Lecture26-27
From Data Abstractions to Abstract Data Types

prof. Marco Bellia, Dip. Informatica, Università di Pisa

April 9, 2014

From Data Abstractions to Abstract Data Types

- Records: Definition, Allocation e Access
- Introduction and Dynamic Allocation of Records
- ADT: Orthogonality, Locality, Data Integrity
- Abstract Data Types: API and ADT
- ADT: Syntax, Semantics, Functions Up and Down
- ADT: Use and Applications in Ocaml

Record: Definition, Allocation e Access

Remind the general structure of a language with Types and Concrete Types (and apply it to the case of the type record)

Table15 – Types and Concrete Data

Syntactic Domains

$D ::= \ldots \mid \texttt{T I}$ (*naming : Static Allocation*)
$\mid \texttt{type I} = \texttt{T}$ (*Type definition and Concrete Types*)
$\mid \ldots$

$E ::= \ldots \mid \texttt{Talloc (T)}$ (*Dynamic Allocation*)
$\mid \texttt{Den(E.I}_k\texttt{)} \mid \texttt{E.I}_k \mid \texttt{Val(I)}$ (*Access*)
$\mid \ldots$

Auxiliary Syntactic Domains

$T ::= T_A \mid \ldots \mid T_F \mid I \mid \ldots$ (*included Polymorphics*)
$T_A ::= \texttt{int} \mid \ldots \mid \texttt{bool}$ (*Atomics*)
$T_D ::= VL_{inf} .. VL_{sup} \mid \ldots \mid *T$ (*Derived*)
$T_S ::= \texttt{Rec } I_1\texttt{:}T_1 \ldots I_n\texttt{:}T_n \texttt{ End} \mid \ldots$ (*Structured*)
$T_F ::= T_1 \times \ldots \times T_n \rightarrow T \mid \ldots$ (*Functions*)

Declaration and (Static) Allocation of Values: Record/1

- The type Record is the second Type that requires a notion of environment to express its data.
- The other Type, we encountered, is the type Function
- Semantics of the Declaration of a binding I: The identifier is bound to the denotation of a mutable value Record that is statically allocated (at the loading time of the program, for instance)

Table15.1 — Declaration of a Variable Record

Semantic Functions

$\mathcal{D}[\![\mathtt{D}]\!]_\rho : \mathtt{Store} \to (\mathtt{Env} \times \mathtt{Store})_\perp$

$\mathcal{D}[\![\mathtt{Rec\ I_1{:}T_1 \ldots I_n{:}T_n\ End\ I}]\!]_\rho(\mathtt{s}) =$

$\mathtt{Let}\{\mathtt{s_0} = \mathtt{s}, \forall \mathtt{T_i}.(\mathtt{l_i}, \mathtt{s_i}) = \mathit{allocate}(\mathtt{T_i}, \mathtt{s_{i-1}})\}$

$\{\forall \mathtt{I_i}.\gamma(\mathtt{I_i}) = \mathtt{l_i}\}$

$(\mathtt{bind}(\mathtt{I}, \mathtt{E_D}(\gamma), \rho), \mathtt{s_n})$

$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Store} \to (\mathtt{Env} \times \mathtt{Store})_\perp$

$\mathcal{E}[\![\mathtt{Den(E.I_k)}]\!]_\rho(\mathtt{s}) = \mathtt{Let}\{\mathtt{E_D}(\gamma) = \mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s})\}(\gamma(\mathtt{I_k}), \mathtt{s})$

$\mathcal{E}[\![\mathtt{E.I_k}]\!]_\rho = \ldots$

Auxiliary Functions and Domains

$\mathtt{E_D} : \mathtt{EnvL} \to \mathtt{Den}$

$\mathtt{Den} ::= \ldots + \mathtt{EnvL}$

$\mathtt{EnvL} ::= \mathtt{I} \to \mathtt{Den}$

Legenda

$\{\forall \mathtt{I_i}.\gamma(\mathtt{I_i}) = \mathtt{l_i}\}$ is a shortening for :

