Data structures

Pedro Vasconcelos

March 4, 2024

Plan

Data structures

This lecture: extend the FUN language with algebraic data structures:

- pairs and tuples;
- generalized products: records;
- sums;
- generalized sums: variants.

Bibliography: Chapter 3 of *Programming languages*, Mike Grant and Scott Smith.
`http://pl.cs.jhu.edu/pl/book/book.pdf`

Why extend the language?

We could encode data structures with just the λ-calculus (e.g. using Church encodings).

Problems:

1. maybe we want to hide the implementation details;
2. maybe we want to static types (the Church encodings are untyped);
3. the Church encodings may be less efficient than a specialized implementation.

Alternative: add data structures to the *language* but use Church encodings in the *implementation*.

Plan

Pairs

Extensions to the Fun language

- Combination of two values
- Corresponds to the cartesian product
- One constructor and two eliminators (projections)

$$
\begin{array}{rcll}
e & ::= & \cdots & \\
 & | & \textbf{pair}\ e_1\ e_2 & \text{constructor} \\
 & | & \textbf{fst}\ e & \text{first projection} \\
 & | & \textbf{snd}\ e & \text{second projection}
\end{array}
$$

Pairs

Extensions to the operational semantics

Augment the set of values with pairs:

$$v ::= \cdots \mid (v_1, v_2)$$

Three new rules:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\textbf{pair}\ e_1\ e_2 \Downarrow (v_1, v_2)}$$

$$\frac{e \Downarrow (v_1, v_2)}{\textbf{fst}\ e \Downarrow v_1} \qquad \frac{e \Downarrow (v_1, v_2)}{\textbf{snd}\ e \Downarrow v_2}$$

Exercise: modify the Haskell interpreters.

Plan

Tuples

$$(e_1, e_2, \ldots, e_n)$$

$n = 0$: empty tuple (unit)

$n = 1$: N/A

$n = 2$: pairs

$n > 2$: triples, quartets, etc.

Tuples

- In a strict language tuples are semantically equivalent to nested pairs, e.g.:

$$(e_1,\ e_2,\ e_3) \equiv (e_1,\ (e_2,\ e_3))$$

- In a lazy language we need to be careful with undefined values:

$$(e_1, \bot) \neq \bot$$

- The empty tuple () is a special case:
  - only one possible value (not a composition)
  - behaves like a “unit value” for compositions
- In Haskell, different size tuples have distinct types
  - the standard requires constructors only for $n \leq 15$
  - access using pattern matching
  - the Prelude defines projections only for pairs (*not* built-in)

Plan

1. Pairs
2. Tuples
3. Records
4. Sums
5. Variants

Records

- Generalization of products with labelled fields, e.g.

$$\{\mathit{name} = \texttt{"John"},\ \mathit{age} = 30\}$$

  instead of

$$(\texttt{"John"}, 30)$$

- Order of fields is not significant, e.g.

$$\{\mathit{name} = \texttt{"John"},\ \mathit{age} = 30\} \equiv \{\mathit{age} = 30,\ \mathit{name} = \texttt{"John"}\}$$

- Projections using field names:

$$\{\mathit{name} = \texttt{"John"},\ \mathit{age} = 30\}.\mathit{name} \equiv \texttt{"John"}$$
$$\{\mathit{name} = \texttt{"John"},\ \mathit{age} = 30\}.\mathit{age} \equiv 30$$

Records

Extensions to the Fun language

$$
\begin{array}{rcll}
e & ::= & \cdots & \\
 & \mid & \{\ell_1 = e_1; \ldots; \ell_n = e_n\} & \text{construction} \\
 & \mid & e.\ell & \text{selection}
\end{array}
$$

- Assume some fixed set of labels ℓ_1, ℓ_2, etc.
- Similar syntax to atributes and methods in objects
- Field names ℓ are are *not* first-class, i.e. you can write

$$record.\ell$$

but not

$$\lambda x.\, record.x$$

Encoding records using tuples

- If the set of fields is known at compile time we can encode records as tuples
- Associates each field with a fixed position in the tuple
- Example:

$$\begin{aligned}
\{x = 5; y = 7; z = 6\} &\equiv \textbf{pair}\ 5\ (\textbf{pair}\ 7\ 6)) \\
e.x &\equiv \textbf{fst}\ e \\
e.y &\equiv \textbf{fst}\ (\textbf{snd}\ e) \\
e.z &\equiv \textbf{snd}\ (\textbf{snd}\ e)
\end{aligned}$$

- Disadvantage: requires the whole program
- Alternative: extend the operational semantics with record values

