandatory construction functions indeed enforc
ants).

Consequences

Clients of an ADT have to use the construction and
functions provided with the ADT.

Clients of a PDT must use the construction functi
serve invariants but pattern matching is still freely a

All the functions defined on an PDT *respect* the P
ants (granted for free by the type-checker!)

Relational data types

A *relational* data type (or RDT) is a private dat
declared *relations*.

The relations define the invariants that must be ve
values of the type.

The notion of relational data type is *not* native t
compiler: it is provided via an external program ge
generates regular Caml code for a relational data typ

The Moca framework

Moca provides a notation to state predefined algebr
between constructors,

Moca provides a notation to define arbitrary rewritt
tween constructors.

Moca provides a module generator, `mocac`, that ger
to implement a corresponding normal form.

Team: Frédéric Blanqui & Pierre Weis (Researche
Bonichon (Post Doc), Laura Lowenthal (Internshi
Hardin (Professor Lip6).

See `http://moca.inria.fr/`.

High level description of relat

We consider relational data types defined using:

- nullary or constant constructors,
- unary or binary constructors,
- nary constructors (argument has type α list).

Arguments cannot be *too complex* (in particular fu

Properties of constructors

A binary constructor op of an RDT `t` can be declare

- `associative` meaning that $\forall x, y, z \in$ `t` $: (x \ op$
  $x \ op \ (y \ op \ z)$,
- `commutative` meaning that $\forall x, y \in$ `t` $: x \ op \ y = y \ o$
- `distributive` with respect to another binary oper
  meaning that $\forall x, y, z \in$ `t` $: (x \ opp \ y) \ op \ z = (x \ op \ y)$

Properties of constructors

A binary constructor op of a RDT `t` can be declared

- having e as its **`neutral`** meaning that $\forall x \in \mathtt{t} : x\ op$
  x,
- having opp as **`opposite`** meaning that $\exists e \in \mathtt{t}, e$ is
  op, and $\forall x \in \mathtt{t} : x\ op\ (opp\ x) = (opp\ x)\ op\ x = e$,
- having z as its **`absorbent`** element meaning th
  $x\ op\ z = z\ op\ x = z$,

Properties of constructors

A unary constructor op of a RDT `t` can be declared

- being **`idempotent`** meaning that $\forall x \in$ **`t`** $:\ op\ (op\ x$
- being **`nilpotent`** wrt z meaning that $\forall x \in$ **`t`** $:\ op$
- being **`involutive`** meaning that $\forall x \in$ **`t`** $:\ op\ (op\ x$

Defining arbitrary relation

A constructor op of a RDT `t` can have one or more
declared as:

- **`rule`** op **`pat`** $\rightarrow$ **`expr`** meaning that any occurrenc
  op **`pat`** has to be rewritten as **`expr`**

Example:

```
rule Bool_not (Bool_true) -> Bool_false
```

The mocac compiler

From these specifications, the *mocac* compiler ge
construction functions that build the normal form o
verifies the algebraic relations and the invariants of
type.

The *mocac* compiler is a module generator for RD

The input for mocac is a file with suffix `.mlm`: it is a
file with specific annotations to define the relations

Examples

A trivial example with no annotations:

```
type bexpr = private
   | Band of bexpr list
   | Bor of bexpr list
   | Btrue
   | Bfalse;;
```

Generated files

Interface:

```
type bexpr = private
   | Band of bexpr list
   | Bor of bexpr list
   | Btrue
   | Bfalse;;
val bfalse : bexpr
val band : bexpr list -> bexpr
val bor : bexpr list -> bexpr
val btrue : bexpr
```

Generated files

Implementation:

```
type bexpr =
   | Band of bexpr list
   | Bor of bexpr list
   | Btrue
   | Bfalse

let rec bfalse = Bfalse
and band x = Band x
and bor x = Bor x
and btrue = Btrue
```

`.mlm` source file

A more realistic example for boolean expressions:

```
type bexpr = private
   | Band of bexpr * bexpr
   begin
     associative
     commutative
     distributive (Bxor)
     neutral (Btrue)
     absorbing (Bfalse)
     opposite (Binv)
   end
```

`.mlm` source file

```
| Bxor of bexpr * bexpr
begin
  associative
  commutative
  neutral (Bfalse)
  opposite (Bopp)
end
```

`.mlm` source file

```
| Btrue
| Bfalse
| Bvar of string

| Bopp of bexpr
begin
  rule Bopp(Btrue) -> Btrue
end

| Binv of bexpr;;
```