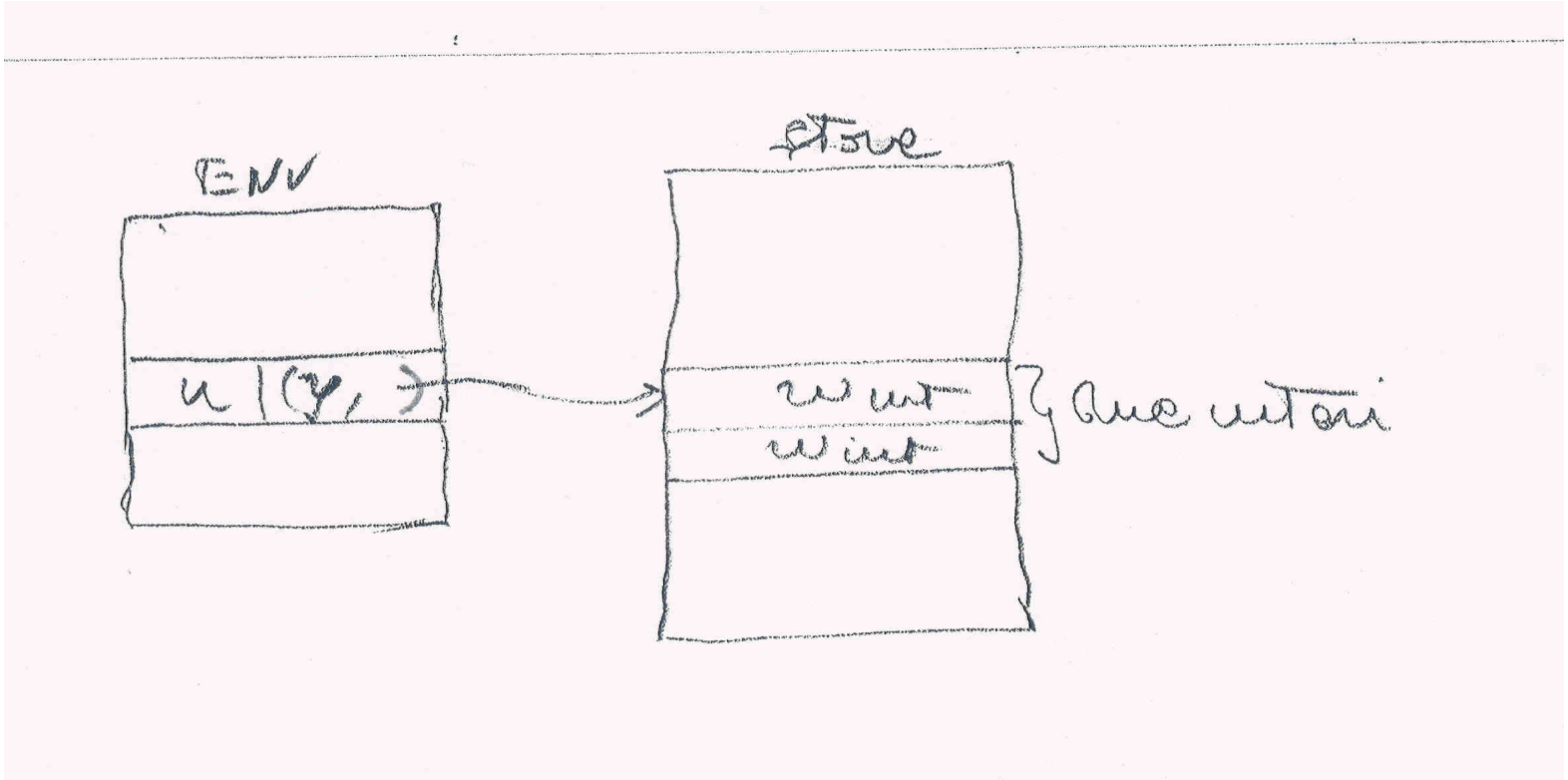
$\mathtt{let\ bind2} = \lambda(\mathtt{I}, \mathtt{D}).\lambda \mathtt{E.bind}(\mathtt{I}, \mathtt{D}, \mathtt{E})$

$\mathtt{in}\ \gamma = (\mathtt{bind2}(\mathtt{I_1}, \mathtt{l_1}) \circ \ldots \circ \mathtt{bind2}(\mathtt{I_n}, \mathtt{l_n}))\ \mathtt{empty}$

Declaration and (Static) Allocation of Values: Record/2

Implementation:

- $\mathtt{E_D}(\gamma)$ is implemented through a pair that contains the environment γ (i.e. a selection function) and the first location of a block of store into which the value of the record has been allocated.
- A graphic picture of: `Rec int x, int y End u`.



