The Abstraction Principle

Muhammad Taimoor Khan
Doktoratskolleg Computational Mathematics
Johannes Kepler University, Linz, Austria

Presentation Outline

- The abstraction principle
- Expression abstractions
- Lazy and eager evaluation
- Other standard abstractions
    - Command abstractions
    - Numeral abstractions
    - Location abstractions
    - Variable abstractions
    - Type structure abstractions
    - Declaration abstractions

The Abstraction Principle – An Overview

The phrases of any semantically meaningful syntactic class may be named – The Abstraction Principle

- Decomposition - nontrivial task into sub *named* tasks
- Design construct for more general purpose
- define I = V (I to V binding – name to body*)*
  - Identifier - syntax domain for name
  - Declaration - syntax domain for definition constructs
- Invocation – I, invoke I, call I

Syntax Domains - Expression Abstractions

P ∈ Program
D ∈ Declaration
C ∈ Command
E ∈ Expression

L ∈ Location
N ∈ Numeral
I ∈ Identifier

P ::= D **in** C
D ::= **fun** I=E | D_1, D_2 | D_1;D_2
C ::= L:=E | C_1;C_2 | **if** E **then** C_1 **else** C_2 **fi** | **while** E **do** C **od** | **skip**
E ::= N | @L | E_1+E_2 | E_1=E_2 | ¬E | I
L ::= loc_i , $i > 0$
N ::= n, $n \in N$

Expression Abstractions

- Expression abstractions – functions

fun *A*=1+@loc_1

loc_1:=*A*+2 – well formed command

- Type attribute is established when function is defined
- Type assignment - set of identifier and type attribute pairs
  - Function type is to be communicated to the expressions/commands that use the function

fun *A*=1+@loc_1

fun *B*=@loc_1=0

π = {*A:intexp, B:boolexp*}*dec*

  - Represents only partial functions – at most one pair $(I:\theta)\in\pi$
    - **fun** *A*=1+@loc_1 ,**fun** *A*=@loc_1=0 – not well formed
    - loc_1:=*A*+2 – ($\pi \vdash loc_1$:=*A*+2: comm) - well-typed

Typing Rules - Expression Abstractions

- Annotate syntax tree
  - Annotate each node – type and type assignment attributes
- Function definition

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

- Function invocation

$$\pi \vdash \textbf{I}:\tau exp \text{ if } (I:\tau exp) \in \pi$$

π - global declarations, otherwise an empty set $\emptyset$

Typing Rules - More

- Case: D_1,D_2

$$\frac{\pi \vdash D_1 : \pi_1 dec \qquad \pi \vdash D_2 : \pi_2 dec}{\pi \vdash D_1,D_2 : (\pi_1 \mathbin{\mathring{\cup}} \pi_2) dec}$$

where $(\pi_1 \mathbin{\mathring{\cup}} \pi_2) = (\pi_1 \cup \pi_2)$ if $\{I \mid (I:\theta_1) \in \pi_1\} \cap \{I \mid (I:\theta_2) \in \pi_2\} = \emptyset$

- Case: $D_1;D_2$

$$\frac{\pi \vdash D_1 : \pi_1 dec \qquad \pi \mathbin{\mathring{\cup}} \pi_1 \vdash D_2 : \pi_2 dec}{\pi \vdash D_1;D_2 : (\pi_1 \mathbin{\mathring{\cup}} \pi_2) dec}$$

Semantics of Abstractions

- Definitions be saved in a structure – environment
  - Denoted by *e*
  - Set of identifier, meaning pairs

    $[[\pi \vdash C: comm]] \in Environment \rightarrow Store \rightarrow Store_{\perp}$

  - Semantics of assignment becomes

    $[[\pi \vdash L:=E: comm]]e\ s = update([[\pi \vdash L: intloc]]e, [[\pi \vdash E: intexp]]e\ s, s)$

- Function definition

  $[[\pi \vdash \textbf{fun}\ I:=E: \{\tau exp\} dec]]e\ s = \{I = f\}$, where $f(s') = [[\pi \vdash E: \tau exp]]e\ s'$

- Function invocation – a crucial clause

  $[[\pi \vdash I: \tau exp]]e\ s = v(s)$, where $(I=v) \in e$

Semantics of Abstractions – Example

fun A:=@loc_1 , loc_2 :=A

where $e_1 = \{A = f\}$, where $f_1(s) = lookup(loc_1, s)$
$f_1(s) = [[\emptyset \vdash @loc_1 : intexp]]\emptyset\ s$

