# Data and Control

A4M36TPJ, 2013/2014

# Data

- Products
- Sums
- Sums-of-Products

# Products

- Positional
- Sequences
- Lists
- Named
- Nonstrict
- Streams

# Positional Products

- Products are compound values that result from gluing other values together.
- Two-dimensional points, Records, …

$$
\begin{aligned}
E ::= &(prod \;\; E*_{component}) \\
&|(get \;\; N_{index} \;\; E_{prod})
\end{aligned}
$$

# Positional Products Operational Semantics

**Values**

$V \in \text{ValueExp} ::= \ \ldots \mid \texttt{(prod } V_1 \ \ldots \ V_n\texttt{)}$

The AnsExp domain and output function $OF$ would also have to be extended to handle `prod` values.

**Evaluation Contexts**

$\mathbb{E} \in \text{EvalContext} ::= \ \ldots \mid \texttt{(prod } V_{i=1}^{k-1} \ \mathbb{E} \ E_{j=k+1}^{n}\texttt{)} \mid \texttt{(get } N_{index} \ \mathbb{E}\texttt{)}$

**New Stateless Reduction Rule**

$\texttt{(get } N_{index} \ \texttt{(prod } V_1 \ \ldots \ V_n\texttt{))} \leadsto V_i, \quad [\texttt{get}]$
where $i = \mathcal{N}[\![N_{index}]\!]$ and $1 \leq i \leq n$

**Operational semantics for CBV products**

# Positional Products Denotational Semantics Example

**New and Modified Semantic Domains**

$$\begin{aligned} \mathit{Prod} &= \mathit{Value}^* \\ v \in \mathit{Value} &= \ldots + \mathit{Prod} \end{aligned}$$

**New Computation Operation**

$\mathit{with\text{-}product\text{-}comp} : \mathit{Comp} \to (\mathit{Prod} \to \mathit{Comp}) \to \mathit{Comp}$

The definition is similar to that of *with-boolean-comp* in Figure 6.26 on page 281.

$$\begin{aligned}
&\mathit{with\text{-}prod\text{-}and\text{-}checked\text{-}index} : \mathit{Comp} \to \mathit{Int} \to (\mathit{Prod} \to \mathit{Int} \to \mathit{Comp}) \to \mathit{Comp} \\
&= \lambda cif\ .\ \mathit{with\text{-}product\text{-}comp}\ c \\
&\qquad\qquad (\lambda v^*\ .\ \textbf{if}\ (1 \le i) \ \wedge\ (i \le (\mathit{length}\ \ v^*)) \\
&\qquad\qquad\qquad\quad \textbf{then}\ (f\ \ v^*\ \ i) \\
&\qquad\qquad\qquad\quad \textbf{else}\ (\mathit{err\text{-}to\text{-}comp}\ \texttt{out-of-bounds-product-index}) \\
&\qquad\qquad\qquad\quad \textbf{end})
\end{aligned}$$

**New Valuation Clauses**

$$\mathcal{E}[\![\texttt{(prod}\ \ E^*\texttt{)}]\!] = \lambda e\ .\ (\mathit{with\text{-}values}\ \ (\mathcal{E}^*[\![E^*]\!]\ \ e)\ \ (\lambda v^*\ .\ (\mathit{Prod} \rightarrowtail \mathit{Value}\ \ v^*)))$$

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(get}\ \ N_{index}\ \ E_{prod}\texttt{)}]\!] \\
&= \lambda e\ .\ \mathit{with\text{-}prod\text{-}and\text{-}checked\text{-}index}\ (\mathcal{E}[\![E_{prod}]\!]\ \ e)\ \mathcal{N}[\![N_{index}]\!] \\
&\qquad\quad (\lambda v^* i\ .\ (\mathit{val\text{-}to\text{-}comp}\ \ (\mathit{nth}\ \ i\ \ v^*)))
\end{aligned}$$

**Denotational semantics for CBV products**

# Positional Product - Immutable Sequence

$$
\begin{aligned}
E ::= &(seq \;\; E*_{component}) \\
&|(seq - get \;\; E_{index} \;\; E_{seq}) \\
&|(seq - get \;\; E_{seq})
\end{aligned}
$$

# Named Products

$$
\begin{aligned}
E ::= & (record \;\; I_{fieldName} \;\; E_{fieldDefn}) \\
& |(select \;\; I_{fieldName} \;\; E_{record})
\end{aligned}
$$

```
(let ((r (record (test (= 0 1)) (yes (* 2 3)) (no (+ 4 5)))))
  (if (select test r) (select yes r) (select no r)))
```

# Non-Strict Products

- Previous products were strict products, in which the expressions specifying the components were **fully evaluated**.

