A Higher-Order Polymorphic Lambda-Calculus With Sized Types

This is where the subtitle would have gone.

Andreas Abel

Slide 1

First APPSEM II Workshop
Nottingham, UK
March 28, 2003

— Work in progress —

Work supported by: PhD Prg. Logic in Computer Science, Munich (DFG)

Setting the stage...

Slide 2

- Curry-Howard-Isomorphism:
  proofs by induction = programs with recursion
- Only *terminating* programs constitute valid proofs.
- Design issue: How to integrate terminating recursion into proof/programming language?

One approach: special forms of recursion

Slide 3

- Tame recursion by restricting to special patterns.
- Iteration/catamorphisms
  e.g. Haskell's `List.fold`
- Primitive recursion/paramorphisms
- Problems:
  - Non-trivial operational semantics makes it harder to understand programs.
  - I do not want to write all of my list-processing functions using `fold`.

Another approach: recursion with termination checking

Slide 4

- Use *general recursion*: `letrec`.
- Has "intuitive" meaning through simple operational semantics.
- In general not normalizing, need termination checking.
- Here we used the *sized types* approach [Hughes et al. 1996] [Barthe et al. 2003?].
- View data as trees.
- *Size* = height = # constructors in longest path of tree.
- Height of input data must decrease in each recursive call.
- Termination is ensured by type-checker.

Sized types in a nutshell

Slide 5

- Sizes are *upper bounds*.
- List^a denotes lists of length $< a$.
- List^∞ denotes list of arbitrary (but finite) length.
- Sizes induce *subtyping*: $\mathsf{List}^a \leq \mathsf{List}^b$ if $a \leq b$.
- In general, sizes are *ordinal numbers*, needed e.g. for infinitely branching trees.
- Size expressions:

$$\begin{array}{rcll} a & ::= & i & \text{variable} \\ & \mid & a+1 & \text{sucessor} \\ & \mid & \infty & \text{ultimate limit, denoting } \Omega \text{ (first uncountable)} \end{array}$$

Example: list splitting

Slide 6

$$\begin{array}{ll} \mathsf{split} : & \forall A{:}{*}.\ \mathsf{List}\ A \to \mathsf{List}\ A \times \mathsf{List}\ A \end{array}$$

$$\begin{array}{lll} \mathsf{split}\ [\,] & = \langle [\,] \quad , [\,] \quad \rangle & \\ \mathsf{split}\ (x :: k\ ) & = \mathsf{case}\ k \quad \mathsf{of} & \\ & \quad [\,] \qquad\quad \to \langle (x :: k) \quad , [\,] \quad \rangle & \\ & \mid (y :: l\ ) \to \mathsf{let}\ \langle xs\ , ys\ \rangle = \mathsf{split}\, l\ \ \mathsf{in} & \\ & \qquad\qquad\quad \langle (x :: xs) \quad , (y :: ys) \quad \rangle & \end{array}$$

- Sized types allow us to express that split denotes a non-size increasing function.

Example: list splitting

Slide 7

$$
\begin{array}{l}
\mathsf{split} : \forall i{:}\mathsf{ord}.\,\forall A{:}*.\ \mathsf{List}^{i} A \to \mathsf{List}\ A \times \mathsf{List}\ A \\
\mathsf{split}\ [\,] \qquad\qquad = \langle [\,] \quad, [\,] \quad \rangle \\
\mathsf{split}\ (x :: k^{i})^{i+1} = \mathsf{case}\ k^{i \le i+1}\ \mathsf{of} \\
\qquad\qquad\qquad\quad [\,] \qquad\quad \to \langle (x :: k) \quad , [\,] \quad \rangle \\
\qquad\qquad\qquad\mid (y :: l^{i}) \to \mathsf{let}\ \langle xs\ , ys\ \rangle = \mathsf{split}\, l^{i}\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \langle (x :: xs) \quad , (y :: ys) \quad \rangle
\end{array}
$$

- To compute split at stage $i+1$, split is only used at stage i.
- Hence, split is terminating.

Example: list splitting

Slide 8

$$
\begin{array}{l}
\mathsf{split} : \forall i{:}\mathsf{ord}.\,\forall A{:}*.\ \mathsf{List}^{i} A \to \mathsf{List}^{i} A \times \mathsf{List}^{i} A \\
\mathsf{split}\ [\,]^{i+1} \qquad = \langle [\,]^{i+1}, [\,]^{i+1} \rangle \\
\mathsf{split}\ (x :: k^{i})^{i+1} = \mathsf{case}\ k^{i \le i+1}\ \mathsf{of} \\
\qquad\qquad\qquad\quad [\,]^{i+1} \quad \to \langle (x :: k)^{i+1}, [\,]^{i+1} \rangle \\
\qquad\qquad\qquad\mid (y :: l^{i}) \to \mathsf{let}\ \langle xs^{i}, ys^{i} \rangle = \mathsf{split}\, l^{i}\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \langle (x :: xs)^{i+1}, (y :: ys)^{i+1} \rangle
\end{array}
$$