Introduction and Dynamic Allocation of a Record

- Creation of a mutable value of type Record (when the Language allows it)
- Reference Type: The access to Records is through pointers

Table15.1.1 – Record Pointers or by Reference Types

Semantic Functions

$$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Store} \to (\mathtt{Val} \times \mathtt{Store})_\perp$$

$$\mathcal{E}[\![\mathtt{Talloc(Rec\ I_1 : T_1 ... I_n : T_n\ End)}]\!]_\rho(\mathtt{s}) =$$
$$\mathtt{Let}\{\mathtt{s_0} = \mathtt{s}, \forall \mathtt{T_i}.(\mathtt{l_i}, \mathtt{s_i}) = \mathit{allocate}(\mathtt{T_i}, \mathtt{s_{i-1}})\}$$
$$\{\forall \mathtt{I_i}.\gamma(\mathtt{I_i}) = \mathtt{l_i}\}$$
$$(\mathtt{E_V}(\gamma), \mathtt{s_n})$$

Auxiliary Functions and Domains

$\mathtt{E_V} : \mathtt{EnvL} \to \mathtt{Val}$ $\qquad$ $\mathtt{E_M} : \mathtt{EnvL} \to \mathtt{Mem}$

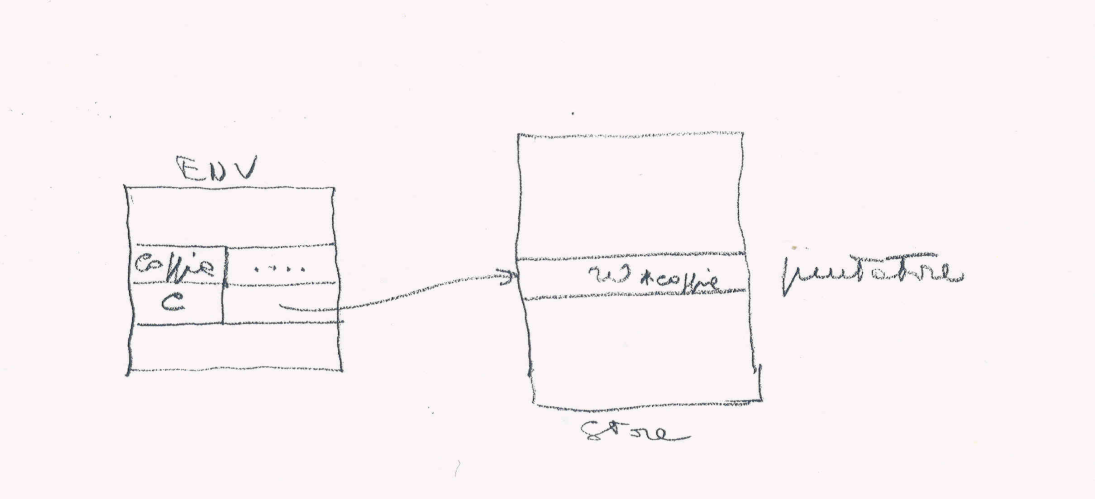
$\mathtt{Val} ::= \ldots + \mathtt{EnvL}$ $\qquad$ $\mathtt{Mem} ::= \ldots + \mathtt{EnvL}$

$\mathtt{EnvL} ::= \mathtt{I} \to \mathtt{Den}$

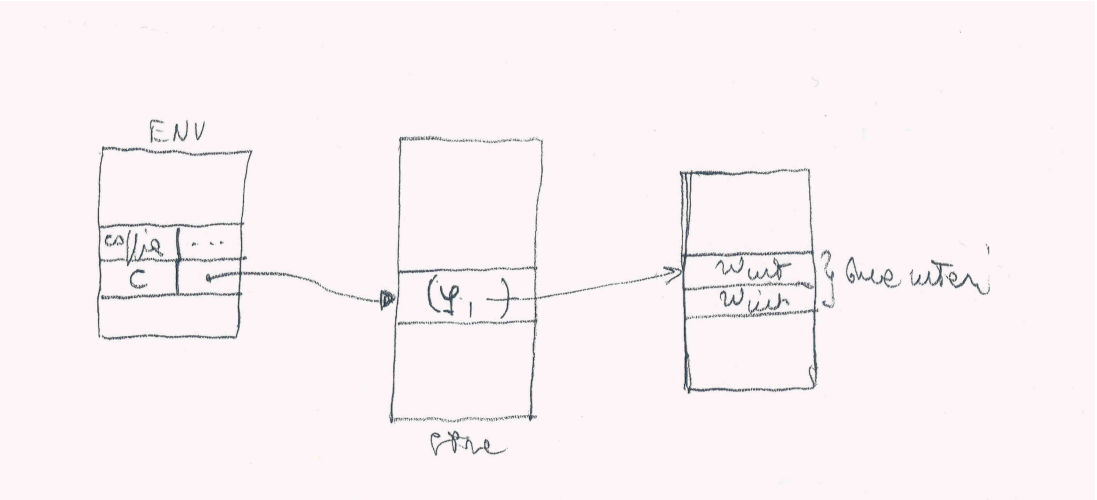
- **Implementation**: Allocation is in the component Heap of `Store`: *allocate* puts on Heap both the components and the access structure (i.e. selection function) that implemented γ, in the static allocation.

