Scribe: Vivek Vishnumurthy and Ashwin Kumar Lecturer: Radu Rugina

1 Parametric Polymorphism (Universal Types)

In classical λ- calculus, we have come across combinators like $true = \lambda x.\, \lambda y.\, x$. In Typed λ-calculus, however, we seem to have a problem – we need to define a different combinator for every primitive type.

$$\begin{aligned} true_{int} &= \lambda x : int.\, \lambda y : int.\, x : int \to int \to int \\ true_{bool} &= \lambda x : bool.\, \lambda y : bool.\, x : bool \to bool \to bool \end{aligned}$$

Is it possible to *parametrize* the type of the variables in the above definitions and get a common definitions for all the types ?

$$true_T = \Lambda T.\, \lambda x : T.\, \lambda y : T.\, x : T \to T \to T$$

1.1 Polymorphic λ-calculus

$$\begin{aligned} e &::= p \,|x\,|e_0 e_1\,|\lambda x : \tau.\, e\,|\Lambda T.\, e\,|e[\tau] \\ \tau &::= int\,|bool\,|\tau_1 \to \tau_2\,|\forall T.\tau\,|T \end{aligned}$$

Examples in C++

- $\Lambda T.\, e$: template ⟨class T⟩ class list. . .;
- $e[\tau]$: list⟨int⟩ * l_1;

The typing construct Γ is a partial function defined as $\Gamma : Var \to Types$. Also define Δ as a set of typed variables. We define Δ so that we do not use universal variables out of scope; i.e., we should not be able to write $\forall T.int \to S$. $\Delta \vdash \tau$ means τ is well formed.

$$\frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2} \qquad \frac{\Delta \cup \{T\} \vdash \tau}{\Delta \vdash \lambda T.\, \tau} \qquad \frac{T \in \Delta}{\Delta \vdash T} \qquad \frac{}{\Delta \vdash int} \qquad \frac{}{\Delta \vdash bool}$$

The typing rules $\Gamma, \Delta \to e : \tau$ mean that with respect to Γ and Δ, e is well typed with type τ.

$$\frac{\Gamma, \Delta \cup \{T\} \vdash e : \tau \quad T \notin \Delta}{\Gamma, \Delta \vdash \Lambda T.\, e : \forall T.z} \qquad \frac{\Gamma, \Delta \vdash e : \forall T.\tau' \quad \Delta \vdash \tau}{\Gamma, \Delta \vdash e[\tau] : \tau'[T \to \tau]}$$

$$\frac{\Gamma, \Delta \vdash e_0 : \tau' \to \tau \quad \Gamma, \Delta \vdash e_1 : \tau'}{\Gamma, \Delta \vdash e_0 e_1 : \tau} \qquad \frac{\Gamma[x \to \tau], \Delta \vdash e : \tau' \quad \Delta \vdash \tau}{\Gamma, \Delta \vdash \lambda x : \tau.\, e : \tau \to \tau'}$$

Though polymorphism does give us more power to express universal types, it is not sufficient to correctly type combinators like $self = \lambda x.\, xx$. Only recursive types can correctly type such combinators.

2 Program Analysis

Methods for verification of program correctness

- **Types:** Types give a light-weight mechanism to realize a weak notion of program correctness. Type-checking ensures that the program does not end in a stuck configuration.
- **Axiomatic Semantics:** This is a heavy-weight mechanism based on Theorem proving. This method is not guaranteed to terminate.

- **Program Analysis:** This technique falls between the above two in terms of complexity, and is guaranteed to terminate. Program Analysis has:
  - An Abstract Domain
  - Abstract Semantics and
  - Abstract Interpretation (Similar to Data Flow Analysis)

Eg: Consider *x:int*. The fact that x is an integer is concrete information, while whether x is positive always is an abstraction.

Differences between Data Flow Analysis and Program Analysis will be explained in the next lecture.

2.1 Lattice

$(L, \sqsubseteq)$ is a complete lattice if

- $\sqsubseteq$ is a partial order
- Any subset $X \subseteq L$ has an lub $\bigsqcup X$, and a glb $\bigsqcap X$

2.1.1 Abstract Operators

We define the following abstract properties using lattices:

- Join Operator: $\bigsqcup$
- Meet Operator: $\bigsqcap$
- Top $= \bigsqcup L$
- Bottom $= \bigsqcap L$

An alternative notation for a complete lattice is: $(L, \sqsubseteq, \bigsqcap, \bigsqcup, \top, \bot)$

2.1.2 Relation between $\sqsubseteq$ and $\sqcap, \bigsqcup$

$$x \sqsubseteq y \text{ iff } x \bigsqcap y = x$$
$$x \sqsubseteq y \text{ iff } x \bigsqcup y = y$$

2.1.3 Properties of $\sqcap, \bigsqcup$

- $\bigsqcap$ is idempotent. i.e.,$x \bigsqcap x = x$ ($\sqsubseteq$ is reflexive)
- $\bigsqcap$ is commutative ($\sqsubseteq$ is anti-symmetric)
- $\bigsqcap$ is associative: $x \bigsqcap (y \bigsqcap z) = (x \bigsqcap y) \bigsqcap z$ ($\sqsubseteq$ is transitive)

$\bigsqcup$ has the same properties as $\bigsqcap$ above.

2.1.4 Chains

A Chain is a totally ordered subset of L.

- An *Ascending Chain* is $\{x_n\}_{n \in \mathbb{N}}$ such that: $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$
- A *Descending Chain* is $\{x_n\}_{n \in \mathbb{N}}$ such that: $x_0 \sqsupseteq x_1 \sqsupseteq \ldots$

2.1.5 Conditions corresponding to program termination

- *Ascending Chain Condition:* Any ascending chain $\{x_n\}_{n\in\mathbb{N}}$ stabilizes: $\exists n_0.\forall n \geq n_0.x_n = x_{n_0}$ (Program Analysis terminates after n_0 steps).
- *Descending Chain Condition:* Any descending chain stabilizes.