Background:

In an undergraduate course called Models of Computation we studied a small language with functions, labelled products and labelled unions. This was a very minimal language, one of the purposes was to express a self interpreter for the language.

Trying to understand records, projections and constructors . . .

Bengt Nordström

bengt@cs.chalmers.se

ChungAng University, Seoul, Korea

on leave from Chalmers University, Göteborg, Sweden

A first small language

$e_1\ e_2$	application
$\lambda x \rightarrow e$	abstraction
c e	constructor application
$\textbf{case}\ e\ \textbf{of}\ \{c_1 : e_1,\ \ldots\}$	case-expression
$[l_1 = e_1;\ \ldots]$	structure
e.l	projection
$\textbf{rec}\ x = e\ \textbf{end}$	recursion
x	variable

Computation rules

- The value of $d\ e'$ is obtained by first computing the value of d. If this is on the form $\lambda x \rightarrow e$, then we compute the value of the expression obtained by substituting e' for all free occurrences of x in e.
- The value of $\mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{c}_1 : e_1;\ \ldots\}$ is obtained by first computing the value of e. If this is on the form $\mathsf{c}_\mathsf{i}\ e$ then we compute the value of $e_i\ e$.
- The program $e.\mathsf{l}_\mathsf{i}$ is computed by first computing the value of e. If this is on the form $[\mathsf{l}_1 = e_1;\ \ldots]$ then we compute the value of e_i.
- The program $\mathbf{rec}\ x = e\ \mathbf{end}$ is computed by computing the value of the expression obtained by substituting $\mathbf{rec}\ x = e\ \mathbf{end}$ for all free occurrences of x in e.

Syntax for case expressions

A case expression which is usually written as

$$\textbf{case } e \textbf{ of } \{\mathsf{c}_1\ x_1 \ldots x_k :\ e_1;$$

$$\vdots$$

$$\mathsf{c}_n\ x_1 \ldots x_l :\ e_n\}$$

can be expressed as:

$$\textbf{case } e \textbf{ of } \{\mathsf{c}_1 :\ \lambda x_1 \ldots \lambda x_k . e_1;$$

$$\vdots$$

$$\mathsf{c}_n :\ \lambda x_1 \ldots \lambda x_l . e_n\}$$

Reduction rules for projection and case

$$[\mathsf{c_1} = \; e_1; \ldots \mathsf{c_n} = \; e_n].c_k \longrightarrow e_k$$

$$\textbf{case } c_k \; a_1 \; \ldots \; a_n \textbf{ of } \{\mathsf{c_1} : \; e_1; \ldots \mathsf{c_n} : \; e_n\} \longrightarrow e_k \; a_1 \; \ldots \; a_n$$

if we change the syntax of the case a little:

$$[\mathsf{c_1} = e_1; \ldots \mathsf{c_n} = e_n].c_k \longrightarrow e_k$$

$$[\mathsf{c_1} = e_1; \ldots \mathsf{c_n} = e_n] \odot (c_k \; a_1 \; \ldots \; a_n) \longrightarrow e_k \; a_1 \; \ldots \; a_n$$

The second rule is more general.

Reduction rules for the general projection

The rule

$$[\mathsf{i}_1 = e_1; \ldots \mathsf{i}_\mathsf{n} = e_n] \;.\; i_k \; a_1 \;\ldots\; a_n \longrightarrow e_k \; a_1 \;\ldots\; a_n$$

can be expressed by the following:

$$\begin{aligned} [].i &\longrightarrow \text{error} \\ [i = a; b].i &\longrightarrow a \\ [i = a; b].j &\longrightarrow b.j && \text{if } i \neq j \\ r.(a\ b) &\longrightarrow (r.a)\ b && \text{if } (a\ b) \text{ is a constructor application} \end{aligned}$$

We compute $r.e$ by fi rst computing the value of r and the value of e.

Summary

We can reduce the ordinary record projection and the case-expression to a generalized projection

$$r.e$$

where the type of r is a labelled product and the type of e is a labelled union.