Records

Extensions to the operational semantics

Values:

$$v \ ::= \ \cdots \mid \{\ell_1 = v_1; \ldots \ell_n = v_n\}$$

Evaluation rules:

$$\frac{e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n}{\{\ell_1 = e_1; \ldots; \ell_n = e_n\} \Downarrow \{\ell_1 = v_1; \ldots; \ell_n = v_n\}}$$

$$\frac{e \Downarrow \{\ell_1 = v_1; \ldots; \ell_i = v_i; \ldots; \ell_n = v_n\}}{e.\ell_i \Downarrow v_i}$$

Records in Haskell

- Records are *nominal*, not *structural*
- Special case of data declarations
- Field names can be used for projections
- Field names must be unique (in a given namespace)

```
data Person = Person { name :: String }
data Company = Company { companyName :: String,
                         owner :: Person }
-- name :: Person -> String
-- companyName :: Company -> String
-- owner :: Company -> Person

main = do
   let p = Person {name="Wile E. Coyote"}
   let c = Company {companyName = "Acme corp.", owner=p}
   print (companyName c ++ " is run by ++ name (owner c))
```

Record update in Haskell

Haskell allows functional updates of records, i.e. creating a new record with some fields updated.

```
data Person = Person { name :: String }
data Company = Company { companyName :: String,
                         owner :: Person }

main = do
   let p = Person {name="Wile E. Coyote"}
   let c = Company {companyName = "Acme corp.", owner=p}
   let q = Person {name="Road Runner"}
   let c'= c { owner = q }
   print (companyName c' ++ " is run by ++ name (owner c'))
```

Records extensions in Haskell

GHC extensions (since 9.2) allow using dot notation and re-using field names.

```haskell
{-# LANGUAGE OverloadedRecordDot #-}
{-# LANGUAGE DuplicateRecordFields #-}

data Person = Person { name :: String }
data Company = Company { name :: String, owner :: Person }

main = do
  let p = Person { name = "Wile E. Coyote" }
  let c = Company { name = "Acme corp.", owner = p }
  print $ c.name ++ " is run by " ++ c.owner.name
```

Record polymorphism

A function

$$\lambda x.\ x.age$$

could be applied to any record with *age* field.
Examples:

- $\{name = \texttt{"Mike"}, age = 20\}$
- $\{model = \texttt{"Volvo"}, age = 5\}$
- General type allows ignoring other fields

$$\{age : \alpha\} \to \alpha$$

- This kind polymorphism is called *subtyping* (sometimes *record subtyping*)
- It is strongly related to object-oriented programming
- Haskell and OCaml do *not* suport record polymorphism; the `age` projection can only be applied to a specific type

Plan

Sums

Extensions to the Fun language

- Alternative between two values
- Corresponds to a disjoint sum
- Two constructors and one eliminator (case)

$$
\begin{array}{lcl}
e & ::= & \cdots \\
 & | & \mathbf{inl}\ e \\
 & | & \mathbf{inr}\ e \\
 & | & \mathbf{case}\ e_0\ \mathbf{of\ inl}\ x \to e_1 \mid \mathbf{inr}\ y \to e_2
\end{array}
$$

Sums

Extensions to the operational semantics

Values:

$$v \; ::= \; \cdots \mid \textbf{inl}\ v \mid \textbf{inr}\ v$$

Evaluation Rules:

$$\frac{e \Downarrow v}{\textbf{inl}\ e \Downarrow \textbf{inl}\ v} \qquad \frac{e \Downarrow v}{\textbf{inr}\ e \Downarrow \textbf{inr}\ v}$$

$$\frac{e_0 \Downarrow \textbf{inl}\ v \qquad e_1[v/x] \Downarrow u}{\textbf{case}\ e_0\ \textbf{of inl}\ x \to e_1 \mid \textbf{inr}\ y \to e_2 \Downarrow u}$$

$$\frac{e_0 \Downarrow \textbf{inr}\ v \qquad e_2[v/y] \Downarrow u}{\textbf{case}\ e_0\ \textbf{of inl}\ x \to e_1 \mid \textbf{inr}\ y \to e_2 \Downarrow u}$$

Plan

Variants

- Generalize sums to alternatives tagged by constructors
- Consider a fixed set of constructor tags c_1, c_2, etc.
- Selection using a *case* expressions (generalizes the sum case)

Variants in Haskell I

- Introduced by data declarations
- Each constructor can different number of arguments
- Each argument can have a different type
- Construct names have to be unique (in a given name space)