$[[\pi_1 \vdash @loc_2 := A : comm]]e_1\ (2,3,4)$
$= update([[\pi_1 \vdash loc_2 : intloc]]e_1, [[\pi_1 \vdash A : intexp]]e_1\ (2,3,4), (2,3,4))$
$= update(loc_2, f_1(2,3,4), (2,3,4))$
$= update(loc_2, lookup(loc_1, (2,3,4)), (2,3,4))$
$= update(loc_2, 2, (2,3,4)) = (2,2,4)$

A is not an integer but a mapping from current store to an integer

Soundness of Typing Rules for Abstractions

Soundness - if typing rules and the semantics are well-formed with the addition of typing assignments attributes to typing rules and the environment arguments to semantics

Proof

Environment *e is type consistent with type assignment* π

if (I:θ) ∈ π exactly when (I=*v*) ∈ *e* and *v* ∈ [[θ]]

Let Env_π = set of all environments that are consistent with π

To show: if π ├ U:θ and *e*∈Env_π , then [[π ├ U:θ]]*e* ∈ [[θ]]

Case: π ├ L:=E:*comm* and *e* ∈ Env_π

π ├ L:*intloc* and π ├ E:*intexp //both should hold as per typing rule*

By inductive hypothesis

[[π ├ L:*intloc*]]*e* ∈ *Location* and [[π ├ E:*intexp*]]*e* ∈ *Store* → *Int*

Hence

[[π ├ L:=E:*comm*]]*e s* = *update*([[π ├ L:*intloc*]]*e,* [[π ├ E:*intexp*]]*e s, s*) ∈ *Store*

=> [[π ├ L:=E:*comm*]]*e* ∈ *Store* → $Store_\perp$

Lazy Evaluation

- Abstraction computed at abstraction invocation time

Example

fun F = @loc_1+1 **in** loc_1:=0; loc_1:=F; loc_2:=F+2

- Result computed each time F is invoked
- Given store (2,3) alters store is (1,4) by lazy evaluation

- Uses the previously discussed typing rules as it is the norm
- Definition

define I_1=V_1, **define** I_2=V_2, …, **define** I_n=V_n **in** U => [V_1/I_1, V_2/I_2, … ,V_n/I_n]U

Example

fun A=1, **fun** B=@loc_1=0 **in while** B **do** loc_1:=A+2 **od**

=> [1/A, @loc_1=0/B]**while** B **do** loc_1:=A+2 **od**

= **while** @loc_1=0 **do** loc_1:=1+2 **od**

Lazy Evaluation – Semantics

Key for lazy evaluation is to provide a store to the body of the function, when it is invoked

[[π ├ **define** I=U: {I:θ}*dec*]]*e s* = {I=*f*}, where *f s'* = [[π ├ U:θ]]*e s'*
[[π ├ **invoke** I:θ]]*e s = f s,* where (I=*f*) ∈ *e*

- Soundness of copy rule

[[├ **define** I_1=V_1, **define** I_2=V_2, …, **define** I_n=V_n **in** C:*comm*]] = [[Ø ├ [V_j/I_j]C:*comm*]] Ø
Need to show for all *s* ∈ *Store*
[[├ **define** I_1=V_1, **define** I_2=V_2, …, **define** I_n=V_n **in** C:*comm*]]*s* = [[Ø ├ [V_j/I_j]C:*comm*]] Ø *s*
First we show
[[├ **define** I_1=V_1, **define** I_2=V_2, …, **define** I_n=V_n **in** C:*comm*]]*s* = [[π_1 ├ C:*comm*]]e_1 *s*
where π_1 = $\{I_j:\theta_j\}_{j \in I}$ *and* e_1 = $\{I_j = [[Ø ├ V_j: \theta_j]]Ø \}_{j \in I}$
Case L:=E
[[π_1 ├ L:=E:*comm*]]e_1 *s = update*([[π_1 ├ L:*intloc*]]e_1 , [[π_1 ├ E:*intexp*]]e_1 *s, s*)
= update([[Ø ├ [V_j/I_j]L:*intloc*]] Ø , [[Ø ├ [V_j/I_j]E:*intexp*]]Ø *s , s*), 1<=*j*<=*n*
By induction hypothesis for L and E
[[Ø ├ [V_j/I_j]L:=E:*comm*]] Ø *s* //where s is arbitrary so proved

