Parameters in Pure Type Systems

Roel Bloo (Eindhoven University of Technology, NL)
Fairouz Kamareddine (Heriot-Watt University, Edinburgh)
Twan Laan and Rob Nederpelt (Eindhoven University of Technology, NL)

5 April 2001

LATIN’02, APRIL 2002, Cancun, Mexico
The Low Level approach of functions

- Historically, functions have long been treated as a kind of meta-objects.

- Function values have always been important, but abstract functions have not been recognised in their own right until the third of the 20th century.

- In the low level approach or operational view on functions, there are no functions as such, but only function values.

- E.g., the sine-function, is always expressed together with a value: \( \sin(\pi) \), \( \sin(x) \) and properties like: \( \sin(2x) = 2\sin(x)\cos(x) \).

- It has long been usual to call \( f(x) \)—and not \( f \)—the function and this is still the case in many introductory mathematics courses.
The revolution of treating functions as first class citizens

- In the nowadays accepted view on functions, they are ‘first class citizens’.
- Abstraction and application form the basis of the $\lambda$-calculus and type theory.
- This is rigid and does not represent the development of logic in 20th century.
- Frege and Russell’s conceptions of functional abstraction, instantiation and application do not fit well with the $\lambda$-calculus approach.
- In *Principia Mathematica* [Whitehead and Russell, 1910$^1$, 1927$^2$]: If, for some $a$, there is a proposition $\phi a$, then there is a function $\phi x$, and vice versa.
- The function $\phi$ is not a separate entity but always has an argument.
\textbf{\textit{\lambda}-calculus does not fully represent functionalisation}

1. Abstraction from a subexpression $2 + 3 \mapsto x + 3$

2. Function construction $x + 3 \mapsto \lambda x \cdot x + 3$

3. Application construction $(\lambda x \cdot (x + 3))2$

4. Concretisation to a subexpression $(\lambda x \cdot (x + 3))2 \rightarrow 2 + 3$

- Cannot identify the original term from which a function has been abstracted.
  
  \[
  \text{let add}_2 = (\lambda x \cdot x + 2) \text{ in } \text{add}_2(x) + \text{add}_2(y)
  \]

- cannot abstract only half way: $x + 3$ is not a function, $\lambda x \cdot x + 3$ is.

- cannot apply $x + 3$ to an argument: $(x + 3)2$ does not evaluate to $2 + 3$. 
Parameters: What and Why

• we speak about *functions with parameters* when referring to functions with variable values in the *low-level* approach. The $x$ in $f(x)$ is a parameter.

• Parameters enable the same expressive power as the high-level case, while allowing us to stay at a lower order. Cf. [Laan and Franssen, 2001] and [Kamareddine et al., 2001].

• Desirable properties of the lower order theory (*decidability, easiness of calculations, typability*) can be maintained, without losing the flexibility of the higher-order aspects.

• This *low-level approach* is still worthwhile for many exact disciplines. It has not been wiped out in *logic* and in computer science, and for good reasons.
Automath

- The first tool for mechanical representation and verification of mathematical proofs, **AUTOMATH**, has a parameter mechanism.

- The representation of a **mathematical text** in **AUTOMATH** consists of a finite list of **lines** where every line has the following format:

\[ x_1 : A_1, \ldots, x_n : A_n \vdash g(x_1, \ldots, x_n) = t : T. \]

Here \( g \) is a new name, an abbreviation for the expression \( t \) of type \( T \) and \( x_1, \ldots, x_n \) are the parameters of \( g \), with respective types \( A_1, \ldots, A_n \).

- Each line introduces a new definition which is inherently parametrised by the variables occurring in the context needed for it.
• Developments of ordinary mathematical theory in **AUTOMATH** [Benthem Jutting, 1977] revealed that this combined definition and parameter mechanism is vital for keeping proofs manageable and sufficiently readable for humans.
The Barendregt Cube

- \( \mathcal{T}_P ::= \nu \mid S \mid \mathcal{T}_P \mathcal{T}_P \mid \lambda\nu : \mathcal{T}_P . \mathcal{T}_P \mid \Pi\nu : \mathcal{T}_P . \mathcal{T}_P \)

- \( \mathcal{V} \) is a set of variables and \( S = \{*, \Box\} \).
(axiom) \[ \langle \rangle \vdash \ast : \Box \]

:start) \[ \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad x \notin \text{DOM} (\Gamma) \]

(weak) \[ \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \quad x \notin \text{DOM} (\Gamma) \]

(\Pi) \[ \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A. B) : s_2} \quad (s_1, s_2) \in R \]

(\lambda) \[ \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A. B) : s}{\Gamma \vdash (\lambda x : A. b) : (\Pi x : A. B)} \]

(appl) \[ \frac{\Gamma \vdash F : (\Pi x : A. B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x:=a]} \]

(conv) \[ \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_{\beta} B'}{\Gamma \vdash A : B'} \]
Different type formation conditions

- $(\Pi) \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x:A \vdash B : s_2}{\Gamma \vdash (\Pi x:A. B) : s_2} \quad (s_1, s_2) \in R$

- $(\Box, \ast)$ takes care of polymorphism. $\lambda 2$ is weakest on cube satisfying this.