- We additionally can infer that split is non-size increasing.
- Using split, we can define merge sort…

Example: merge sort

Slide 9

$$\begin{array}{l}
\textsf{merge} : \quad \textsf{List}\ \textsf{Int} \to \quad \textsf{List}\ \textsf{Int} \to \textsf{List}\ \ \textsf{Int} \\[1ex]
\textsf{msort} : \quad \textsf{List}\ \textsf{Int} \to \textsf{List}\ \ \textsf{Int} \\[1ex]
\textsf{msort}\ [\,] \qquad = [\,] \\
\textsf{msort}\ (x :: k\ ) = \textsf{case}\ k \quad \textsf{of} \\
\qquad\qquad\qquad\qquad [\,] \qquad \to x :: [\,] \\
\qquad\qquad\qquad\quad \mid (y :: l\ ) \to \textsf{let}\ (xs\ \ , ys\ \ ) = \textsf{split}\, l\ \ \textsf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textsf{merge}\ (\textsf{msort}\ (x :: xs) \qquad) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textsf{msort}\ (y :: ys) \qquad)
\end{array}$$

Example: merge sort

Slide 10

$$\begin{array}{l}
\textsf{merge} : \forall i : \textsf{ord}.\ \textsf{List}^{i}\ \textsf{Int} \to \forall j : \textsf{ord}.\ \textsf{List}^{j}\ \textsf{Int} \to \textsf{List}^{\infty}\ \textsf{Int} \\[1ex]
\textsf{msort} : \forall i : \textsf{ord}.\ \textsf{List}^{i}\ \textsf{Int} \to \textsf{List}^{\infty}\ \textsf{Int} \\[1ex]
\textsf{msort}\ [\,]^{i+1} \qquad = [\,] \\
\textsf{msort}\ (x :: k^{i}) = \textsf{case}\ k^{j+1=i}\ \textsf{of} \\
\qquad\qquad\qquad\qquad [\,] \qquad \to x :: [\,] \\
\qquad\qquad\qquad\quad \mid (y :: l^{j}) \to \textsf{let}\ (xs^{j}, ys^{j}) = \textsf{split}\, l^{j}\ \textsf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textsf{merge}\ (\textsf{msort}\ (x :: xs)^{j+1=i}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textsf{msort}\ (y :: ys)^{j+1=i})
\end{array}$$

F^ω: smoothing the presentation

Slide 11

- Kinds.

$$\begin{array}{rcll} \kappa & ::= & * & \text{types} \\ & | & \mathsf{ord} & \text{ordinal sizes} \\ & | & \kappa \xrightarrow{+} \kappa' & \text{covariant type constructors} \\ & | & \kappa \xrightarrow{-} \kappa' & \text{contravariant type constructors} \\ & | & \kappa \xrightarrow{0} \kappa' & \text{invariant type constructors} \end{array}$$

- "Subconstructors" $F \leq G : \kappa$. E.g.,

$$\frac{X{\leq}Y : \kappa \vdash F\,X \leq G\,Y : \kappa'}{F \leq G : \kappa \xrightarrow{+} \kappa'}$$

- Well-kindedness definable by $F : \kappa \iff F \leq F : \kappa$

Inductive types

Slide 12

- Inductive constructors.

$$\mu_\kappa : \mathsf{ord} \xrightarrow{+} (\kappa \xrightarrow{+} \kappa) \xrightarrow{+} \kappa$$

- Example: $\mathsf{List} = \lambda i \lambda A.\ \mu_* i\,(\lambda X.\, 1 + A \times X)$.
- Axiom: Fixpoint is reached at stage ∞.

$$\mu\,a \leq \mu\,\infty : (\kappa \xrightarrow{+} \kappa) \xrightarrow{+} \kappa$$

- Recursion over inductive types:

$$\frac{\begin{array}{l} F : * \xrightarrow{+} * \\ G : \mathsf{ord} \xrightarrow{+} * \\ i : \mathsf{ord} \vdash s : (\mu\,i\,F \to G\,i) \to \mu\,(i+1)\,F \to G\,(i+1) \end{array}}{\mathsf{fix}^\mu\ s : \forall i{:}\mathsf{ord}.\, \mu\,i\,F \to G\,i}$$

Higher-rank inductive types

- Inductive functors: μ_κ for $\kappa = * \to *$.
- E.g., $\mathsf{Term}\, A$, de Bruijn terms with free variables in A:

$$\mathsf{Term} = \mu_{*\to*}\infty \lambda T \lambda A.\, A + T(1 + A) + TA \times TA$$

Slide 13

Conclusions

Sized types:

- Conceptually *lean* way of ensuring termination.
- Well-typedness ensures termination.
- No external static analysis required.

Slide 14

System F^ω:

- Size expressions can be integrated into constructors.
- Sized types scale to higher-order polymorphism.

Goal: extend to dependent types.