0.1 Type abstraction

In this section we will study type calculi that allow us to express the various forms of polymorphism, including *inclusion polymorphism* (due to inheritance), *parametric polymorphism* (due to generics) and *intersection types* (due to overloading), in a syntactic way, by means of appropriate *type expressions*.

The type calculi are based on the typed lambda calculus originally introduced in [Ca84] to study the semantics of multiple inheritance. We will first study some simple extensions to the typed lambda calculus and then discuss examples involving universal quantification (defining parametric types), existential quantification (hiding implementation details) and bounded quantification (modeling subtypes derived by inheritance). For those not familiar with the lambda calculus, a very elementary introduction is given below. For each calculus, examples will be given to relate the insights developed to properties of the C++ type system.

The lambda calculus The lambda calculus provides a very concise, yet powerful formalism to reason about computational abstraction. The introduction given here has been taken from [Barend], which is a standard reference on this subject.

0-1

Lambda calculus – ***very informal*** $\boxed{\lambda}$

- variables, abstractor λ, punctuation (,)

Lambda terms – Λ

- $x \in \Lambda$ variables
- $M \in \Lambda \Rightarrow \lambda x.M \in \Lambda$ abstraction
- $M \in \Lambda$ and $N \in \Lambda \Rightarrow MN \in \Lambda$ application

Slide 0-1: The lambda calculus – terms

Syntactically, lambda terms are built from a very simple syntax, figuring variables, the abstractor λ (that is used to bind variables in an expression), and punctuation symbols. Abstractors may be used to abstract a lambda term M into a function $\lambda x.M$ with parameter x. The expression $\lambda x.M$ must be read as denoting the function with body M and formal parameter x. The variable x is called the bound variable, since it is bound by the abstractor λ. In addition to function abstraction, we also have (function) application, which is written as the juxtaposition of two lambda terms. See slide 0-1.

Behaviorally, lambda terms have a number of properties, as expressed in the laws given in slide 0-2.

The most important rule is the *beta conversion* rule, which describes in a manner of speaking how parameter passing is handled. In other words function call, that is the application $(\lambda x.M)N$, results in the function body M in which N is substituted for x. Two other laws are the so-called extensionality axioms, which express how equality of lambda terms is propagated into application and function

Laws

- $(\lambda\, x.M)N = M[x := N]$ conversion
- $M = N \Rightarrow MZ = NZ$ and $ZM = ZN$
- $M = N \Rightarrow \lambda\, x.M = \lambda\, x.N$

0-2

Slide 0-2: The lambda calculus – laws

abstraction. These laws impose constraints upon the models characterizing the meaning of lambda terms.

Substitution

- $x[x := N]{\equiv}N$
- $y[x := N]{\equiv}y$ if $x{\neq}y$
- $(\lambda\, y.M)[x := N]{\equiv}\lambda\, y.(M[x := N])$
- $(M_1 M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N])$

0-3

Slide 0-3: The lambda calculus – substitution

Substitution is defined by induction on the structure of lambda terms. A variable y is replaced by N (for a substitution $[x := N]$) if y is x and remains y otherwise. A substitution $[x := N]$ performed on an abstraction $\lambda\, y.M$ results in substituting N for x in M if x is not y. If x is identical to y, then y must first be replaced by a fresh variable (not occurring in M). A substitution performed on an application simply results in applying the substitution to both components of the application. See slide 0-3.

Some examples of *beta conversion* are given in slide 0-4. In the examples, for simplicity we employ ordinary arithmetical values and operators. This does not perturb the underlying λ-theory, since both values and operations may be expressed as proper λ-terms.

Examples

$$(\lambda x.x)1 = x[x := 1] = 1$$
$$(\lambda x.x + 1)2 = (x + 1)[x := 2] = 2 + 1$$
$$(\lambda x.x + y + 1)3 = (x + y + 1)[x := 3] = 3 + y + 1$$
$$(\lambda y.(\lambda x.x + y + 1)3)4) =$$
$$((\lambda x.x + y + 1)3)[y := 4] = 3 + 4 + 1$$

0-4

Slide 0-4: Beta conversion – examples