Parameterization

Seminar Formal Methods 1: Type Systems

Thomas Klambauer

2009-11-30

Based on *The Structure of Typed Programming Languages* by David A. Schmidt [Sch94].

The Parameterization Principle

"Phrases from any semantically meaningful syntactic class may be parameters."

(Here only abstractions will be parameterized.)

With syntax domains U and V, define a parameterized abstraction:

$$\textbf{define } I_1(I_2 : \theta) = V \textbf{ in } \textbf{ invoke } I_1(U)$$

Typing rules

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

$$\frac{\pi \vdash U : \theta_1}{\pi \vdash \ \textbf{invoke}\ I(U) : \theta_2}\ \text{if}\ (I : \theta_1 \to \theta_2) \in \pi$$

Parameterizations

Abstraction U	Parameters V	Result
Function	Expression, Type	n-ary Function, Template Function
Procedure	Expression, Type	n-ary Proc, Template Proc
Class	Type, Numeral	Template/Generic
Module	Type	Ada Modules

Expression Parameters Evaluation

$$\textbf{var } A : \textbf{ newint } ; \textbf{ proc } P(M : \mathit{intexp}) = A := M; A := M$$
$$\textbf{in } A := 1; \textbf{ call } P(@A + 1)$$

Eager	Lazy
Call-by-value	Call-by-name

Parameter Transmission Semantics

Eager Evaluation

$$[\![\pi \vdash \ \textbf{proc}\ I_1(I_2 : \tau exp) = C : \{I_1 : \tau exp \to comm\} dec]\!]\ e\ s = (\{I_1 = p\}, s)$$
$$\text{where } p\ v\ s' = [\![\pi \uplus \{I_2 : \pi exp\} \vdash C : comm]\!](e \uplus \{I_2 = v\})s'$$

$$[\![\pi\ \textbf{call}\ I_1(E) : comm]\!]\ e\ s = p\ ([\![\pi \vdash E : \tau exp]\!](e, s))\ s$$
$$\text{where } (I_1 = p) \in e$$

$$[\![\pi \vdash I_2 : \tau exp]\!]\ e\ s = v, \text{ where } (I_2 = v) \in e$$

Parameter Transmission Semantics

Lazy Evaluation

$$[\![\pi \vdash \ \textbf{proc}\ l_1(l_2 : \tau exp) = C : \{l_1 : \tau exp \to comm\} dec]\!]\ e\ s = (\{l_1 = p\}, s)$$
$$\text{where } p\ f\ s' = [\![\pi \uplus \{l_2 : \tau exp\} \vdash C : comm]\!](e \uplus \{l_2 = f\})\ s'$$

$$[\![\pi \vdash \ \textbf{call}\ l_1(E) : comm]\!]\ e\ s = p\ f\ s \text{ where } (l_1 = p) \in e$$
$$\text{and } f\ s' = [\![\pi \vdash E : \tau exp]\!]\ e\ s'$$

$$[\![\pi \vdash l_2 : \tau exp]\!]\ e\ s = f\ s, \text{ where } (l_2 = f) \in e$$

Parameter Copy rules

Lazy evaluation

$$\textbf{define } I_1(I_2 : \tau) = U \cdots \textbf{ invoke } I_1(V)$$
$$\equiv [V/I_2]U$$

Syntax change : **define** $I_1 = \lambda I_2 : \tau.U \cdots$ **invoke** $I_1(V)$

New rule (lazy): $(\lambda I : \theta.U)V \Rightarrow [V/I]U$

New rule (eager): $(\lambda I : \theta.U)V \Rightarrow [V/I]U$ *(if V is a value)*

Lambda calculus β-Rule!

Jensen's device

```
1   begin
2      integer i;
3      real procedure sum (i, lo, hi, term);
4         value lo, hi;
5         integer i, lo, hi;
6         real term;
7         comment term is passed by-name, and so is i;
8      begin
9         real temp;
10        temp := 0;
11        for i := lo step 1 until hi do
12           temp := temp + term;
13        sum := temp
14     end;
15     comment note the correspondence between the mathematical notation and the
             call to sum;
16     print (sum (i, 1, 100, 1/i))
17  end
```

Type equivalence

Required to typecheck actual and formal parameter accordance.

Structural

$$\frac{\pi \vdash D : \pi_1 dec}{\pi \vdash \textbf{record } D \textbf{ end } : \pi_1 class} \qquad \frac{\pi \vdash D_1 : \pi_1 dec \quad \pi \cup \pi_1 \vdash D_2 : \pi_2 dec}{\pi \vdash D_1; D_2 : \pi_1 \cup \pi_2 dec}$$

Name

$$\frac{\pi \vdash D : \pi_1 dec}{\pi \vdash \textbf{record } D \textbf{ end } : occ\pi_1 class} \text{ with occ unique}$$

Type-structure Parameters

$$\textbf{module}\ M(T : \text{Type-structure})$$
$$= \{\ \textbf{var}\ A : T;\ \ \textbf{proc}\ SAVE(X : T) = A := @X\}$$
$$\cdots\ \textbf{import}\ M(\ \textbf{newint}\ )\cdots\ \textbf{in}\ \ \textbf{call}\ SAVE(A)$$

- Type parameters are values from the *semantics*
- **newint** , **record** , . . . are *storage allocators*, not *type attributes*
- M typing: $\Delta class \rightarrow \{A : \Delta, SAVE : \Delta \rightarrow comm\} dec$
- *Dependent types*

The Correspondence Principle

For every abstraction form, **define** $I_2 = U$, *there may be a corresponding parameterization form,*
define $I_1(I_2 : \theta) = \cdots$ **in** $\cdots I_1(U)$, *(and vice versa) such that the semantics of binding* I_2 *to* U *is the same in both.*

Consider

var A : **newint** ; **fun** $F = @A + 1$
proc $P(X : \mathit{intexp}) = A := X + F; A := X + F$ **in** $\ldots$

Correspondence

- Numeral parameters treated as expressions
- Constant functions? **fun** $I = N$
- Optimization
- *Partial Evaluation*
- Static & Dynamic parameters

Parameterization is Lambda Abstraction

"Phrases from any semantically meaningful syntactic class may be parameters."

- Parameterization is distinct from Abstraction
- They are "orthogonal" to each other

Thank you!

Thank you for your attention!

I would be pleased to answer any upcoming questions!

Thomas@Klambauer.info

http://klambauer.info

References I

David A. Schmidt.

The Structure of Typed Programming Languages (Foundations of Computing).
The MIT Press, 1994.