The parametrization Principle
Programming Languages CS442

David Toman

School of Computer Science
University of Waterloo
The Parametrization Principle

Semantically meaningful phrases can serve as parameters.

Definition: \textbf{define} \( l_1(l_2 : \theta) = V \)

Invocation: \textbf{invoke} \( l_1(U) \)

\( \Rightarrow \) We allow only \textit{parametrized abstractions} so far

\( \ldots \) but technically, all valid phrases can be parametrized
Typing Rules

for definitions:

\[ \pi \cup \{ l_2 : \theta_1 \} \vdash V : \theta_2 \]

\[ \pi \vdash \text{define } l_1(l_2 : \theta_1) = V : \{ l_1 : \theta_1 \to \theta_2 \} \text{dec} \]

Do we need the type attribute \( \theta_1 \) for \( l_2 \)?
Could we “typecheck” at “invocation” places?

for invocations:

\[ \pi \vdash U : \theta_1 \]

\[ \pi \vdash \text{invoke } l(U) : \theta_2 \quad \text{if } (l : \theta_1 \to \theta_2) \in \pi \]
Expression Parameters

Extra syntax: 
\[ D ::= \ldots \mid \textbf{proc} \ P(I: \tau \exp) = C \quad \tau \in \{\text{int, bool}\} \]
\[ C ::= \ldots \mid \textbf{call} \ P(E) \]
\[ E ::= \ldots \mid I \]

When to evaluate \( E \)?

\textbf{CBV} (call-by-value)

evaluate \( E \) to \( v \), then substitute \( v \) for \( I \) and evaluate \( P \)

\textbf{CBN} (call-by-name)

substitute \( E \) for \( I \) and evaluate \( P \)

\[ \ldots \Rightarrow \text{just like lazy/eager abstractions!} \]
Expression Parameters: semantics

Definition:

\[
\llbracket \pi \vdash \texttt{proc } P(l: \tau \exp = C) \rrbracket e \ s = (\{l = p\}, s)
\]

where \( p \ v \ s = \llbracket \pi \cup \{l: \tau \exp \vdash C\}(e \cup \{l = v\}) \rrbracket s \)

Invocation:

CBV

\[
\llbracket \pi \vdash \texttt{call } P(E) : \texttt{comm} \rrbracket e \ s = p \ (\llbracket \pi \vdash E : \tau \exp \rrbracket e \ s) \ s \quad (P = p) \in e
\]

\[
\llbracket \pi \vdash l : \tau \exp \rrbracket e \ s = v \quad (l = v) \in e
\]

CBN

\[
\llbracket \pi \vdash \texttt{call } P(E) : \texttt{comm} \rrbracket e \ s = p \ (\llbracket \pi \vdash E : \tau \exp \rrbracket e) \ s \quad (P = p) \in e
\]

\[
\llbracket \pi \vdash l : \tau \exp \rrbracket e \ s = f \ s \quad (l = f) \in e
\]
Can we use a “copy” rule like for lazy abstractions?

yes—but we’d like to reuse the “copy rule” for abstractions

Idea

Make parametrized phrases into valid syntactic objects

⇒ needs syntax for this is the parameter

Idea

Replace “define $P(I : \theta) = V$” with “define $P = \lambda I : \theta. V$”.

Copy Rules:

for abstraction same as for lazy abstractions

for parameter(s) $(\lambda I : \theta. U) \ V \Rightarrow [V/I]U$
Other “standard” Parameters

Commands

lazy parameters (commands passed using a “pointer”?)
What would eagerly evaluated command parameter do?

\[
\text{proc } P(H : \text{comm}) = \ldots \text{if err then } H \text{ else } \ldots \text{fi}
\]

⇒ backtracking point!

Modules

allows bigger and modular modules.

Type Structures

comming later as dependent types…
Type Equivalence

Idea

Type attribute of the formal parameter must match the one of the actual parameter.

⇒ what does “match” mean???

• name equivalence

• structural equivalence (+ brands??)


• subtyping
The Correspondence Principle

Idea

Meaning of identifiers (with a given type attribute) should be independent on whether the identifier was bound using abstraction or as an parameter.

\[
\text{define } l = U \text{ in } V \text{ is the same as } (\lambda l. V) U
\]
parametrization and abstractions go “hand in hand”.

eager/lazy evaluation of abstractions is the same issue as CBV/CBN parameter transmission modes.

questions:

1. how would typing rules look for “type at invocation location”?
2. how about other “parameter transmission” modes?

⇒ call-by-reference? call-by-value-result?