Interpretation and Compilation of Programming Languages Part 8 - Data Abstraction

João Costa Seco

May 21, 2014

1 Introduction

Functional abstraction is a form of adding operations to a programming language, and hence extending it. When combined with the declaration of new names and operators (like `Haskell` and `C++`), and together with modularization mechanisms, the essence of the new operations becomes indistinguishable from built-in operations. An orthogonal form of abstraction used in programming languages is data abstraction, which adds new data types to the language by composition of existing types.

In this lecture we presented the data types for Pairs and Records, as operators that aggregate heterogeneous values, and Collections (Arrays, Lists, etc.) that aggregate values of the same type.

2 Pairs

We introduce pairs as the base mechanism for data abstraction. Pairs are primitive in many languages such as Lisp, OCaml, Scala, etc. They can otherwise be defined using more complex mechanisms (classes and objects).

We introduce the type of a pair as the product of two types, given that the elements of such type are all possible pairs of elements that belong to each of the base types. When types are seen as sets of values, the pair type is the Cartesian product of the two base types. The destructors (the expressions that allow the inspection of a value) for pairs are the expressions `fst e` and `snd e`, whose denotation is the first/second component of the

```
data AST =
    Num Int
  | Add AST AST
  ...
  | Pair AST AST
  | Fst AST
  | Snd AST
```

Figure 1: Pair Operations

```
data Value =
    Integer Int
    ...
  | PairV Value Value
```

Figure 2: Pair Value

pair value denoted by expression `e`. See figure 1 for the abstract syntax of expressions and for the values in figure 2.

The (*call-by-value*) semantics of the newly introduced expressions is quite straightforward and consists in evaluating the subexpression and defining a new pair value, or the opposite, where a pair is inspected and one of the components is used as a result. See the operational semantics in figure 3. The *call-by-name* evaluation strategy of pairs is achieved by only evaluating the components of pairs if they are indeed used. Thus the pair is built with the delayed values. See figure 4 and notice that `c_eval` is called when the pairs need to be inspected.

Given this semantics, one can define programs that refer to infinite values for instance without the evaluation entering a never ending recursive cycle. Consider the following code:

```
decl head = fun x -> fst x in
decl tail = fun x -> snd x in
decl evenNumbers =
  decl f = fun g -> fun x -> (x, (g g) (x+2)) in (f f) 2 in
head (tail (tail evenNumbers))
```

This (untyped) code uses the technique of passing the continuation to a function to emulate recursive calls. Notice that under a *call-by-value* semantics, it will enter an infinite sequence of function calls `(g g)`. The name

```
eval (Pair e e') env mem =
         let (v,mem') = eval e env mem in
         let (v',mem'') = eval e' env mem' in
         case (v,v') of
              (Undefined, _) -> (Undefined,mem'')
              (_, Undefined) -> (Undefined,mem'')
              (v,v') -> (PairV v v',mem'')

eval (Fst e) env mem =
         let (v,mem') = eval e env mem in
         case v of
              PairV v v' -> (v,mem')
              _ -> (Undefined,mem')

eval (Snd e) env mem =
         let (v,mem') = eval e env mem in
         case v of
              PairV v v' -> (v',mem')
              _ -> (Undefined,mem')
```

Figure 3: Pair operations semantics

`evenNumbers` represents here an infinite list, that accepts the operations `head` and `tail`. The tail of the list is computed by a recursive call to the generator function `f`, which is delayed until in fact the values are requested. (Try this example in branch `L6.call-by-name`).

2.1 Typing

As far as typing is concerned, the type of pair values corresponds to a type composed by the types of its component values. See Figure 5 for the type language, and Figure 6 for the typing function.

Notice that pairs are the most primitive data abstraction mechanism, that allows the definition of a new datatype by composition of two other existing types. Any mechanism, such as lists, records, arrays, etc., can be operationally defined using pairs (see example above). However, from a typing point of view, new mechanisms offer a higher level of abstraction and

```
eval (Pair e e') env mem = (PairV (Delay e env) (Delay e' env),mem)

eval (Fst e) env mem =
         let (v,mem') = c_eval e env mem in
         case v of
              PairV v v' -> (v,mem')
              _ -> (Undefined,mem')


eval (Snd e) env mem =
         let (v,mem') = c_eval e env mem in
         case v of
              PairV v v' -> (v',mem')
              _ -> (Undefined,mem')
```

Figure 4: Pair operations *call-by-name* semantics

```
data Type =
    IntType
    ...
  | PairType Type Type
  | NoType
```

Figure 5: Pair types

pragmatic utility in many programming situations. We now present other kinds of data abstraction mechanisms. Notice that tuples with more than two components are trivial extensions of pairs and product types.

