The Abstraction Principle

Programming Languages CS442

David Toman

School of Computer Science
University of Waterloo

The Abstraction Principle

Semantically meaningful phrases can be named.

- location $\Rightarrow$ alias
- numeral $\Rightarrow$ const
- expression $\Rightarrow$ function
- command $\Rightarrow$ procedure
- declaration $\Rightarrow$ module
- type structure $\Rightarrow$ class

. . . where/when do *variables* come in??

Declarations

Declaration: **define** $I = V$ (or just $I = V$)

Invocation: **invoke** I (or just I)

- how does this fit with *typing rules*?
- what does it *mean*?
- design choices (do declarations *execute*)?

Syntax: adding functions

$$
\begin{array}{lcl}
P & ::= & D \,\textbf{in}\, C \\
D & ::= & \textbf{fun}\, I = E \mid D, D \mid D; D \\
C & ::= & L := E \mid C; C \mid \textbf{if}\, E \,\textbf{then}\, C \,\textbf{else}\, C \,\textbf{fi} \mid \textbf{while}\, E \,\textbf{do}\, C \,\textbf{od} \mid \textbf{skip} \\
E & ::= & N \mid @L \mid E + E \mid \neg E \mid E = E \mid I \\
L & ::= & loc_i \quad (i > 0) \qquad\qquad N ::= n \quad (n \textit{ an integer numeral})
\end{array}
$$

Type Assignments

- How do we determine *type annotation* for *identifiers*?

Idea

Type Assignment = set of identifier-type attribute pairs

- *declarations add new pairs*
- *invocations look up type attributes for identifiers*

- Typing assertions extended to handle type assignments:

$$\pi \vdash P : \theta$$

π type assignment, P fragment of syntax, θ type attribute

$\Rightarrow \pi$ describes “free identifiers” in P!

Typing Rules for Functions

Declaration:

$$\frac{\pi \vdash E : \tau exp}{\pi \vdash \textbf{fun}\, F = E : \{F : \tau exp\} dec}$$

Invocation:

$$\pi \vdash F : \tau exp \;\; if \;\; (F : \tau exp) \in \pi$$

Glue:

$$\frac{\emptyset \vdash D : \pi dec \qquad \pi \vdash C : comm}{\emptyset \vdash D \,\textbf{in}\, C : comm}$$

$$\frac{\pi \vdash D_1 : \pi_1 dec \qquad \pi \vdash D_2 : \pi_2 dec}{\pi \vdash D_1, D_2 : \pi_1 \uplus \pi_2 dec}$$

$$\frac{\pi \vdash D_1 : \pi_1 dec \qquad \pi \uplus \pi_1 \vdash D_2 : \pi_2 dec}{\pi \vdash D_1; D_2 : \pi_1 \uplus \pi_2 dec}$$

Semantics

- How do we determine *meanings* for *identifiers*?

Idea

Environment = set of identifier-meaning pairs

- *declarations add new pairs*
- *invocations look up meanings for identifiers*

- Environments are *additional inputs* to semantics functions

$$[\![\pi \vdash P : \theta]\!] \in [\![\pi]\!] \to \mathit{Store}_{\perp} \to [\![\theta]\!]$$

Semantics of Function Declarations

Declaration: $[\![\pi \vdash \textbf{fun}\, F = E : \{F : \tau exp\} dec]\!] e\ s = \{F = f\}$
$$where\ \ f = [\![\pi \vdash E : \tau exp]\!] e$$

Invocation: $[\![\pi \vdash F : \tau exp]\!] e\ s = (f\ s)\ \ where\ \ (F = f) \in e$

Glue: $[\![\emptyset \vdash D\, \textbf{in}\, C : comm]\!] s =$
$$[\![\pi \vdash C : comm]\!]([\![\emptyset \vdash D : \pi dec]\!] \emptyset\ s)\ s$$
$[\![\pi \vdash D_1, D_2 : (\pi_1 \uplus \pi_2) dec]\!] e\ s =$
$$[\![\pi \vdash D_1 : \pi_1 dec]\!] e\ s \cup [\![\pi \vdash D_2 : \pi_2 dec]\!] e\ s$$

Semantics of Function Declarations (variant)

Declaration: $[\![\pi \vdash \textbf{fun}\, F = E : \{F : \tau exp\} dec]\!] e\ s = \{F = v\}$

$$where\ \ v = [\![\pi \vdash E : \tau exp]\!] e\ s$$

Invocation: $[\![\pi \vdash F : \tau exp]\!] e\ s = v \ \ where\ \ (F = v) \in e$