Introduction and Dynamic Allocation of a Record/2

- $\mathtt{E_D}(\gamma)$ is implemented through a pair containing γ and the first location of a block of store into which the value of the record has been allocated.
- A graphic picture of type `Coppia` = `Rec int x, int y End`; `Coppia` $*$ `c`;.



- and again: `c` = `Talloc(Coppia)`;



Abstract Data Types: API-ADT

- Orthogonality between use and implementation: User does not need to know its implementation in order to use it (different implementations of a same API may co-exist in a Ocaml program)
- Program Locality in collecting in the API everything is needed to use it whilst everything that is needed to run it is collected in the ADT.
- Protection of the data integrity against accesses and modifications that are due to an improper use of the data
- API (specification part) and ADT (implementation part) together in a language construct.

Table15.2 – Types and ADT(including API)

Syntactic Domain

D ::= ... | T I (*naming : Static Allocation*)
| type I = T (*Type Definition*)
| ADT N : $\overline{Z;}$ {$\overline{U;}$ $\overline{V;}$ $\overline{O;}$}
| ...

Auxiliary Syntactic Domains

T ::= ... (*Types, included Polymorphics*)
N ::= I | ... (*Type Names, included Polymorphic Names*)
Z ::= T I (*Type and Name of Visible Operators*)
U ::= type I = T (*Types, included Auxiliaries*)
V ::= I = E (*Auxiliary Variables*)
O ::= I ($\overline{P}$) C (*Operations*)

Notational Remark

For each domain A, $\overline{A}$ *stands for a sequence* $A_1 \ldots A_n$ *of arbitrary lenght n, of elements of* A.

ADT: Syntax

Table15.2 – Tipi e ADT
Syntactic Domains
D ::= ... | ADT N : $\overline{\texttt{Z;}}$ {$\overline{\texttt{U;}}$ $\overline{\texttt{V;}}$ $\overline{\texttt{O;}}$} | ...

Example

ADT Q:

```
int×int→Q mk;  Q→int fst;  Q→int snd;
{type P = Rec int x,int y End;
 mk(int x, int y){P u = Talloc(P); u.x=x; u.y=y; up(u)};
 fst(Q x){down(x).x};
 snd(Q x){down(x).y};}
```

- What does prevent us from executing `Q w = mk(2,5);` and then to change `w.x` by executing `w.x=10`?
- The answer is: (Semantic) Operations `up`, `down` that *transform* from concrete to abstract values and vice-versa (resp.)
- The concrete syntax does not show the presence of operators `up` and `down` (similarly to what happens with `Den` and `Val` with expressions)
- `up` e `down` may be added, in analogy to `Val` and `Den`, during static analysis.

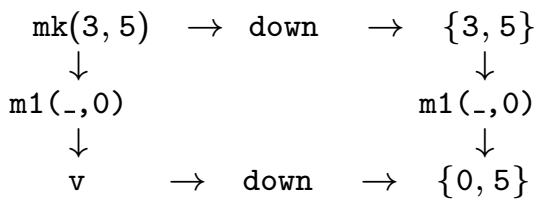
ADT: Semantics/1

- Operations `up`, `down` *transform* from concrete to abstract values and vice-versa
- The definitions of `up`, `down` is basic for the meaning of an abstract data type. Consider the ADT `Q`:
  - The functions that are defined within the ADT, when are dealing with (i.e. for access, or modification) a value `v` of type `Q`, in effect are working with the concrete representation of `v`.
  - Whenever these functions compute a value `v'` of type `Q` that has to be returned outside the ADT, then the returned value is `up(v')`
  - All the code outside the ADT can only deal with the abstract values (i.e. `up(v')`) of the ADT and only through the functions of the ADT that are declared be "public" (i.e. visible) operations
- Semantic operations `up` e `down` correspond to morphisms on algebras (i.e. the one of the signature and the one of an implementation)