Eager Evaluation

- Abstraction is computed at definition time only

Example

fun F = @loc_1+1 **in** loc_1:=0; loc_1:=F; loc_2:=F+2

- Given store (2,3) alters store is (3,5) by eager evaluation

Semantics

- Store available at definition= Store available at invocation

[[π ├ **fun** I=E: {I:τ*exp*}]]*e s* = { I= [[π ├ E:τ*exp*]]*e s* }
[[π ├ I:τ*exp*]]*e s* = *v,* where (I=*v*) ∈ *e*

Example

[[├ **fun** *A*=1, **fun** *B*=@loc_1=0 **in while** B **do** loc_1:=*A*+2 **od**:*prog*]](0,2,4)
= [[π ├ **while** B **do** loc_1:=*A*+2 **od**:*comm*]]*e* (0,2,4)
where π = {*A*:*intexp*, *B*:*boolexp*} and
e = [[ø ├ **fun** *A*=1, **fun** *B*=@loc_1=0:π*dec*]]Ø(0,2,4)
….
w(0,2,4)

Lazy evaluation vs Eager evaluation

- Lazy evaluation - call by name-style
- Lazy evaluation can be understood as the *copy rule*
  - Copy rule performs simultaneous substitutions

Example loc_1:=0; loc_1:=(@loc_1+1); loc_2:=(@loc_1+1)+2

  - Copy rule cannot work directly for $D_1;D_2$
- Copy rule fails in eager evaluation
- Eager evaluation - call by value-style
- Store is to be supplied immediately to the body of the abstraction
- Improper meaning of the abstraction's body causes improper meaning of the entire program

Example

$$[[\pi \vdash \textbf{loop}:intexp]]e\ s = \bot$$

fun *A* =**loop in skip**

Eager evaluation demands result as $\bot$

Lazy evaluation executes **skip** and terminates

Other Standard Abstractions

- Definition construct must match invocation construct
- Addition
  - Command abstraction - procedures
  - Numeral abstractions - const
  - Location abstractions - alias

P ::= D **in** C
D ::= **fun** I=E | **proc** I=C | **const** I=N | **alias** I=L | D_1, D_2 | D_1;D_2
C ::= L:=E | C_1:C_2 | **if** E **then** C_1 **else** C_2 **fi** | **while** E **do** C **od** | **skip** | **call** I
E ::= N | @L | E_1+E_2 | E_1=E_2 | ¬E | I
L ::= loc_i | I
N ::= *n* | I

Alias is merely a name for a location and is not a variable declaration

Other Standard Abstractions – command abstraction

- General pattern of an abstraction typing rule

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

$$\pi \vdash \textbf{invoke } I{:}\theta, \text{ if } (I{:}\ \theta) \in \pi$$

D ::= **proc** I=C
C ::= **call** I

- Semantics depends on evaluation – function-like
  - Can use lazy evaluation
  - Can use eager evaluation

Variable Abstractions

- Different from alias abstraction
  - alias abstraction is a name for already allocated location
- Variable abstraction
  - allocates a fresh location in store
  - initializes it – may be
  - binds to an identifier
  - part command and part abstraction
- Replaces location numbers with variable declarations

D ::= … | **var I**
C ::= L:=E | C1;C2 | …
E ::= N | @L | …
L ::= **I**

Variable Abstractions – Semantics

- Typing rules

$$\frac{\pi \vdash \mathsf{I}: \mathit{intloc}}{\pi \vdash \textbf{var}\ \mathsf{I}: \{\mathsf{I}:\mathit{intloc}\}\mathit{dec}} \qquad ; \qquad \pi \vdash \mathsf{I}: \mathit{intloc}, \text{ if } (\mathsf{I}: \mathit{intloc}) \in \pi$$

Example – **var** *X*; **var** *Y* **in** *X*:=0; *Y*:=@*X*+1

- Semantics of variable abstractions
  - Declaration can alter the store

$$[[\pi \vdash \textbf{var}\ \mathsf{I}: \{\mathsf{I}:\mathit{intloc}\}\mathit{dec}]]e\ s = (\{\mathsf{I}=l\}, s')$$
$$\text{where } (l, s') = \mathit{allocate}(s)$$
$$\text{and } \mathit{allocate}: \mathit{Store} \rightarrow (\mathit{Location} \times \mathit{Store})$$

  - Part abstraction – binding I to *l*
  - Part command – updates *s* to *s'*
  - Semantics depends on the evaluation

Type Structure Abstractions

- It is illuminating to separate the part command and part abstraction in the syntax

D ∈ Declaration
T ∈ Type-structure
  D ::= …. | **var** I:T | **class** I=T
  T ::= **newint** | I

- Type structure represents command-like construct
- Type structure is a storage allocation routine
- Semantics
  - When evaluated with current environment and store, alters the store and returns value bounded identifier