- Another type of products are non-strict products, in which the **component computations themselves** are stored within the product value and are performed **only when their values are "demanded."**

# Non-Strict Products Operational Semantics

**Values**

$V \in \text{ValueExp} ::= \; \ldots \mid (\texttt{nprod}\ E_1\ \ldots\ E_n)$

The AnsExp domain and output function $OF$ would also have to be extended to handle `nprod` values.

**Evaluation Contexts**

$\mathbb{E} \in \text{EvalContext} ::= \; \ldots \mid (\texttt{nget}\ N_{index}\ \mathbb{E})$

**New Stateless Reduction Rule**

$(\texttt{nget}\ N_{index}\ (\texttt{nprod}\ E_1\ \ldots\ E_n)) \rightsquigarrow E_i, \quad [\texttt{nget}]$
where $i = \mathcal{N}[\![N_{index}]\!]$ and $1 \leq i \leq n$

**Operational semantics for CBN products**

# Non-Strict Products Denotational Semantics

**New and Modified Semantic Domains**

$$NProd = Comp^*$$

$$v \in Value = \ldots + NProd$$

**New Computation Operation**

$$\mathit{with\text{-}nprod\text{-}and\text{-}checked\text{-}index} : Comp \to Int \to (NProd \to Int \to Comp) \to Comp$$

The definition is similar to that of *with-prod-and-checked-index* in Figure 10.1 on page 543.

**New Valuation Clauses**

$$\mathcal{E}[\![\texttt{(nprod } E^*\texttt{)}]\!] = \lambda e\,.\,(NProd \rightarrowtail Value\ (\mathcal{E}^*[\![E^*]\!]\ e))$$

$$\mathcal{E}[\![\texttt{(nget } N_{index}\ E_{prod}\texttt{)}]\!]$$
$$= \lambda e\,.\ \mathit{with\text{-}nprod\text{-}and\text{-}checked\text{-}index}\ (\mathcal{E}[\![E_{prod}]\!]\ e)\ \mathcal{N}[\![N_{index}]\!]$$
$$(\lambda c^* i\,.\,(nth\ i\ c^*))$$

**Denotational semantics for CBN products**

# Sums

- A **sum** is a data structure that can hold one of several different kinds of values. Sums are used in situations where programmers use the terms “either” or “one of” to informally describe a data structure

- For example:

    - A linked list is either a list node (with head and tail components) or the empty list.

    - A graphics system might support shapes that are either circles, rectangles, or triangles.

    - In a banking system, a transaction might be one of deposit, withdrawal, transfer, or balance query.

# Sums

- A sum value pairs an underlying value, which we call its **payload**, with a **tag** that indicates which kind of value the payload is.
- Processing a sum value usually involves performing a case analysis on its tag and manipulating its payload accordingly.

# Sums

$$E ::= (one \;\; I_{tag} \;\; E_{payload})$$
$$\;|(tagcase \;\; E_{disc} \;\; I_{payload} \;\; (I_{tag} \;\; E_{body})* \;\; (else \;\; E_{else})*$$

$\texttt{(one } I_{tag} \;\; E_{payload}\texttt{)} \rightsquigarrow_{ds} \texttt{(pair (sym } I_{tag}\texttt{) } E_{payload}\texttt{)}$

$$
\begin{array}{l}
\texttt{(tagcase } E_{disc} \;\; I_{payload} \;\; \texttt{(}I_i \;\; E_i\texttt{)}_{i=1}^{n} \;\; \texttt{(else } E_{else}\texttt{)}^{?}\texttt{)} \\
\quad \rightsquigarrow_{ds} \texttt{(let ((}I_{disc} \;\; E_{disc}\texttt{))} \qquad\qquad \{I_{disc}\ \mathit{fresh}\} \\
\qquad\qquad \texttt{(let ((}I_{tag} \;\; \texttt{(fst } I_{disc}\texttt{)))} \;\; \{I_{tag}\ \mathit{fresh}\} \\
\qquad\qquad\quad \texttt{(cond} \\
\qquad\qquad\qquad \texttt{((sym=? } I_{tag} \;\; \texttt{(sym } I_i\texttt{)) (let ((}I_{payload} \;\; \texttt{(snd } I_{disc}\texttt{))) } E_i\texttt{))}_{i=1}^{n} \\
\qquad\qquad\qquad \texttt{(else } E_{else}\texttt{)}^{?}\texttt{)))}
\end{array}
$$

# Sum-of-Products

- In practice, sum and product data are often used together in idiomatic ways.

- Many common data structures can be viewed as a **tree constructed from different kinds of nodes**, each of which has multiple components.

# Sum-of-Products Examples

- A **shape in a simple geometry system** is either:
  - a circle with a radius;
  - a rectangle with a width and a height;
  - a triangle with three side lengths.
- A **list of integers** is either:
  - an empty list;
  - a list node with an integer head and an integer-list tail.