Example

ADT Q:

```
int × int → Q mk;  Q → int fst;  Q → int snd;  Q × int → () m1;
{type P = Rec int x, int y End;
 mk(int x, int y){P u = Talloc(P); u.x=x; u.y=y; up(u)};
 fst(Q x){down(x).x};
 snd(Q x){down(x).y};
 m1(Q x, int v){down(x).x=v};
}
```



ADT: Semantics/2

- Semantic operations `up` e `down` correspond to morphisms on algebras (i.e. the one of the signature e and the one of an implementation)
- The Denotational Semantics, we define, is oriented to give insights toward the ADT implementation
- In this semantics, `up` and `down` are two functions that differ one another in the way they access the values.
- Always, the access is based on a notion of **key**
- Each ADT has a specific key: In the denotational semantics, the key is the environment, γ, that the semantics associates to the ADT (similarly to what the semantics associates to records, see previous slides.)
- Hence, a value of the ADT is a pair $(\gamma, \texttt{loc})$, where γ is the semantics of the ADT and `loc` is the location of the store (namely, the Heap) where is the structure, that has been allocated, for the value of the ADT.
  The Concrete Value: $\texttt{c} \equiv (\gamma, \texttt{loc})$,
  The Abstract Value: $\texttt{v} \equiv \texttt{up}(\texttt{c})$
  $$\texttt{up}(\texttt{c}) = \lambda \texttt{key.}\ \texttt{if}(\texttt{key} = \gamma, \texttt{c}, \perp_{\texttt{Mem}})$$
  The Concrete Value, given an abstract value (with key) γ:
  $$\texttt{down}(\texttt{v}) = \texttt{v}(\gamma)$$
- In no way the Abstract Value can access `loc`.
- Operation `up` makes pair $(\gamma, \texttt{loc})$ accessible only by using the expected key
- Operation `down` applies the key: The key is known only to the code within the ADT.
- To increase readability, later on, we will use the following notation: $\rho = \texttt{bind}(\texttt{i}, \texttt{d}, \delta)$ is written as $(\rho(\texttt{i}) = \texttt{d}) \oplus \delta$(possibly, without parentheses)

ADT: Semantics/3

Declaration of a binding for a variable of type ADT and definition of up and down

Table15.3 – ADT

Funzioni Semantiche

$$\mathcal{D}[\![\mathtt{D}]\!]_\rho : \mathtt{Store} \to (\mathtt{Env} \times \mathtt{Store})_\perp$$

$$\mathcal{D}[\![\mathtt{ADT\ N} : \mathtt{T_1\ f_1}; \ldots; \mathtt{T_n\ f_n};\ \{\overline{\mathtt{U};}\ \overline{\mathtt{V};}\ \overline{\mathtt{O};}\}\ \mathtt{I}]\!]_\rho(\mathtt{s}) =$$

$$\mathtt{Let}\{\mathtt{Key} = \lambda(\gamma, \mathtt{loc}).\lambda\mathtt{key}.\mathtt{if}(\mathtt{key} = \gamma, (\gamma, \mathtt{loc}), \perp_{\mathtt{Mem}}))\}$$

$$\{\sigma_1 = \mathtt{Y}\sigma.\mathcal{D}_E[\![\overline{\mathtt{U};}]\!]_\sigma \circ \mathcal{D}_E[\![\overline{\mathtt{V};}]\!]_\sigma \circ \mathcal{D}_E[\![\overline{\mathtt{O};}]\!]_\sigma \circ (\sigma(\mathtt{up}) = \mathtt{Key}) \oplus (\sigma(\mathtt{down}) = \lambda\mathtt{v}.\mathtt{v}(\sigma)) \oplus \rho\}$$

$$\{\rho_1 :\ \rho_1(\mathtt{f_1}) = \sigma_1(\mathtt{f_1}) \oplus \ldots \oplus \rho_\mathtt{n}(\mathtt{f_n}) = \sigma_1(\mathtt{f_n}) \oplus \rho\}$$

$$\{(\mathtt{l_1}, \mathtt{s_1}) = \mathtt{allocate}(\mathtt{ADT}, \mathtt{s})\}$$

$$\{\mathtt{s_2} = \mathtt{upd}(\mathtt{l_1}, \mathtt{up}(\sigma_1, \perp_{\mathtt{Mem}}), \mathtt{s_1}), \rho_2 = \mathtt{bind}(\mathtt{I}, \mathtt{l_1}, \rho 1)\}$$

$$(\rho_2, \mathtt{s_2})$$

- allocate(ADT, s) allocates, regardless of the specific ADT, a location suitable to contain an generic abstract value
- The Abstract Values, of any types, occupy a same amount of space for allocating the value returned by up.

