Data Abstraction

Recall the central ideas of data abstraction:

- Define a **representation type** together with **operations** that manipulate values of that type.
- Hold the representation type **abstract** from clients of the ADT to ensure representation independence.

Existential Types vs. Dot Notation

Existential types provide a **closed-scope** abstraction mechanism.

- Representation type held abstract for use within a particular expression
- All clients of the abstraction must lie in the same scope
- Not a good fit with **let** binding

Dot Notation vs. Existential Types

Dot notation provides an **open-scope** abstraction mechanism.

- Direct access to representation type and operations of a package permitted
- Abstraction separated from binding
- Abstract types are intrinsically abstract, not just held abstract in a specified scope

Dot Notation: Decomposing Existential Types

- Existential types replaced by **signatures**, which describe the association of a type with operations on it.
- Packages are separated into two constructs:
  - a **structure** consisting of a representation type together with its associated operations
  - **sealing** a structure with a signature

Dot Notation: Decomposing Existential Types (cont.)

- The **open** construct is broken into two constructs:
  - **binding** a structure to an identifier for use within a scope
  - **projecting** the type and value components from a structure
Dot Notation Types and Expressions

We extend the syntax of PolyMinML with the following constructs:

**Types**

\[ \tau ::= \text{sig}(t, \tau) \mid \text{rpn}(e) \]

**Expressions**

\[ e ::= \text{str}(\tau, e) \mid \text{seal}(e, \text{sig}(t, \tau)) \mid \text{ops}(e) \mid \text{let}(e_1, x, e_2) \]

Values

We must know when an open expression \( e \) is a value:

\[ x \text{ value} \quad \text{lambda}(\tau, x, e) \text{ value} \]

\[ \text{Lambda}(t, e) \text{ value} \quad \text{str}(\tau, e) \text{ value} \]

Paths

Restricting the expression \( e \) in \( \text{rpn}(e) \) to be a path as defined below, suffices to insure that \( e \) is a path.

This in turn insures that type equality will never involve expression equality except for variables, so type equality remains syntactic identity.

Type Formation

The type formation rules are mutually recursive with the type rules:

\[ \Delta, \rho \vdash \text{type} \quad \Delta, \Gamma \vdash e : [t \leftarrow \rho] \tau \quad \Delta, \Gamma \vdash e : \text{sig}(t, \tau) \]

\[ \Delta, \Gamma \vdash e : \text{rpn}(p) : \text{type} \]

\[ \Delta, \Gamma \vdash e : \text{sig}(t, \tau) \quad \Delta, \Gamma \vdash e : \text{str}(e, \text{sig}(t, \tau)) : \text{sig}(t, \tau) \]

Type Rules

\[ \Delta, \Gamma \vdash e_1, e_2 : \tau \quad \Delta, \Gamma \vdash e_1 : \tau \]

\[ \Delta, \Gamma \vdash \text{let}(e_1, x, e_2) : \tau \]

Determinacy

Valid \( \text{rpn}(e) \) types must have expressions \( e \) that are statically well-determined, or **determinate**.

\[ x \text{ det} \quad e \text{ det} \quad \text{rpn}(e) \text{ det} \]
Substitution

- Dynamic semantics involve substituting structure values for variables.
- Must restrict possible substitutions to avoid creation of illegal type expressions.
  - First define substitution of a determinate expression $e$ for a variable $x$ in a path $p$, written as $\{e \leftarrow x\}p$
  - Second define substitution of a determinate expression $e$ for a variable $x$ in a type $\tau$, written as $\{e \leftarrow x\}\tau$

See section 22.1.3 for details.

Safety

Type formation is preserved by substitution of determinate expressions for free variables.

**Lemma 1 (Substitution in Type Formation)**
If $\Delta, \Gamma, x : \tau \vdash \sigma$ type, $\Delta, \Gamma \vdash e : \tau$ and $e$ det, then $\Delta, \Gamma \vdash \{e \leftarrow x\} \sigma$ type.

**Theorem 2 (Preservation)**
If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

**Theorem 3 (Progress)**
If $e : \tau$ then either $e$ value or there exists $e'$ such that $e \mapsto e'$.

Dynamic Semantics

- $e \mapsto e'$
  - $\text{ops}(e) \mapsto \text{ops}(e')$
  - $\text{ops}(\text{str}(\rho, e)) \mapsto e$

- $e_1 \mapsto e'_1$
  - $\text{let}(e_1, x, e_2) \mapsto \text{let}(e'_1, x, e_2)$
  - $\text{let}(e_1, x, e_2) \mapsto (e_1 \leftarrow x)e_2$

Encoding Existentials

Existential types may be encoded in terms of signatures and structures.

- $\exists(x) : = \text{sig}(t, \sigma)$.
- pack $\tau$ with $v$ as $\exists(x)\text{end} : = \text{seal}(\text{str}(\tau, v), \text{sig}(t, \sigma))$.
- open $e$ as $\text{with } x : \sigma \text{ in } e'\text{end} : = \text{let}(e, y, [t \leftarrow \text{rpn}(y), \text{ops}(y)]e')$, where $y \# e'$. 

12

13

14

15