3 Records

Records, also known as tagged products, are aggregations of heterogeneous components, identified by labels. They are present in languages like Pascal, OCaml, Haskell as Records, and in C as structs, and are the base mechanism to implement objects in Object-oriented languages. The base language construction expression is the record literal expression, `{ label = E, ... }`, that in other languages appear as

```
typecheck (Pair e e') env =
          case (typecheck e env, typecheck e' env) of
              (NoType,_) -> NoType
              (_,NoType) -> NoType
              (t,t') -> PairType t t'

typecheck (Fst e) env =
          case typecheck e env of
              NoType -> NoType
              PairType t t' -> t

typecheck (Snd e) env =
          case typecheck e env of
              NoType -> NoType
              PairType t t' -> t'
```

Figure 6: Typing of pair operations

Expression	Language
`(X:0.0, Y:0.0)`	Pascal
`{ true, 15}`	C (fields are placed in the defined order)
`{ .flag = true, .value = 15 }`	C (or you can define the label)
`{a:0, b:1}`	Javascript

and the base destruction operation is field selection, `E.label`, that can also take different forms

Expression	Language
`a.X`	Pascal
`s.flag`	C (if `s` is the struct)
`s->flag`	C (if `s` is a pointer)
`o.f`	Javascript
`with(o){...f...}`	Javascript (opens the object IN the environment)

Notice that in languages like C or Pascal, the definition of a state variable of type record is a first step to initialize a record, whose fields can then be initialized separately, like in

```
typedef struct { int a; int b; } pair;
...
```

```
data AST =
    Num Int
  | Add AST AST
  ...
  | Record [(String,AST)]
  | Select AST String
```

Figure 7: Records abstract syntax

```
pair p;
p.a = 1;
p.b = 2;
...
```

The abstract syntax of record construction is shown in Figure 7, and the operational semantics, whose results are depicted in Figure 8, is given by Figure 9. The semantics of the constructor consists in iterating and evaluating the comprised expressions, and to aggregate the values to build the corresponding record value. The selection expression is evaluated by selecting the value that is associated with the label, in a record value denoted by the expression in the selection operation.

Notice that we opted here to evaluate the expressions comprised in the record expression immediately. The variant of the operational semantics implementing *call-by-name* is depicted in Figure 10.

In this case, we can observe that an example such as:

```
decl v = var(0) in
  v := !v + fst { a = (2,1), b = v := !v + 1 }.a; !v
```

will have value `2` as denotation, which means that the assignment in the `b` field expression is not evaluated. A similar example to the one above representing lists is also possible to write.

Exercise 1. Write the example mentioned above.

```
data Value =
    Integer Int
    ...
  | RecordV [(String,Value)]
```

Figure 8: Record values

```
eval (Record fields) env mem =
          (RecordV $ zip labels values, mem')
          where
          (values,mem') = foldr evalOne ([],mem) fieldExps
          (labels,fieldExps) = unzip fields
          evalOne = \e -> \(vs,mem') ->
                       let (v,mem'') = eval e env mem' in (v:vs,mem'')

eval (Select e label) env mem =
          case c_eval e env mem of
              RecordV fields -> head [ v | (l,v) <- fields, l == label]
              _ -> Undefined
```

Figure 9: Record operational semantics

```
eval (Record fields) env mem =
          (RecordV $ zip labels values, mem)
          where
          values = map (\e -> Delay e env) fieldExps
          (labels,fieldExps) = unzip fields

eval (Select e label) env mem =
          case c_eval e env mem of
              RecordV fields -> head [ v | (l,v) <- fields, l == label]
              _ -> Undefined
```

Figure 10: Record operational semantics *call-by-name*

3.1 Typing

The typing of records is similar to pairs, but indexed by the record labels. The type language for records is depicted in Figure 11 and the typing function is defined in Figure 12.

```
data Type =
    IntType
    ...
  | RecordType [(String,Type)]
```

Figure 11: Record types

```
typecheck (Record fields) env =
          if NoType `elem` types then NoType
          else RecordType $ zip labels types
          where
          types = map (\e -> typecheck e env) fieldExps
          (labels,fieldExps) = unzip fields

typecheck (Select e label) env =
          case typecheck e env of
              RecordType types -> head [ t | (l,t) <- types, l == label]
              _ -> NoType
```

Figure 12: Typing records