ADT: Semantics/4

Example

Apply it to the ADT `Q`, we introduced for cartesian points,
ADT Q:

```
  int × int → Q mk;  Q → int fst;  Q → int snd;  Q × int → () m1;
  {type P = Rec int x, int y End;
   mk(int x, int y){P u = Talloc(P); u.x=x; u.y=y; up(u)};
   fst(Q x){down(x).x};
   snd(Q x){down(x).y};
   m1(Q x, int v){down(x).x=v;}
  }
```

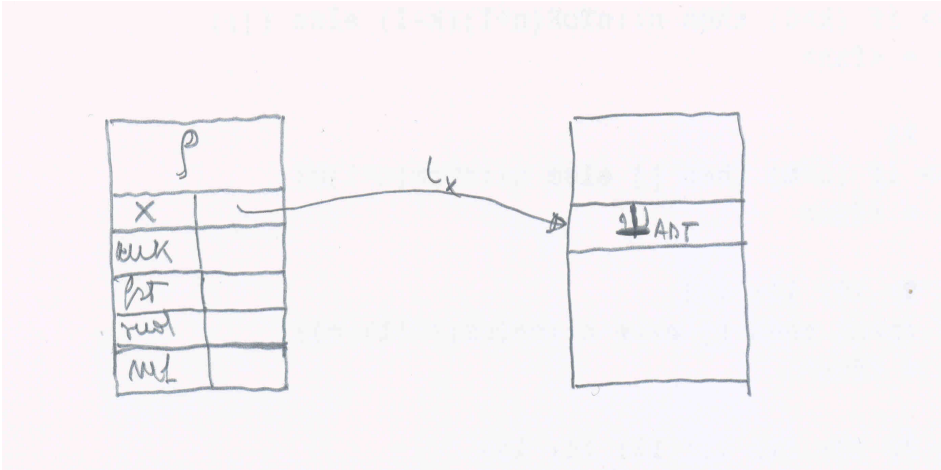
and to the code below:

```
  Q x;
  x = mk(3,5); ...  fst(x)...
```

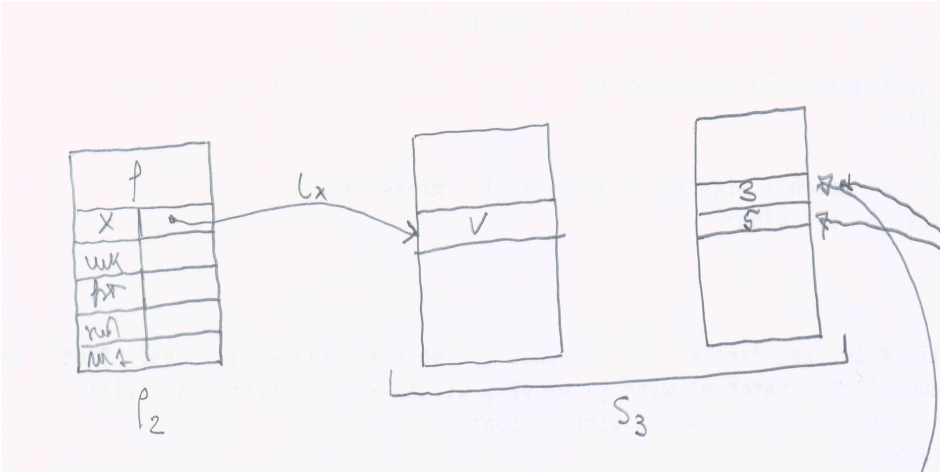
$$\mathcal{D}[\![\texttt{Q x}]\!]_\rho(\texttt{s}) = (\rho_2, \texttt{s}_2)$$
$$\texttt{where:}$$
$$\rho_2(\texttt{x}) = \texttt{l}_\texttt{x} \oplus \rho_2(\texttt{mk}, \texttt{fst}, \texttt{snd}, \texttt{m1}) = \sigma_1(\texttt{mk}, \texttt{fst}, \texttt{snd}, \texttt{m1}) \oplus \rho$$
$$\texttt{s}_2 = \texttt{upd}(\texttt{l}_\texttt{x}, \lambda\texttt{key.if}(\ldots), \texttt{s}_1)$$
$$\texttt{where: } \texttt{s}_1 = \texttt{allocate}(\texttt{ADT}, \texttt{s})$$
$$\mathcal{E}[\![\texttt{Call mk(byValue 3, byValue 5)}]\!]_{\rho_2}(\texttt{s}_2) = (\texttt{v}, \texttt{s}_3)$$
$$\texttt{where:}$$
$$(\texttt{v}, \texttt{s}_3) = \texttt{g}(3,5)(\texttt{s}_2), \texttt{where F(g)} = \rho_2(\texttt{mk})$$
$$\texttt{v} = \texttt{up}(\texttt{u}) = \lambda\texttt{key.if}(\texttt{key} = \sigma_1, (\sigma_1, \texttt{E}_\texttt{M}(\gamma)), \perp_{\texttt{Mem}})$$
$$\textit{for}\ \ \gamma : \gamma(x) = \texttt{l}_\texttt{x} \oplus \gamma(y) = \texttt{l}_\texttt{y} \oplus \texttt{empty}$$
$$\texttt{s}_3 : \texttt{s}_3(\texttt{l}_\texttt{x}) = 3 \wedge \texttt{s}_3(\texttt{l}_\texttt{y}) = 5 \wedge (\forall \texttt{l}, \texttt{s}_2(\texttt{l}) \neq \perp_{\texttt{Mem}} \supset \texttt{s}_3(\texttt{l}) = \texttt{s}_2(\texttt{l}))$$
$$\ldots$$
$$\texttt{to be completed...}$$