Generated interface

```
type bexpr = private
  | Band of bexpr * bexpr
  (*
    associative
    commutative
    distributive (Bxor)
    neutral (Btrue)
    absorbing (Bfalse)
    opposite (Binv)
  *)
  ...
```

Generated implementation

Type definition + simple operators

```
type bexpr = ...

let rec bvar x = Bvar x
and bopp x =
  match x with
  | Btrue -> Btrue
  | Bfalse -> Bfalse
  | Bopp x -> x
  | Bxor (x, y) -> bxor (bopp x, bopp y)
  | _ -> Bopp x
and bfalse = Bfalse
```

Generated implementatio

Binary associative + commutative operators are mo

```
and band z =
  match z with
  | Bfalse, _ -> Bfalse
  | _, Bfalse -> Bfalse
  | Btrue, y -> y
  | x, Btrue -> x
  | Binv x, y -> insert_opp_in_band x y
  | x, Binv y -> insert_opp_in_band y x
  | Bxor (x, y), z -> bxor (band (x, z), band (y,
  | x, Bxor (y, z) -> bxor (band (x, y), band (x,
  | Band (x, y), z -> band (x, band (y, z))
  | x, y -> insert_in_band x y
```

Generated implementation

Insertion in a `band` comb

```
and insert_in_band x u =
  match u with
  | Band (Binv y, t) when y = x -> t
  | Band (y, t) when x <= y ->
      begin try delete_in_band (Binv x) u with
        Not_found -> Band (x, u)
      end
  | Band (y, t) -> Band (y, insert_in_band x t)
  | Binv y when y = x -> Btrue
  | _ when x < u -> Band (x, u)
  | _ -> Band (u, x)
```

Generated implementatio

Deletion in a `band` comb (note that `band` is commut

```
and insert_opp_in_band x u =
  match u with
  | Band (y, t) when y = x -> t
  | Band (y, t) -> Band (y, insert_opp_in_band x t
  | _ when x = u -> Btrue
  | _ -> insert_in_band (Binv x) u
and delete_in_band x u =
  match u with
  | Band (y, t) when y = x -> t
  | Band (y, (Band (_, _) as t)) -> Band (y, delet
  | Band (y, t) when x = t -> y
  | _ -> raise Not_found
```

Generated implementatio

The inverse operator cannot be defined on the ab
ment...

```
and binv x =
  match x with
  | Bfalse -> failwith "Division by Absorbing elem
  | Btrue -> Btrue
  | Binv x -> x
  | Band (x, y) -> band (binv x, binv y)
  | _ -> Binv x
and btrue = Btrue
and bxor z = ...
```

`.mlm` source file

Two binary operators and their associated (ring-like

```
type aexpr = private
  | Add of aexpr * aexpr
  begin
    associative
    commutative
    neutral (Zero)
    opposite (Opp)
  end
```

`.mlm` source file

```
| Mul of aexpr * aexpr
begin
  associative
  commutative
  distributive (Add)
  neutral (One)
  absorbing (Zero)
  opposite (Inv)
end
| One
| Zero
| Var of string
| Opp of aexpr
| Inv of aexpr;;
```

Generated interface

Just regular: export the RDT type and its constr
tions:

```
type aexpr = private
  | Add of aexpr * aexpr ...

val var : string -> aexpr
val opp : aexpr -> aexpr
val mul : aexpr * aexpr -> aexpr
val inv : aexpr -> aexpr
val add : aexpr * aexpr -> aexpr
val zero : aexpr
val one : aexpr
```

Generated implementation

```
type aexpr =
    | Add of aexpr * aexpr
 ...

let rec var x = Var x
and opp x =
  match x with
  | Zero -> Zero
  | Opp x -> x
  | Add (x, y) -> add (opp x, opp y)
  | _ -> Opp x
```

Generated implementatio

Binary operators:

```
and mul z =
  match z with
  | Zero, _ -> Zero
  | _, Zero -> Zero
  | One, y -> y
  | x, One -> x
  | Inv x, y -> insert_opp_in_mul x y
  | x, Inv y -> insert_opp_in_mul y x
  | Add (x, y), z -> add (mul (x, z), mul (y, z))
  | x, Add (y, z) -> add (mul (x, y), mul (x, z))
  | Mul (x, y), z -> mul (x, mul (y, z))
  | x, y -> insert_in_mul x y
```

Generated implementation

Insertion

```
and insert_in_mul x u =
  match u with
  | Mul (Inv y, t) when y = x -> t
  | Mul (y, t) when x <= y ->
      begin try delete_in_mul (Inv x) u with
      | Not_found -> Mul (x, u)
      end
  | Mul (y, t) -> Mul (y, insert_in_mul x t)
  | Inv y when y = x -> One
  | _ when x < u -> Mul (x, u)
  | _ -> Mul (u, x)
```