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
-- Mon, Tue, ... :: Weekday

data Maybe a = Nothing | Just a
-- Nothing :: Maybe a
-- Just :: a -> Maybe a
```

Variants in Haskell II

Case and pattern matching to scrutinze constructed values:

```
isWeekend :: Weekday -> Bool
isWeekend w = case w of
                Sat -> True
                Sun -> True
                _ -> False

fromMaybe :: Maybe a -> a
fromMaybe def opt
  = case opt of
       Nothing -> def
       Just v -> v
```

Variants

Extensions to the Fun language

$$
\begin{array}{rcll}
e & ::= & \cdots & \\
 & \mid & c(e) & \text{constructor} \\
 & \mid & \textbf{case}\ e_0\ \textbf{of} & \text{selection} \\
 & & \quad c_1(x_1) \rightarrow e_1 & \\
 & & \quad c_2(x_2) \rightarrow e_2 & \\
 & & \quad\quad \vdots & \\
 & & \quad c_n(x_n) \rightarrow e_n &
\end{array}
$$

- Patterns *bind* variables x_i inside e_i
- Simple patterns: the order does not matter
- Alternatives don't have to be exaustive

Variants

Extensions to the operational semantics

$$v \ ::= \ \cdots \ | \ c(v)$$

$$\frac{e \Downarrow v}{c(e) \Downarrow c(v)}$$

$$\frac{e \Downarrow c_j(v_j) \qquad e_j[v_j/x_j] \Downarrow v}{\left(\begin{array}{l}\textbf{case } e \textbf{ of} \\ \quad c_1(x_1) \to e_1 \\ \quad\quad \vdots \\ \quad c_j(x_j) \to e_j \\ \quad\quad \vdots \\ \quad c_n(x_n) \to e_n\end{array}\right) \Downarrow v}$$

Enconding lists using variants and tuples

- Two constructors: *nil* for the empty list and *cons* for non-empty lists
- The argument of *cons* is a pair (head and tail)
- The argument of *nil* is not relevant (i.e. empty tuple)

$$
\begin{aligned}
\texttt{[]} &\equiv \mathrm{nil}() \\
\texttt{(:)} &\equiv \lambda h.\, \lambda t.\, \mathrm{cons}\ (\mathbf{pair}\ h\ t) \\
\mathrm{null} &\equiv \lambda x.\, \mathbf{case}\ x\ \mathbf{of}\ \begin{array}{l} \mathrm{nil}(x) \to \mathrm{True} \\ \mathrm{cons}(p) \to \mathrm{False} \end{array} \\
\mathrm{head} &\equiv \lambda x.\, \mathbf{case}\ x\ \mathbf{of}\ \mathrm{cons}(p) \to \mathbf{fst}\ p \\
\mathrm{tail} &\equiv \lambda x.\, \mathbf{case}\ x\ \mathbf{of}\ \mathrm{cons}(p) \to \mathbf{snd}\ p
\end{aligned}
$$

Projections vs. case expressions

Example: recursive function for the length of a list.

```
length xs = if null xs then 0 else 1 + length (tail xs)

length xs = case xs of
              [] -> 0
              (x:xs') -> 1 + length xs'
```

The case expression:

- make structural recursion explicit
- avoids the need to build an intermediate boolean value (more efficient)

General pattern matching

- Multiple equations with patterns can be converted into a sigle definition with case expressions
- Nested patterns can be converted into nested case expressions with *simple* patterns

Efficient Compilation of Pattern-Matching, P. Wadler. Chapter 5 of *The Implementation of Functional Programming Languages*, S. L. Peyton Jones.

Example

Determine the last element of a list.

```
last [x] = x
last (x:xs) = last xs
```

1st transformation:

```
last xs = case xs of
          (x:[]) -> x
          (x:xs') -> last xs'
```

2nd transformation:

```
last xs = case xs of
          (x:xs') -> case xs' of
                     [] -> x
                     (x':xs'') -> last xs'
```

Example

Determine the last element of a list.

```
last [x] = x
last (x:xs) = last xs
```

1st transformation:

```
last xs = case xs of
          (x:[]) -> x
          (x:xs') -> last xs'
```

2nd transformation:

```
last xs = case xs of
          (x:xs') -> case xs' of
                      [] -> x
                      (x':xs'') -> last xs'
```

Example

Determine the last element of a list.

```
last [x] = x
last (x:xs) = last xs
```

1st transformation:

```
last xs = case xs of
          (x:[]) -> x
          (x:xs') -> last xs'
```

2nd transformation:

```
last xs = case xs of
          (x:xs') -> case xs' of
                     [] -> x
                     (x':xs'') -> last xs'
```