A Higher-Order Polymorphic Lambda-Calculus With Sized Types

This is where the subtitle would have gone.

Andreas Abel

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...

- 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

- 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

- 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

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

\[
\begin{align*}
a & ::= \quad \text{variable} \\
& \mid a + 1 \quad \text{successor} \\
& \mid \infty \quad \text{ultimate limit, denoting } \Omega \text{ (first uncountable)}
\end{align*}
\]

Example: list splitting

\[
\begin{align*}
split : \quad & \forall A : \ast, \text{List } A \rightarrow \text{List } A \times \text{List } A \\
split [ ] & = ([], []) \\
split (x :: k) & = \text{case } k \text{ of} \\
& \quad [ ] \rightarrow ([x :: k], []) \\
& \quad (y :: l) \rightarrow \text{let } (xs, ys) = \text{split } l \text{ in} \\
& \quad \quad ([x :: xs], (y :: ys))
\end{align*}
\]

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

\[ \text{split} : \forall i:\text{ord} \forall A : *, \text{List}^i A \to \text{List}^i A \times \text{List}^i A \]
\[ \text{split} \ [\ ] = ( [\ ] , [\ ] ) \]
\[ \text{split} \ (x :: k)^{i+1} = \text{case } k^{i \leq i+1} \text{ of} \]
\[ \quad [\ ] \to ( (x :: k) ^{i+1} , [\ ] ^{i+1} ) \]
\[ \quad (y :: l)^{i+1} \to \text{let } (xs^i , ys^i) = \text{split} l^i \text{ in} \]
\[ \quad ( (x :: xs)^{i+1} , (y :: ys)^{i+1} ) \]

- To compute \( \text{split} \) at stage \( i + 1 \), \( \text{split} \) is only used at stage \( i \).
- Hence, \( \text{split} \) is terminating.

Example: list splitting

\[ \text{split} : \forall i:\text{ord} \forall A : *, \text{List}^i A \to \text{List}^i A \times \text{List}^i A \]
\[ \text{split} \ [\ ]^{i+1} = ( [\ ]^{i+1} , [\ ]^{i+1} ) \]
\[ \text{split} \ (x :: k)^{i+1} = \text{case } k^{i \leq i+1} \text{ of} \]
\[ \quad [\ ]^{i+1} \to ( (x :: k)^{i+1} , [\ ]^{i+1} ) \]
\[ \quad (y :: l)^{i+1} \to \text{let } (xs^i , ys^i) = \text{split} l^i \text{ in} \]
\[ \quad ( (x :: xs)^{i+1} , (y :: ys)^{i+1} ) \]

- We additionally can infer that \( \text{split} \) is non-size increasing.
- Using \( \text{split} \), we can define merge sort...
Example: merge sort

merge: \text{List \ Int} \rightarrow \text{List \ Int} \rightarrow \text{List \ Int}

msort: \text{List \ Int} \rightarrow \text{List \ Int}

msort [] = []

msort (x :: k) = case k of
  [] \rightarrow x :: []
  | (y :: l) \rightarrow \text{let} (xs, ys) = \text{split} l \text{ in}
      \begin{align*}
      \text{merge} & (\text{msort} (x :: xs) ) \\
      \text{(msort} & (y :: ys))
    \end{align*}

Slide 10

Example: merge sort

merge: \forall i. \text{List}^i \text{ Int} \rightarrow \forall j. \text{List}^j \text{ Int} \rightarrow \text{List}^\infty \text{ Int}

msort: \forall i. \text{List}^i \text{ Int} \rightarrow \text{List}^\infty \text{ Int}

msort [i + 1] = []

msort (x :: k) = case k^j+1=i of
  [] \rightarrow x :: []
  | (y :: l) \rightarrow \text{let} (xs^j, ys^j) = \text{split} l^j \text{ in}
      \begin{align*}
      \text{merge} & (\text{msort} (x :: xs))^{j+1=i} \\
      \text{(msort} & (y :: ys))^{j+1=i}
    \end{align*}
$F^\omega$: smoothing the presentation

- Kinds.
  \[ \kappa ::= \ast \quad \text{types} \]
  \[ | \text{ord} \quad \text{ordinal sizes} \]
  \[ | \kappa \rightarrow^+ \kappa' \quad \text{covariant type constructors} \]
  \[ | \kappa \rightarrow^- \kappa' \quad \text{contravariant type constructors} \]
  \[ | \kappa \rightarrow^0 \kappa' \quad \text{invariant type constructors} \]

  \text{“Subconstructors”} \quad F \leq G : \kappa. \text{E.g.,}

  \[ X \leq Y : \kappa \vdash FX \leq GY : \kappa' \]
  \[ \frac{}{F \leq G : \kappa \rightarrow^+ \kappa'} \]

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

Inductive types

- Inductive constructors.
  \[ \mu : \text{ord} \rightarrow^+ (\kappa \rightarrow^+ \kappa) \rightarrow^+ \kappa \]

- Example: List = $\lambda i \lambda A. \mu_i (\lambda X. 1 + A \times X)$.

- Axiom: Fixpoint is reached at stage $\infty$.

  \[ \mu a \leq \mu \infty : (\kappa \rightarrow^+ \kappa) \rightarrow^+ \kappa \]

- Recursion over inductive types:

  \[ F : \ast \rightarrow^+ \ast \]
  \[ G : \text{ord} \rightarrow^+ \ast \]
  \[ i : \text{ord} \vdash s : (\mu i F \rightarrow G i) \rightarrow (\mu (i + 1) F \rightarrow G (i + 1)) \]
  \[ \frac{}{\text{fix}^\mu s : \forall i : \text{ord}. \mu i F \rightarrow G i} \]
Higher-rank inductive types

- Inductive functors: $\mu_\kappa$ for $\kappa = * \to *$.
- E.g., Term $A$, de Bruijn terms with free variables in $A$:
  \[
  \text{Term} = \mu_{\kappa \to \omega} \lambda T \lambda A. A + T(1 + A) + TA \times TA
  \]

Conclusions

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

System $F^\omega$:
- Size expressions can be integrated into constructors.
- Sized types scale to higher-order polymorphism.

Goal: extend to dependent types.