Glue: $[\![\emptyset \vdash D\ \textbf{in}\ C : comm]\!] s =$

$$[\![\pi \vdash C : comm]\!]([\![\emptyset \vdash D : \pi dec]\!] \emptyset\ s)\ s$$

$[\![\pi \vdash D_1; D_2 : (\pi_1 \uplus \pi_2) dec]\!] e\ s =$

$$[\![\pi \vdash D_1 : \pi_1 dec]\!] e\ s \cup [\![\pi \uplus \pi_1 \vdash D_2 : \pi_2 dec]\!] e\ s$$

$\Rightarrow$ any other options?

Soundness of Typing

- type attributes (so far):

$$\theta ::= \tau exp \mid intloc \mid comm \mid \pi dec$$
$$\tau ::= int \mid bool$$
$$\pi ::= \{I : \theta_I\}_{I \in \mathcal{I}}$$

- meanings:

$$[\![\tau exp]\!] = Store \to [\![\tau]\!] \qquad [\![comm]\!] = Store \to Store_\perp$$
$$[\![\tau dec]\!] = Store \to [\![\pi]\!] \qquad [\![\{I : \theta_I\}_{I \in \mathcal{I}}]\!] = \{I : [\![\theta_I]\!]\}_{I \in \mathcal{I}}$$

- $e \in Env_\pi$ an environment consistent with π

$(I : \theta) \in \pi$ if and only if $(I : v) \in e$ and $v \in [\![\theta]\!]$.

Theorem (Soundness od Typing)

$$\pi \vdash U : \theta, e \in Env_\pi \Rightarrow [\![\pi \vdash U : \theta]\!] e \in [\![\theta]\!]$$

The COPY Rule

Idea

$$[\![\textbf{\textit{fun}}\, F = E\, \textbf{\textit{in}}\, P]\!] = [\![P[E/F]]\!]$$

- does it *preserve* type attributes?
- is this rule *sound*?

  $\Rightarrow$ depends on the semantic used!

Other Standard Abstractions: Syntax

$$
\begin{array}{lcl}
P & ::= & D\ \mathbf{in}\ C \\
D & ::= & \mathbf{fun}\ I = E \mid \mathbf{proc}\ I = C \mid \mathbf{const}\ I = N \mid \mathbf{alias}\ I = L \mid D, D \mid D; D \\
C & ::= & L := E \mid C; C \mid \mathbf{if}\ E\ \mathbf{then}\ C\ \mathbf{else}\ C\ \mathbf{fi} \mid \mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{od} \mid \mathbf{skip} \mid I \\
E & ::= & N \mid @L \mid E + E \mid \neg E \mid E = E \mid I \\
L & ::= & loc_i \ \ (i > 0) \mid I \qquad\qquad N ::= n \ \ (n \text{ an integer numeral}) \mid I
\end{array}
$$

Other Standard Abstractions: Typing and Semantics

- Typing rules:

  Declarations:

$$\frac{\pi \vdash U : \theta}{\textbf{define}\, I = U : \{I : \theta\} dec}$$

  Invocations:

$$\textbf{invoke}\, I : \theta \quad \textit{if} \quad (I : \theta) \in \pi$$

- Semantics:

  $\Rightarrow$ same considerations as in the **fun** case

Recursive Abstractions

- all abstractions (so far) are *stratified*

  $\Rightarrow$ definition(s) must *strictly* precede invocation(s)

- what would happen if we *relaxed* this requirement?
  - Typing rule:

$$\frac{\pi \uplus \{I : comm\} \vdash C : comm}{\textbf{rec-proc}\, I = C : \{I : comm\} dec}$$

  - Semantics:

$$[\![\pi \uplus \{I : comm\} \vdash C : comm]\!] e\ s = \{I = p\}$$
$$\text{where}\ \ p = [\![\pi \uplus \{I : comm\} \vdash C : comm]\!](e \cup \{I = p\})$$

- Copy rule?

Summary

- abstraction gives many standard syntactic constructs

  $\Rightarrow$ in a *uniform* way!

- semantic definitions *fix* how to process declarations

  $\Rightarrow$ even here eager/lazy makes difference!

- Questions:
  1. typing rules for *dynamic* variant of functions?
  2. what do *eager* commands do?
  3. is there really any use for *eager* declarations at all?