- $(\Box, \Box)$ takes care of type constructors. $\lambda \omega$ is weakest on cube satisfying this.

- $(\ast, \Box)$ takes care of term dependent types. $\lambda P$ is weakest on cube satisfying this.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda \rightarrow$</td>
<td>(*, *)</td>
<td>(□, *)</td>
<td>(*, □)</td>
<td>(□, □)</td>
</tr>
<tr>
<td>$\lambda 2$</td>
<td>(*, *)</td>
<td>(□, *)</td>
<td>(*, □)</td>
<td>(□, □)</td>
</tr>
<tr>
<td>$\lambda P$</td>
<td>(*, *)</td>
<td>(□, *)</td>
<td>(*, □)</td>
<td>(□, □)</td>
</tr>
<tr>
<td>$\lambda \omega$</td>
<td>(*, *)</td>
<td>(□, *)</td>
<td>(*, □)</td>
<td>(□, □)</td>
</tr>
<tr>
<td>$\lambda P2$</td>
<td>(*, *)</td>
<td>(□, *)</td>
<td>(*, □)</td>
<td>(□, □)</td>
</tr>
<tr>
<td>$\lambda \omega$</td>
<td>(*, *)</td>
<td>(□, *)</td>
<td>(*, □)</td>
<td>(□, □)</td>
</tr>
<tr>
<td>$\lambda P \omega$</td>
<td>(*, *)</td>
<td>(□, *)</td>
<td>(*, □)</td>
<td>(□, □)</td>
</tr>
<tr>
<td>$\lambda C$</td>
<td>(*, *)</td>
<td>(□, *)</td>
<td>(*, □)</td>
<td>(□, □)</td>
</tr>
</tbody>
</table>
Systems of the Barendregt Cube
<table>
<thead>
<tr>
<th>System</th>
<th>Rel. system</th>
<th>Names, references</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda\rightarrow)</td>
<td>(\lambda^T)</td>
<td>simply typed (\lambda)-calculus; [Church, 1940], [Barendregt, 1984] (Appendix A), [Hindley and Seldin, 1986] (Chapter 14)</td>
</tr>
<tr>
<td>(\lambda^2)</td>
<td>(F)</td>
<td>second order typed (\lambda)-calculus; [Girard, 1972], [Reynolds, 1974]</td>
</tr>
<tr>
<td>(\lambda_p)</td>
<td>AUT-QE</td>
<td>[Bruijn, 1968]</td>
</tr>
<tr>
<td></td>
<td>LF</td>
<td>[Harper et al., 1987]</td>
</tr>
<tr>
<td>(\lambda_{p2})</td>
<td>POLYREC</td>
<td>[Longo and Moggi, 1988]</td>
</tr>
<tr>
<td>(\lambda_\omega)</td>
<td>POLYREC</td>
<td>[Renardel de Lavalette, 1991]</td>
</tr>
<tr>
<td>(\lambda_\omega)</td>
<td>(F_\omega)</td>
<td>[Girard, 1972]</td>
</tr>
<tr>
<td>(\lambda_C)</td>
<td>CC</td>
<td>Calculus of Constructions; [Coquand and Huet, 1988]</td>
</tr>
</tbody>
</table>
The Barendregt Cube

\((\square, \ast) \in R\)

\((\square, \square) \in R\)

\((\ast, \square) \in R\)
• LF (see [Harper et al., 1987]) is often described as $\lambda P$ of the Barendregt Cube.

• [Geuvers, 1993] shows that the use of the $\Pi$-formation rule $(\ast, \square)$ is very restricted in the practical use of LF.

• This use is in fact based on a parametric construct rather than on $\Pi$-formation.

• We will find a more precise position of LF on the Cube (between $\lambda \rightarrow$ and $\lambda P$).
• We only consider an explicit version of a subset of ML.