[[π ├ **T**=U: δ*class*]] ∈ Env_π → *Store* → ([[δ]]X*Store*) *s* = ({I=*f*}, *s*), where *f s'* = [[π ├ U:θ]]*e s'*

Type Structure Abstractions – newint

- **newint** allocates storage and returns the name
- Syntax

$$T ::= \textbf{newint}$$

- Typing rule

$$\pi \vdash \textbf{newint}:\textit{intloc class}$$

- Semantics

$$[[\pi \vdash \textbf{newint}:\textit{intloc class}]]e\ s = \textit{allocate}(s)$$

where allocate: *Store* → (*Location*X*Store*)

Type Structure Abstractions – var I:T

- Syntax

$$D ::= \textbf{var}\ I{:}T$$
$$T ::= \textbf{newint} \mid I$$

- Typing rules

$$\frac{\pi \vdash T{:}\delta \mathit{class}}{\pi \vdash \textbf{var}\ I{:}T{:}\ \{I{:}\delta\}\mathit{dec}}$$

$$\pi \vdash I{:}\theta, \text{ if } (I{:}\theta) \in \pi$$

- Semantics

$[[\pi \vdash \textbf{var}\ I{:}T{:}\ \{I{:}\delta\}\mathit{dec}]]e\ s = (\{I{=}v\}, s')$, where $(v, s') = \{\pi \vdash T{:}\delta \mathit{class}\}e\ s$

Type Structure Abstractions – class I=T

- Syntax

D::=**class** I=T

T::=**newint** | I

- Typing rules

$$\frac{\pi \vdash T: \delta class}{\pi \vdash \textbf{class}\ I=T: \{I:\delta class\}dec}$$

$$\pi \vdash I:\theta, \text{ if } (I:\theta) \in \pi$$

- Semantics

[[$\pi \vdash$ **class** I=T: {I:δ*class*}*dec*]]*e s* = ({I=*p*}, *s*), where *p s'*={$\pi \vdash$ T:δ*class*}*e s'*

  - Class – lazily evaluated type structure
  - Variable declaration – eagerly evaluated type structure

- Class is more interesting in a record structure

Type Structure Abstractions – Example

var *A*:**newint;**
class *R* = **record var** *C*:**newint**; **proc** *P*=(*C*:=@*A*+1) **end;**
var *F:R*; **var** *G:R*;
in … A … F.C …. **call** F.P …. G.C. …. **call** G.P …;

(var A:newint$^{\text{intloc class}}_{\emptyset}$)$^{\pi 1 dec}$;

Declaration Abstractions

- Declarations can themselves be abstracted – known as *module*

D ::= … | **module** I={D} | **import** I

- Typing rules

$$\frac{\pi \vdash \mathsf{D}:\pi_1 dec}{\textbf{module}\ \mathsf{I}=\{\mathsf{D}\}:\ \{\mathsf{I}:\ \pi_1 dec\}dec}$$

$$\pi \vdash \textbf{import}\ \mathsf{I}=\ \pi_1 dec, \text{ if } (\mathsf{I}=\ \pi_1 dec) \in \pi$$

- Semantics
  - Storage cells are not allocated until the module is imported – lazy evaluation
  - Eager evaluation of module is essential when indexing of components is used

$$[[\pi \vdash \textbf{module}\ \mathsf{I}=\{\mathsf{D}\}:\ \{\mathsf{I}:\pi_1 dec\}dec]]e\ s = (\{\mathsf{I}=f\}, s)$$
where $f(s') = [[\pi \vdash \mathsf{D}:\pi_1 dec]]e\ s'$
$$[[\pi \vdash \textbf{import}\ \mathsf{I}:\pi_1 dec]]e\ s = f(s), \text{ where } (\mathsf{I}=f) \in e$$

Example

module *M* = {**class** *K* = **newint**; **var** *A*:*K*; **proc** *P*=*A*:=0};
var *B*:**newint**;
in call *M.P*; *B*:=@*M.A*

Abstractions as Record Introduction Principle

Abstraction Principle says "semantically meaningful syntactic class" may be named

- Type attributes as a binding between name to a value
  - It is proved by the denotational semantics
- Consider a compound declaration as a set of unique identifiers

$$\{I_1=v_1, I_2=v_2, I_3=v_3, \ldots, I_n=v_n\}$$

  - The set is an environment but is also a record
  - Fundamental operation on records is indexing
    - X.I – extracts the value that is bound to identifier I within X

Phrases of any semantically meaningful syntactic class may be components of records

Thank you for patience!
Questions?