Another view of records

We can look at a record

$$[\mathsf{i}_1 = e_1; \ldots \mathsf{i}_\mathsf{n} = e_n]$$

as a list of defi nitions. We then have to defi ne a new kind of projection operator (called $\|$) which should work like a local let. The expression

$$[\mathsf{i}_1 = e_1; \ldots \mathsf{i}_\mathsf{n} = e_n] \parallel e$$

should express a local defi nition:

$$\textbf{let}\ [\mathsf{i}_1 = e_1; \ldots \mathsf{i}_\mathsf{n} = e_n]\ \textbf{in}\ e$$

The constructors in e are now looked as defi ned constants.

Reduction rules for $\|$

We now need to formulate reduction rules for the $\|$-operator:

$$
\begin{aligned}
[].i &\longrightarrow \text{error} & [] \parallel i &\longrightarrow i \\
[i = a; b].i &\longrightarrow a & [i = a; b]. \parallel i &\longrightarrow a \\
[i = a; b].j &\longrightarrow b.j & [i = a; b] \parallel j &\longrightarrow b \parallel j \\
r.(a\ b) &\longrightarrow (r.a)\ b & r \parallel (a\ b) &\longrightarrow (r \parallel a)\ (r \parallel b)
\end{aligned}
$$

But we also have to express what happens when we compute an expression of the shape $r \parallel e$ where the computation of r and e has got stuck in an identifi er which is to be defi ned.

The syntax of a small language

x	variable	i	constructor
$e_1\ e_2$	application	$e_1 \parallel e_2$	projection
$\lambda x.e$	abstraction	$[]$	void defi nition
		$[i = e_1; e_2]$	defi nition

Semantics

The following expressions are computed to themselves:

- A constructor i.
- A lambda-expression $\lambda x.e$
- $[]$
- $[i = e_1; e_2]$

We never compute open expressions, so there is no need to explain how to compute a variable. It remains to explain how an application and a projection is computed.

How to compute an application $a\ b$

We fi rst compute the value of a.

constructor If the value of a is a constructor i then we return the value $i\ b$.

application If the value of a is an application $(c\ d)$ then we compute the value b' of b and return the value $(c\ d)\ b'$.

abstraction If the value of a is an abstraction $\lambda x.c$ then we perform the β-reduction $(\lambda x.c)\ b \longrightarrow c[x \leftarrow b]$ and continue the computation.

record If the value of a is a defi nition list then there is an error.

projection If the value of a is a projection $c \parallel d$ then we return the value $(c \parallel d)\ b$.

How to compute a projection $r \parallel b$?

- When we compute the projection we fi rst compute the value of b.
- The general structure of a value is the same as for an expression, except that variables cannot occur.
- If the value of b is an expression e which is not a constructor, then we project along the parts of e (since we want the defi nitions in r to hold in the entire expression e) and continue the computation.
- If the value of b is a constructor i, then we compute the value of r.
  - If this computes to a record $[i_1 = c_1; \ldots; i_n = c_n]$, then we perform the projection.
  - If the constructor i is not defi ned, then we return the identifi er i as the result.

How to compute a projection $r \parallel b$?

To compute a projection $r \parallel b$ we first compute the value of b.

constructor If the value of b is a constructor i then we compute the value of r. If r computes to a record then we can use the following reductions and continue to compute the result of the reduction.

$$[i = c; d] \parallel i \longrightarrow c$$

$$[j = c; d] \parallel i \longrightarrow d \parallel i \quad \text{if } i \neq j$$

$$[] \parallel i \longrightarrow i$$

If the value v of r is not a record, then we finish the computation and return the value $v \parallel i$.

application If the value of b is an application $(c\ d)$ then we perform the following reduction

$$r \parallel (c\ d) \longrightarrow (r \parallel c\ r \parallel d)$$

and continue the computation.

abstraction If the value of b is an abstraction $\lambda x.c$:

$$r \parallel (\lambda x.c) \longrightarrow \lambda x.(r \parallel c)$$

record If the value of b is a defi nition list then we perform the reductions

$$r \parallel [] \longrightarrow []$$

$$r \parallel [i = c; d] \longrightarrow [i = r \parallel c; r \parallel d]$$

projection If the value of b is a projection $c \parallel d$ then we perform the reduction

$$r \parallel (c \parallel d) \longrightarrow (r \parallel c) \parallel (r \parallel d)$$

and continue the computation.