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 mathematical quotient structures,
- define automatic computation of canonical representant of values.

Usual data type definition kinds

There are three classical *kinds* of data type definitions:

- *sum* type definitions (disjoint union of sets with tagged summands),
- *product* type definitions (anonymous cartesian products) (cartesian products with named components)
- *abbreviation* type definitions (short hands to name type expressions)

Visibility of data type definitions

There are two classical *visibility* of a data type definitions:

- *concrete* visibility: the implementation of the type is visible,
- *abstract* visibility: the implementation of the type is hidden,

Consequence of visibility for programmers

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 programs

For concrete types, the representation of values is manifest:

- the compiler can perform type based optimization,
- the debugger (and the toplevel) can show (print) values.

For abstract types, the representation of values is hidden:

- the compiler cannot perform type based optimization,
- the debugger and the toplevel system just print values as `<abstr>`.

Visibility management constructs

Modules are used to define visibility of data type definitions.

- the implementation defines the data type as concrete,
- the interface exports the data type as concrete/or abstract.

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

Defining invariants

Usual (concrete) data types implement *free* data structures:

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

By *free* we mean the usual mathematical meaning: no restriction on the construction of values of the set (type), provided the 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 `mile` value).

Non free data types

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

Many data structures are not free having various validity constraints.

The usual feature of programming languages to deal with non free data structure is to provide abstract visibility and abstract data types (or ADT).

ADT as Non free data type

Using an ADT, the constructors, labels, or type expression synonym of the type are no more accessible to build spurious undesired values.
Construction of values is restricted to *construction functions* defined in the implementation module of the abstract data type.

Advantage: non free data types invariants are properly handled.
Drawback: inspection of values is no more a built in feature. Inspection functions should be provided explicitely by the implementation 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 Positive i;;
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" else k;;

let float_of_km k = k;;

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

Private visibility

To provide pattern matching for non free data types, we introduced a new visibility for data type definitions: the *private* visibility.

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

In short, private data type are:

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

- fully compatible with pattern matching.

Examples

All the quotient sets you need can be implemented as private types.

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

Formulas, groups, . . .

Definition of private data types

As abstract and concrete data types, private data types are implemented using modules:

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

- in the *interface* of their defining module, private data types are simply declared as *private*.

Usage of a private data type

In client modules:

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

Consequences

The module that implements a private data type:

- must export *construction functions* to build the values,
- has not to provide *destruction functions* to access inside the values.

The pattern matching facility is available for private data types.

Comparison with abstract data types

Abstract data types also provide invariants, but:

- once defined, an ADT is *closed*: new functions on the ADT are mere compositions of those provided by the module.

- once defined, a private data type is *still open*: arbitrary new functions can be defined via pattern matching on the representation of values.

Consequences

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

Consequences

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

Clients of a PDT must use the construction functions, to preserve invariants but pattern matching is still freely available.

All the functions defined on an PDT *respect* the PDT's invariants (granted for free by the type-checker!)

Relational data types

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

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

The notion of relational data type is *not* native to the Caml compiler: it is provided via an external program generator that generates regular Caml code for a relational data type definition.

The Moca framework

Moca provides a notation to state predefined algebraic relations between constructors,

Moca provides a notation to define arbitrary rewritting rules between constructors.

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

Team: Frédéric Blanqui & Pierre Weis (Researchers), Richard Bonichon (Post Doc), Laura Lowenthal (Internship), Thérèse Hardin (Professor Lip6).

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

High level description of relations

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 functionnal).

Properties of constructors

A binary constructor op of an RDT `t` can be declared as:

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

Properties of constructors

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

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

Properties of constructors

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

- being `idempotent` meaning that $\forall x \in \texttt{t} : \ op\ (op\ x) = \ op\ x$,
- being `nilpotent` wrt z meaning that $\forall x \in \texttt{t} : \ op\ (op\ x) = z$,
- being `involutive` meaning that $\forall x \in \texttt{t} : \ op\ (op\ x) = x$,

Defining arbitrary relations

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

- `rule` op `pat` $\rightarrow$ `expr` meaning that any occurrence of pattern 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 generates the construction functions that build the normal form of values that verifies the algebraic relations and the invariants of a relational type.

The *mocac* compiler is a module generator for RDTs.

The input for mocac is a file with suffix `.mlm`: it is a regular Caml 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 implementation

Binary associative + commutative operators are more tricky

```
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, z))
  | x, Bxor (y, z) -> bxor (band (x, y), band (x, z))
  | 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 implementation

Deletion in a `band` comb (note that `band` is commutative)

```
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, delete_in_band x t)
  | Band (y, t) when x = t -> y
  | _ -> raise Not_found
```

Generated implementation

The inverse operator cannot be defined on the absorbing element...

```
and binv x =
  match x with
  | Bfalse -> failwith "Division by Absorbing element"
  | 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) stuff:

```
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 construction functions:

```
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 implementation

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_in_mul x t)
  | 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 element"
  | One -> One
  | Inv x -> x
  | Mul (x, y) -> mul (inv x, inv y)
  | _ -> Inv x
...
and zero = Zero
and one = One
```

Maximal sharing generation

The moca compiler also provides values represented as maximally shared trees.

You just have to use the `-sharing` option of the compiler.

Hence the `.mlm` source file for maximally "arith" values is the same.

Generated interface

The interface is slightly modified to incorporate the hash codes into values:

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

Generated interface

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

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

Generated implementation

The implementation defines the types and the hash code generator:

```
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 values:

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

Generated implementation

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 + Obj.tag (Obj.repr x))
     | Mul (_, x1, x2) ->
         get_hash_aexpr x1 + (get_hash_aexpr x2 + Obj.tag (Obj.repr x))
     | Var (_, x1) -> Hashtbl.hash x1 + Obj.tag (Obj.repr x)
     | Opp (_, x1) -> get_hash_aexpr x1 + Obj.tag (Obj.repr x)
     | Inv (_, x1) -> get_hash_aexpr x1 + Obj.tag (Obj.repr x)
     | One -> 1
     | Zero -> 0)
```

Generated implementation

Then those functions are encapsulated into a weak hash table:

```
module Hashed_aexpr =
  struct type t = aexpr let equal = equal_aexpr let hash = hash_aexpr end

module Shared_aexpr = Weak.Make (Hashed_aexpr)

let table_aexpr = Shared_aexpr.create 1009
```

Generated implementation

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_aexpr v in v
...
```

Generated implementation

Then the normalisation functions also use the maximal sharing (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 user's set of relations.

We generate automatic test beds for the generated construction functions.

We wrote a paper at ESOP'07: it states the framework, provides definitions of the desired construction functions, proves the correctness of the construction functions in simple cases.

Future work

Still need to:

- prove the generated code (i.e. provide a proof for each generated implementation),
- or prove the code generator (better: once and for all).

Not so easy :(

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

- for Focal (more work to do, need pattern matching first),
- for Tom/Gom (Pierre-Étienne Moreau, INRIA Lorraine) ?