Generated implementation

Deletion

```
and insert_opp_in_mul x u =
  match u with
  | Mul (y, t) when y = x -> t
  | Mul (y, t) -> Mul (y, insert_opp_in_mul x t)
  | _ when x = u -> One
  | _ -> insert_in_mul (Inv x) u
and delete_in_mul x u =
  match u with
  | Mul (y, t) when y = x -> t
  | Mul (y, (Mul (_, _) as t)) -> Mul (y, delete_i
  | Mul (y, t) when x = t -> y
  | _ -> raise Not_found
```

Generated implementation

Definition of inverse, and so on

```
and inv x =
  match x with
  | Zero -> failwith "Division by Absorbing elemen
  | One -> One
  | Inv x -> x
  | Mul (x, y) -> mul (inv x, inv y)
  | _ -> Inv x
...
and zero = Zero
and one = One
```

Maximal sharing generatio

The moca compiler also provides values represented a
shared trees.

You just have to use the `-sharing` option of the con

Hence the `.mlm` source file for maximally “arith” v
same.

Generated interface

The interface is slightly modified to incorporate the
into values:

```
type info = { mutable hash : int };;
type aexpr = private
   | Add of info * aexpr * aexpr
   ...
;;
```

Generated interface

Construction functions are similar; an additional equa
is also provided (to benefit from the sharing to get
with ==)

```
val var : string -> aexpr
...
val eq_aexpr : aexpr -> aexpr -> bool
```

Generated implementation

The implementation defines the types and the hash
ator:

```
type info = { mutable hash : int }
type aexpr =
   | Add of info * aexpr * aexpr
    ...

let mk_info h = {hash = h}
```

Generated implementation

The implementation defines an equality to share va

```
let rec equal_aexpr x y = x == y;;
```

Generated implementatio

Then the hash key access functions for the RDT

```
let rec get_hash_aexpr x =
  match x with
  | Add ({hash = h}, _x1, _x2) -> h
  | Mul ({hash = h}, _x1, _x2) -> h
  | Var ({hash = h}, _x1) -> h
  | Opp ({hash = h}, _x1) -> h
  | Inv ({hash = h}, _x1) -> h
  | One -> 1
  | Zero -> 0
```

Generated implementation

Then the hash code computation function

```
let rec hash_aexpr x =
  succ
    (match x with
     | Add (_, x1, x2) ->
         get_hash_aexpr x1 + (get_hash_aexpr x2 +
     | Mul (_, x1, x2) ->
         get_hash_aexpr x1 + (get_hash_aexpr x2 +
     | Var (_, x1) -> Hashtbl.hash x1 + Obj.tag (O
     | Opp (_, x1) -> get_hash_aexpr x1 + Obj.tag
     | Inv (_, x1) -> get_hash_aexpr x1 + Obj.tag
     | One -> 1
     | Zero -> 0)
```

Generated implementatio

Then those functions are encapsulated into a weak

```
module Hashed_aexpr =
  struct type t = aexpr let equal = equal_aexpr le

module Shared_aexpr = Weak.Make (Hashed_aexpr)

let table_aexpr = Shared_aexpr.create 1009
```

Generated implementatio

The basic construction functions use sharing:

```
let rec mk_Add x1 x2 =

  let info = {hash = 0} in

  let v = Add (info, x1, x2) in

  let _ = info.hash <- hash_aexpr v in
  try Shared_aexpr.find table_aexpr v with
  | Not_found -> let _ = Shared_aexpr.add table_ae
...
```

Generated implementation

Then the normalisation functions also use the max
(calling `mk_Add`, `mk_Opp`):

```
let rec var x = mk_Var x
and opp x =
  match x with
  | Zero -> Zero
  | Opp (_, x) -> x
  | Add (_, x, y) -> add (opp x, opp y)
  | _ -> mk_Opp x

and mul z = ...
and zero = Zero
and one = One
```

Current state of mocac

We use a KB completion tool to complete the u
relations.

We generate automatic test beds for the generated
functions.

We wrote a paper at ESOP'07: it states the fram
vides definitions of the desired construction function
correctness of the construction functions in simple

Future work

Still need to:

- prove the generated code (i.e. provide a proof fo erated implementation),
- or prove the code generator (better: once and f

Not so easy :(

We need also to integrate/interface *mocac* to other

- for Focal (more work to do, need pattern match
- for Tom/Gom (Pierre-Étienne Moreau, INRIA L