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: **define** $I_1(I_2 : \theta) = V$

Invocation: **invoke** $I_1(U)$

$\Rightarrow$ We allow only *parametrized abstractions* so far

…but technically, all valid phrases can be parametrized

Typing Rules

for definitions:

$$\frac{\pi \uplus \{I_2 : \theta_1\} \vdash V : \theta_2}{\pi \vdash \textbf{define}\, I_1(I_2 : \theta_1) = V : \{I_1 : \theta_1 \rightarrow \theta_2\} dec}$$

Do ve need the type attribute θ_1 for I_2?

Could we “typecheck” at “invocation” places?

for invocations:

$$\frac{\pi \vdash U : \theta_1}{\pi \vdash \textbf{invoke}\, I(U) : \theta_2} \quad \textit{if} \;\; (I : \theta_1 \rightarrow \theta_2) \in \pi$$

Expression Parameters

Extra syntax: $D ::= \ldots \mid \textbf{proc}\, P(I : \tau exp) = C \qquad \tau \in \{int, bool\}$

$C ::= \ldots \mid \textbf{call}\, P(E)$

$E ::= \ldots \mid I$

When to evaluate E?

CBV (call-by-value)

evaluate E to v, then substitute v for I and evaluate P

CBN (call-by-name)

substitute E for I and evaluate P

$\ldots \Rightarrow$ just like lazy/eager abstractions!

Expression Parameters: semantics

Definition:

$$[\![\pi \vdash \textbf{proc}\, P(I : \tau exp = C]\!] e\ s = (\{I = p\}, s)$$
$$where\ p\ v\ s = [\![\pi \uplus \{I : \tau exp \vdash C]\!](e \uplus \{I = v\})\ s$$

Invocation:

CBV

$$[\![\pi \vdash \textbf{call}\, P(E) : comm]\!] e\ s = p\ ([\![\pi \vdash E : \tau exp]\!] e\ s)\ s \qquad (P = p) \in e$$
$$[\![\pi \vdash I : \tau exp]\!] e\ s = v \qquad (I = v) \in e$$

CBN

$$[\![\pi \vdash \textbf{call}\, P(E) : comm]\!] e\ s = p\ ([\![\pi \vdash E : \tau exp]\!] e)\ s \qquad (P = p) \in e$$
$$[\![\pi \vdash I : \tau exp]\!] e\ s = f\ s \qquad (I = f) \in e$$

CBN and Copy Rule

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

$\Rightarrow$ *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?

$$\textbf{proc}\, P(H : comm) = \ldots \textbf{if}\, err\ \textbf{then}\, H\ \textbf{else} \ldots \textbf{fi}$$

$\Rightarrow$ 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??)

  Chapter 8: How the Language Got its Spots
  in Greg Nelson (editor), System Programming with Modula-3, Prentice Hall, 1991.

- 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.

$$\textbf{define}\, I = U\, \textbf{in}\, V \textit{ is the same as} (\lambda I.V)U$$

Summary

- 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?

  $\Rightarrow$ call-by-reference? call-by-value-result?