# Sum-of-Products Examples

As a simple example, consider the following list of geometric shapes:

```
(list (one rectangle (record (width 3) (height 4)))
      (one triangle (record (side1 5) (side2 6) (side3 7)))
      (one square (record (side 2))))
```

```
(def (perim shape)
  (tagcase shape r
    (square (* 4 (select side r)))
    (rectangle (* 2 (+ (select width r) (select height r))))
    (triangle (+ (select side1 r)
                 (+ (select side2 r) (select side3 r))))))
```

# Data Declarations

- Programming with “raw” sums and products is cumbersome and error-prone.
- It is very reasonable to introduce data declaration constructs into the language.

# Data Declarations

```
(def-data shape
  (square side)
  (rectangle width height)
  (triangle side1 side2 side3))
```

```
(list (square 2) (rectangle 3 4) (triangle 7 8 9))
```

```
(list (one square (prod 2))
      (one rectangle (prod 3 4))
      (one triangle (prod 5 6 7)))
```

# Continuations

- A computation can be viewed **as an iteration over currently evaluated expression** and the **continuation of the current expression**.
- Thanks to enhanced control of the program flow, continuations can be used to return multiple values, nonlocal exits, error-handling and backtracking.

# Continuation Passing Style Multiply Example (No CPS)

$$\begin{aligned}
multiply &: R^* \to R \\
multiply(\langle\rangle) &= 1 \\
multiply(\langle a_1, a_2, \ldots, a_n\rangle) &= a_1 \cdot multiply(\langle a_2, \ldots, a_n\rangle) \quad \text{if } a_1 \neq 0 \\
multiply(\langle a_1, \ldots, a_n\rangle) &= 0 \quad \text{otherwise}
\end{aligned}$$

# Continuation Passing Style Multiply Example (CPS)

$$multiplyCPS : R^* \to R$$

$$multiplyCPS(\langle a_1, \ldots, a_n \rangle) = mult(\langle a_1, \ldots, a_n \rangle, \lambda x.x)$$

$$mult : R^* \times (R \to R) \to R$$

$$mult(\langle\rangle, k) = k(1)$$

$$mult(\langle a_1, a_2, \ldots, a_n \rangle, k) = mult(\langle a_2, \ldots, a_n \rangle, \lambda x.k(a_1 \cdot x)) \quad \text{if } a_1 \neq 0$$

$$mult(\langle a_1, \ldots, a_n \rangle, k) = 0 \quad \text{if } a_1 = 0$$

# Continuation Passing Style RegExp Matcher Example

$$match : RegExp \times A^* \to Boolean$$

$$match(r, \langle a_1, \ldots, a_n \rangle) = true \quad \text{if } m(r, \langle a_1, \ldots, a_n \rangle, \lambda x.x) = \langle\rangle$$

$$match(r, \langle a_1, \ldots, a_n \rangle) = false \quad \text{otherwise}$$

# Continuation Passing Style RegExp Matcher Example

$$A^{\perp} = A^* \cup \{\perp\}$$

$$m : RegExp \times A^{\perp} \times (A^{\perp} \to A^{\perp}) \to A^{\perp}$$

$$\begin{aligned}
m(r, \perp, k) &= k(\perp) \\
m(\epsilon, \langle\rangle, k) &= k(\langle\rangle) \\
m(\epsilon, \langle a_1, \ldots, a_n\rangle, k) &= k(\langle a_1, \ldots, a_n\rangle) \\
m(a, \langle\rangle, k) &= k(\perp)
\end{aligned}$$

# Continuation Passing Style RegExp Matcher Example

$$m(a, \langle a_1, a_2, \ldots, a_n \rangle, k) = k(\langle a_2, \ldots, a_n \rangle) \quad \text{if } a = a_1$$

$$m(a, \langle a_1, \ldots, a_n \rangle, k) = k(\perp) \quad \text{if } a \neq a_1$$

$$m(r_1 \cdot r_2, \langle a_1, \ldots, a_n \rangle, k) = m(r_1, \langle a_1, \ldots, a_n \rangle, \lambda x.m(r_2, x, k))$$

# Continuation Passing Style RegExp Matcher Example

$$
\begin{aligned}
m(r_1 + r_2, \langle a_1, \ldots, a_n \rangle, k) = m(r_1, \langle a_1, \ldots, a_n \rangle,& \\
\lambda x.\text{if } k(x) = \langle\rangle \text{ then } \langle\rangle \text{ else } m(r_2, \langle a_1, \ldots, a_n \rangle, k))&
\end{aligned}
$$

# Continuation Passing Style RegExp Matcher Example

$$m(r^*, \langle a_1, \ldots, a_n \rangle, k) = m(\epsilon + (r \cdot r^*), \langle a_1, \ldots, a_n \rangle, k)$$