ADT: Semantics/5

- A graphical view of the solution:



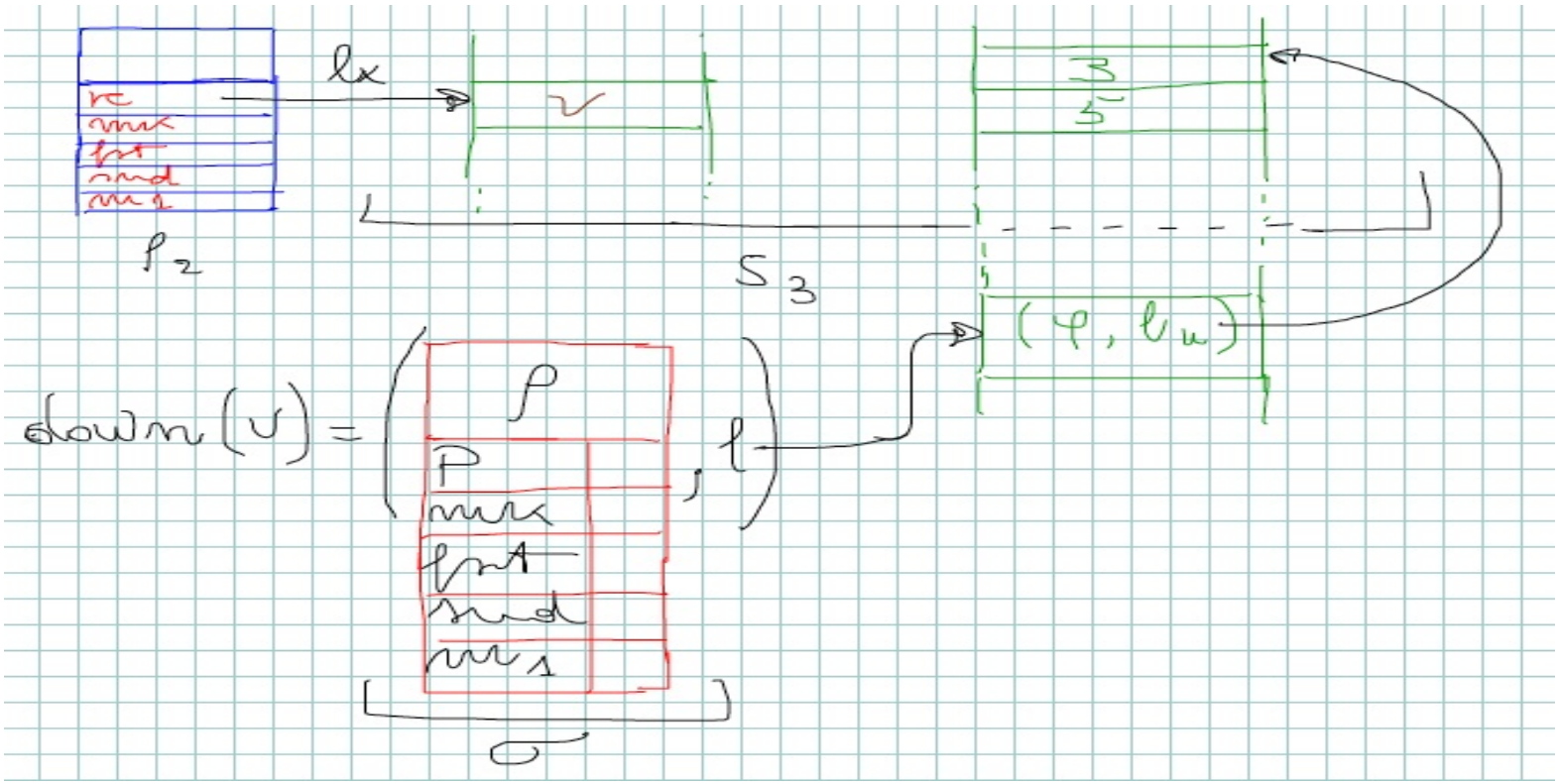
- `Q x;`



- `x = mk(3,5);`

ADT: Semantics/6

- A graphical view of the solution:
- `x = mk(3,5);` A complete view:



ADT: API in Ocaml

An API in Ocaml for an ADT of RELAZIONE (Binary, Polymorphic)

Example

```
module type RELAZIONE =
 sig type ('a,'b) relazione
  val relazioneC: unit -> ('a,'b) relazione
  val isUno:  ('a,'b) relazione -> 'a -> bool
  val isDue:  ('a,'b) relazione -> 'b -> bool
  val getUno:  ('a,'b) relazione -> 'b -> 'a list
  val getDue:  ('a,'b) relazione -> 'a -> 'b list
 end;;
```

- It introduces a module API of name RELAZIONE for an ADT with name ('a,'b) `relazione`: It is a polymorphic type in the type variables a and b.
- The module contains, in the order, the signature of the defined type and that of each of the operations that are "public" and can be used outside of the ADT.

ADT: Module ADT in Ocaml/1

An ADT in Ocaml for the API RELAZIONE of the type `('a,'b) relazione`

Example

```
module Relazione =
 (struct
  type ('a,'b) relazione = ('a * 'b) list
  let relazioneC = []
  let isUno(r,x) = List.fold_right (||) (List.map(fun u -> fst(u)=x) r) false
  let isDue(r,y) = List.fold_right (||) (List.map(fun u -> snd(u)=x) r) false
  let getUno (r,y) = let g = fun u ->if(snd(u)=y)then[fst(u)]else[]
                     in List.fold_right(List.append)(List.map g r) []
  let getDue (r,x) = let g = fun u ->if(fst(u)=x)then[snd(u)]else[]
                     in List.fold_right(List.append)(List.map g r) []
end:RELAZIONE);;
```

- It furnishes an implementation, ADT, of the abstract type of name `('a,'b) relazione`.
- It uses a module with name RELAZIONE which contains the `struct...end` construct.
