

Existential Types

Lecture 19

CS 565

4/24/09

Data Abstraction

- Ability to hide (abstract) concrete implementation details
 - Modularity builds on data abstraction
 - Improves program structure and minimizes dependencies
 - One of the most influential developments of the 1970s
 - Key element for much of the success of object orientation in the 1980s
-

Example of Abstraction

- Cartesian points
- Introduce the “abstype” language construct:

abstype point implements

$mk : \text{real} \rightarrow \text{real} \rightarrow \text{point}$

$xc : \text{point} \rightarrow \text{real}$

$yc : \text{point} \rightarrow \text{real}$

is

$\langle \text{point} = \text{real} \times \text{real},$

$mk = \lambda x. \lambda y. (x, y)$

$xc = \text{fst},$

$yc = \text{snd} \rangle$

- Shows a concrete implementation
 - Allows the rest of the program to access the implementation through an abstract interface
 - Only the interface need to be publicized
 - Allows separate compilation
-

Existential Types

- Provides two views:
 - From the outside, the actual representation is hidden or opaque
 - From the inside, the object has the type defined by the representation
 - Example:
 - The package: `abstype p implements`
 $\langle a : p, f : p \rightarrow p \rangle$
is $\langle p = \text{Nat}, a = 0, f = \text{inc} \rangle$
 - and has type $\exists \tau. \langle a : \tau, f : \tau \rightarrow \tau \rangle$
 - Note that a given existential type may be associated with many different implementations
-

Example

abstype p implements

$a : p$

$f : p \rightarrow p$

is $\langle p = \text{Nat}, a = 0, f = \text{inc} \rangle$

has type $\exists \tau. \langle a : \tau, f : \tau \rightarrow \tau \rangle$

as does

abstype p implements

$a : p$

$f : p \rightarrow p$

is $\langle p = \text{Bool}, a = \text{true}, f = \text{not} \rangle$

Data Abstraction

- It is useful to separate the creation of the abstract type and its use
- Extend the syntax:

Terms ::= ... | $\langle t = \tau, e : \sigma \rangle$ | open e_a as $t, x : \sigma$ in e_b

Types ::= ... | $\exists t. \sigma$

- The expression " $\langle t = \tau, e : \sigma \rangle$ " takes the concrete implementation e and "packs it" as a value of an abstract type t with hidden representation type τ and actual type σ
 - The "open" expression allows e_b to access the abstract type expression e_a using the name x , the unknown type of the concrete implementation t
-

Example with Abstraction

- $C = \{mk = \lambda x.\lambda y.(x,y), xc = fst, yc = snd\}$ is a concrete implementation of points as $real \times real$
 - We want to hide the type of the representation
 σ is the following type:
 $\{mk : real \rightarrow real \rightarrow point, xc : point \rightarrow real, yc : point \rightarrow real\}$
 - Note that $C : [real \times real/point]\sigma$
 - $A = \langle point = real \times real, C : \sigma \rangle$ is an expression of the abstract type $\exists point.\sigma$
 - We want clients to access only the second component of A and just use the abstract name "point" for the first component:
 $open\ A\ as\ point,\ P\ in\ \dots\ P.xc(P.mk\ 1.0\ 2.0)\ \dots$
-

Typing Rules for Existential Types

- We add the following typing rules:

$$\frac{\Gamma \vdash [\tau/t]e : [\tau/t]\sigma}{\Gamma \vdash \langle t = \tau, e : \sigma \rangle : \exists t. \sigma}$$

$$\frac{\Gamma \vdash e_a : \exists t. \sigma \quad \Gamma, t, p : \sigma \vdash e_b : \tau}{\Gamma \vdash \text{open } e_a \text{ as } t, p : \sigma \text{ in } e_b : \tau} \quad t \notin FV(\Gamma \cup \tau)$$

- The restriction in the rule for “open” ensures that t does not escape its scope
-

Evaluation Rules for Abstract Types

- We add a new form of value

$$v ::= \dots \mid \langle t = \tau, v : \sigma \rangle$$

- This is just like v but with some type decorations that make it have an existential type

$$\frac{e_a \Downarrow \langle t = \tau, v : \sigma \rangle \quad [v/x][\tau/t]e_b \Downarrow v'}{\text{open } e_a \text{ as } t, x : \sigma \text{ in } e_b \Downarrow v'}$$

- At the time e_b is evaluated, abstract-type variables are replaced with concrete values
 - If we ignore the type issues "open e_a as $t, x : \sigma$ in e_b " is like "let $x : \sigma = e_a$ in e_b "
 - What is different is that e_b cannot know statically what the concrete type of x is so it cannot take advantage of it
-

Abstract Types as a Specification Mechanism

- Just like polymorphism, existential types are mostly a type checking mechanism
 - A function of type $\exists t. t \text{ List} \rightarrow \text{int}$ does not know statically the type of the list elements supplied as arguments. Therefore no operations are allowed on them
 - But the function will know at run-time the actual value of t
 - “There are no type variables at run-time”
 - Same goes for existentials
 - These type mechanisms are very powerful form of static checking
-

Data Abstraction and Static Checking

- Example: file descriptors
 - Solution 1:
 - Represent file descriptors as "int" and export the interface {open:string → int, read:int → data}
 - An untrusted client of the interface calls "read"
 - How can we know that "read" is invoked with a file descriptor that was obtained from "open"?
 - We must keep track of all integers that represent file descriptors
 - We design the interface such that all such integers are small integers and we can essentially keep a bitmap
 - This becomes expensive with more complex (e.g. pointer-based) representations
-

Data Abstraction and Static Checking

- Solution 2:
 - Use the same representation but export an abstraction of it
$$\exists fd. File = \exists fd. \{open : string \rightarrow fd, read : fd \rightarrow data\}$$
 - A possible value:
$$Fd = \langle fd = int, \{ open = \dots, read = \dots \} : File \rangle : \exists fd. File$$
 - Now the untrusted client e
$$open Fd \text{ as } fd, x : File \text{ in } e$$
 - At run-time “ e ” can see that file descriptors are integers
 - But that will not help. It cannot cast 7 as a file descriptor
 - Static checking with no run-time costs !
-

Data Abstraction and Static Checking

- Solution 2:
 - Use the same representation but export an abstraction of it
$$\exists fd. File = \exists fd. \{open : string \rightarrow fd, read : fd \rightarrow data\}$$
 - A possible value:
$$Fd = \langle fd = int, \{ open = \dots, read = \dots \} : File \rangle : \exists fd. File$$
 - Now the untrusted client e
$$open Fd \text{ as } fd, x : File \text{ in } e$$
 - At run-time “ e ” can see that file descriptors are integers
 - But that will not help. It cannot cast 7 as a file descriptor
 - Static checking with no run-time costs !
 - Catch: you must be able to type check “ e ”
-