• In ML, one can define the polymorphic identity by:

\[ \text{Id}(\alpha:* ) = (\lambda x:\alpha . x) : (\alpha \to \alpha) \]  \hspace{1cm} (1)

• But in ML, it is not possible to make an explicit \( \lambda \)-abstraction over \( \alpha : * \) by:

\[ \text{Id} = (\lambda \alpha : * . \lambda x:\alpha . x) : (\Pi \alpha : * . \alpha \to \alpha) \]  \hspace{1cm} (2)

• The type \( \Pi \alpha : * . \alpha \to \alpha \) does not belong to the language of ML and hence the \( \lambda \)-abstraction of equation (2) is not possible in ML.
ML

- Therefore, we can state that **ML does not have a $\Pi$-formation rule ($\Box$, $\star$).**
- Nevertheless, **ML has some parameter mechanism ($\alpha$ parameter of $\text{Id}$)**
- **ML has limited access to the rule ($\Box$, $\star$) enabling equation (1) to be defined.**
- **ML’s type system is none of those of the eight systems of the Cube.**
- **We place the type system of ML on our refined Cube (between $\lambda 2$ and $\lambda \omega$).**
Extending PTSs with parameters and definitions

Figure 1: The hierarchy of parameters, constants and definitions
\[\mathcal{T}_P ::= \mathcal{V} \mid S \mid \mathcal{C}(\mathcal{L}_T) \mid (\mathcal{T}_P \mathcal{T}_P) \mid (\lambda \mathcal{V}: \mathcal{T}_P. \mathcal{T}_P) \mid (\Pi \mathcal{V}: \mathcal{T}_P. \mathcal{T}_P) \mid (\mathcal{C}(\mathcal{L}_V) = \mathcal{T}_P : \mathcal{T}_P \text{ in } \mathcal{T}_P);\]

- Parametric constructs are \(c(b_1, \ldots, b_n)\) with \(b_1, \ldots, b_n\) terms of certain types. \(\mathcal{C}\) is a set of constants, \(b_1, \ldots, b_n\) are called the parameters of \(c(b_1, \ldots, b_n)\).

- \(\mathcal{R}\) allows several kinds of \(\Pi\)-constructs. We also use a set \(\mathcal{P}\) of \((s_1, s_2)\) where \(s_1, s_2 \in \{*, \Box\}\) to allow several kinds of parametric constructs.

- \((s_1, s_2) \in \mathcal{P}\) means that we allow parametric constructs \(c(b_1, \ldots, b_n) : A\) where \(b_1, \ldots, b_n\) have types \(B_1, \ldots, B_n\) of sort \(s_1\), and \(A\) is of type \(s_2\).
• If both \((*, s_2) \in \mathbf{P}\) and \((\Box, s_2) \in \mathbf{P}\) then combinations of parameters allowed. For example, it is allowed that \(B_1\) has type \(*\), whilst \(B_2\) has type \(\Box\).
(δ1): \[ \Gamma, c(\Delta) = a : A, \Gamma_2 \vdash c(b_1, \ldots, b_n) \rightarrow_\delta a[x_i := b_i]_{i=1}^n \]

(δ2): \[ \frac{c \notin \text{CONS}(b)}{\Gamma \vdash c(\Delta) = a : A \text{ in } b \rightarrow_\delta b} \]

(δ3): \[ \frac{\Gamma, c(\Delta) = a : A \vdash b \rightarrow_\delta b'}{\Gamma \vdash c(\Delta) = a : A \text{ in } b \rightarrow_\delta c(\Delta) = a : A \text{ in } b'} \]

\[ \begin{array}{l}
\quad \Gamma, \Delta \vdash a \rightarrow_\delta a' \\
\quad \Gamma \vdash c(\Delta) = a : A \text{ in } b \rightarrow_\delta c(\Delta) = a' : A \text{ in } b
\end{array} \]

\[ \begin{array}{l}
\quad \Gamma, \Delta \vdash A \rightarrow_\delta A' \\
\quad \Gamma \vdash c(\Delta) = a : A \text{ in } b \rightarrow_\delta c(\Delta) = a' : A \text{ in } b
\end{array} \]

\[ \Gamma, \Delta_i \vdash B_i \rightarrow_\delta B_i' \]

\[ \Gamma \vdash c(\Delta) = a : A \text{ in } b \rightarrow_\delta c(x_1 : B_1, \ldots, x_i : B_i', \ldots, x_n : B_n) = a : A \text{ in } b \]

\[ \begin{array}{l}
\quad \Gamma \vdash a \rightarrow_\delta a' \\
\quad \Gamma \vdash ab \rightarrow_\delta a'b
\end{array} \]

\[ \begin{array}{l}
\quad \Gamma \vdash b \rightarrow_\delta b' \\
\quad \Gamma \vdash ab \rightarrow_\delta ab'
\end{array} \]

\[ \begin{array}{l}
\quad \Gamma, x : A \vdash a \rightarrow_\delta a' \\
\quad \Gamma \vdash \lambda x : A. a \rightarrow_\delta \lambda x : A. a'
\end{array} \]

\[ \begin{array}{l}
\quad \Gamma \vdash \lambda x : A. a \rightarrow_\delta \lambda x : A'. a
\end{array} \]