- The module contains, in the order, the implementation of the type (and the definition of the possibly auxiliary, required types), and of the operations that are visibles outside of the ADT, and of the possibly auxiliary. required operations.

ADT: Module ADT in Ocaml/2

Example

```
module Relazione =
 (struct
  type ('a,'b) relazione = ('a * 'b) list
  let relazioneC = []
  let isUno(r,x) = List.fold_right (||) (List.map(fun u -> fst(u)=x) r) false
  let isDue(r,y) = List.fold_right (||) (List.map(fun u -> snd(u)=x) r) false
  let getUno (r,y) = let g = fun u ->if(snd(u)=y)then[fst(u)]else[]
                     in List.fold_right(List.append)(List.map g r) []
  let getDue (r,x) = let g = fun u ->if(fst(u)=x)then[snd(u)]else[]
                     in List.fold_right(List.append)(List.map g r) []
end:RELAZIONE);;
```

- The module contains, in the order, the implementation of the type (and the definition of the possibly auxiliary, required types), and of the operations that are visible outside of the ADT, and of the possibly auxiliary, required operations.
- All the auxiliary definitions are not visible outside of the module.
- `struct...end:A` qualifies ADT as relative to the module API of name A
- An API may have more than one ADT: Each ADT furnishes a distinct implementation

API and ADT: Implementation

- A repository contains all the definitions of the ADT that are accessible through a key that is unique for each ADT
- The repository, the keys and the functions up and down, for each key, and their insertions in the abstract syntax may be generated by the compiler.

Example

The ADT showed in the previous slides for `('a,'b)relazione` contains an error. The two modules, in fact, API and ADT, are espressed in Ocaml but the analyzer of Ocaml does not recognize such an ADT as an implementation of the API.
(1) Say what error is contained in the ADT;
(2) Say how the API could be modified in a way that the ADT, given in the slides, be an implementation of it.