\[ \begin{array}{l}
\quad \Gamma, x : A \vdash a \rightarrow_\delta a' \\
\quad \Gamma \vdash \Pi x : A. a \rightarrow_\delta \Pi x : A. a'
\end{array} \]

\[ \begin{array}{l}
\quad \Gamma \vdash \Pi x : A. a \rightarrow_\delta \Pi x : A'. a
\end{array} \]

\[ \begin{array}{l}
\quad \Gamma \vdash a_j \rightarrow_\delta a'_j
\end{array} \]

\[ \begin{array}{l}
\quad \Gamma \vdash c(a_1, \ldots, a_n) \rightarrow_\delta c(a_1, \ldots, a'_j, \ldots, a_n)
\end{array} \]
\( \vec{\vdash} \text{-weak} \)

\[
\frac{\Gamma \vdash \vec{C} b : B, \Delta \vdash \vec{C} A : s, \Gamma, \Delta_i \vdash \vec{C} B_i : s_i \quad (s_i, s) \in P \quad (i = 1, \ldots, n)}{\Gamma, c(\Delta) : A \vdash \vec{C} b : B}
\]

\[
\frac{\Gamma_1, c(\Delta) : A, \Gamma_2 \vdash \vec{C} b_i : B_i[x_j := b_j]_{j=1}^{i-1} \quad (i = 1, \ldots, n)}{\Gamma_1, c(\Delta) : A, \Gamma_2 \vdash \vec{C} A : s \quad (\text{if } n = 0)}
\]

\( \vec{\vdash} \text{-app} \)

\[
\frac{\Gamma_1, c(\Delta) : A, \Gamma_2 \vdash \vec{C} c(b_1, \ldots, b_n) : A[x_j := b_j]_{j=1}^{n}}{\Gamma_1, c(\Delta) : A, \Gamma_2 \vdash \vec{C} c(b_1, \ldots, b_n) : A[x_j := b_j]_{j=1}^{n}}
\]

Figure 2: Typing rules for parametric constants
Bloo, Kamareddine, Laan and Nederpelt
<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D-weak)</td>
<td>( \Gamma \vdash^D b : B \quad \Gamma, \Delta \vdash^D a : A : s \quad \Gamma, \Delta_i \vdash^D B_i : s_i ) ((s_i, s) \in P) ((i = 1, \ldots, n))</td>
</tr>
<tr>
<td></td>
<td>( \Gamma, c(\Delta) = a : A \vdash^D b : B )</td>
</tr>
<tr>
<td></td>
<td>( \Gamma_1, c(\Delta) = a : A, \Gamma_2 \vdash^D b_i : B_i[x_j := b_j]_{j=1}^{i-1} ) ((i = 1, \ldots, n))</td>
</tr>
<tr>
<td>(D-app)</td>
<td>( \Gamma_1, c(\Delta) = a : A, \Gamma_2 \vdash^D a : A ) ((\text{if } n = 0))</td>
</tr>
<tr>
<td></td>
<td>( \Gamma_1, c(\Delta) = a : A, \Gamma_2 \vdash^D c(b_1, \ldots, b_n) : A[x_j := b_j]_{j=1}^{n} )</td>
</tr>
<tr>
<td>(D-form)</td>
<td>( \Gamma, c(\Delta) = a : A \vdash^D B : s )</td>
</tr>
<tr>
<td></td>
<td>( \Gamma \vdash^D c(\Delta) = a : A \text{ in } B : s )</td>
</tr>
<tr>
<td>(D-intro)</td>
<td>( \Gamma, \Delta \vdash^D b : B \quad \Gamma \vdash^D c(\Delta) = a : A \text{ in } B : s )</td>
</tr>
<tr>
<td></td>
<td>( \Gamma \vdash^D c(\Delta) = a : A \text{ in } b : c(\Delta) = a : A \text{ in } B )</td>
</tr>
<tr>
<td>(D-conv)</td>
<td>( \Gamma \vdash^D b : B \quad \Gamma \vdash^D B' : s \quad \Gamma \vdash B =_\delta B' )</td>
</tr>
<tr>
<td></td>
<td>( \Gamma \vdash^D b : B' )</td>
</tr>
</tbody>
</table>

Figure 3: Typing rules for parametric definitions
Conclusions

- Parameters enable the same expressive power as the high-level case, while allowing us to stay at a lower order. E.g. first-order with parameters versus second-order without [Laan and Franssen, 2001].

- Desirable properties of the lower order theory (decidability, easiness of calculations, typability) can be maintained, without losing the flexibility of the higher-order aspects.

- Parameters enable us to find an exact position of type systems in the generalised framework of type systems.

- Parameters describe the difference between developers and users of systems.
Bibliography


Bloo, Kamareddine, Laan and Nederpelt


