Lecture 9-10
Expressions: Formalization, Use, Implementation

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

March 14, 2014

Expressions: Formalization, Use, Implementation

- Expressions: Referential Transparency and Side Effects
- Divergent Expressions: strict and non-strict operators (functions)
- Structured Values, Lazy Constructors, Eager and Lazy Evaluation (Evaluators)
- Finite, Finitary and Infinite Values
- Mutable Values and Assignment (operator)

Expressions: Referential Transparency and Side Effects/1

- Expressions with Referential Transparency.
  - Expressions can be replaced by the value without affecting program behaviour
    - Pure Functional Languages
    - Semantic Function

      $\mathcal{E}: \ \mathtt{E} \rightarrow \mathtt{Env} \rightarrow \mathtt{State} \rightarrow \mathtt{Val}$

Example

Expression $e_1 + e - e_1$ can be replaced by e provided that:
1) $e_1 \in \mathtt{Val}$ (i.e. its evaluation terminates and computes a value);
2) Operators $+$ e $-$ stand for addition and subtraction, resp..

Expressions: Referential Transparency and Side Effects/1

- Expressions with Side Effects
  - Expressions do not compute only values: In addition they may modify the computation state
  - Procedural Languages
  - Funzione Semantica

$$\mathcal{E}: \; \mathtt{E} \rightarrow \mathtt{Env} \rightarrow \mathtt{State} \rightarrow (\mathtt{Val} \times \mathtt{State})$$

Example

Expression $e_1 + e - e_1$ cannot be replaced by e even when (1) and (2) of previous slide hold.
Show a concrete case.

- **Use**
  - to Introduce, in the program, values possibly as the result of more or less complicated compositions of (primitive or user defined) operators on values or identifiers (e.g. naming).
  - to Implement algorithms that are based on (mutable and immutable) value manipulations
  - In Functional Languages, all the computable functions are expressed only through programs that implement algorithms that are based on value manipulations (state modification sequences are not needed)

Expressions: Implementation

- **Compile Time.** Decomposition of expressions in a sequence of elementary applications (for state based machines)
  - **Only one operator** is involved in each application of the sequence
  - **Decomposition** may lead to different forms:
    - Infix Notation. Abstract Tree and Depth-First Visit: $3 * x + 2$ becomes $tree(tree(3, *, x), +, 2)$.
    - Prefix Notation. Inner Sequencing: $3 * x + 2$ become the sequence $[+, *, 3, x, 2]$.
    - Postfix Notation. Sequencing: $3 * x + 2$ become the sequence $[3, x, *, 2, +]$ – reversed is perfect for the stack of the intermediate values
    - Machine Code: $3 * x + 2$ become the sequence [iconst_3, iload_0,imul,iconst_2,iadd]
- **RI Stack**: A workspace, contained in each AR, for dealing with operator applications
- **AR Stack**: When user defined operations occur in the expression

Expressions: Divergent E.

- Expressions do not compute always values

Example

What does the C expression, below compute?

$$0 * \mathit{fact}(-3)$$

when *fact* is the user defined C procedure:

$\texttt{int}\ \mathit{fact}(\texttt{int}\ n)\{\texttt{return}((n == 0)\ ?\ 1 : n * \mathit{fact}(n-1));\}$

- To deal with divergent expressions, $\mathcal{E}$ becomes:
  - $\mathcal{E}: \ \texttt{E} \rightarrow \texttt{Env} \rightarrow \texttt{State} \rightarrow \texttt{Val}_{\perp}$
  - $\mathcal{E}: \ \texttt{E} \rightarrow \texttt{Env} \rightarrow \texttt{State} \rightarrow (\texttt{Val} \times \texttt{State})_{\perp}$
- Divergent Expressions augment the Language Expressivity with:
  - the non-strict evaluation form (call by need, ...)
  - the lazy evaluation form and
  - the finitary, infinite values.

Expressions: Strict and non-strict Operators (functions)/1

Example

What do the two C expressions compute?

$$(x < 0) || (\mathit{fact}(x) > 0)$$
$$(x < 0) | (\mathit{fact}(x) > 0)$$

- Can lead to non-terminating computations:
  - $\mathcal{E}: \ \mathtt{E} \longrightarrow \mathtt{Env} \longrightarrow \mathtt{State} \longrightarrow \mathtt{Val}_{\perp}$
  - $\mathcal{E}: \ \mathtt{E} \longrightarrow \mathtt{Env} \longrightarrow \mathtt{State} \longrightarrow (\mathtt{Val} \times \mathtt{State})_{\perp}$
- Some languages have operators
  - **non-strict**: $||, \&\&\ldots$
  - **strict**: $|, \&\ldots$
- Semantics, Implementation, Use

Expressions: Strict and non-strict Operators (functions)/2

Semantics of op: $\texttt{Val}^k \rightarrow \texttt{Val}$

- Definition of a strict, $\mathrm{op}_{\perp_S}$, and of a non-strict., $\mathrm{op}_{\perp_{NS}}$, operator of op.
- $\texttt{Val}$ is extended into: $\texttt{Val}_\perp$
- Strict $\mathrm{op}_{\perp_S}$. All the arguments are evaluated to a value of the op domain.

$$\mathrm{op}_{\perp_S}(e_1, ..., e_k) = \begin{cases} \mathrm{op}(e_1, ..., e_k) & \text{sse } (\forall i \in [1..k])\ e_i \in \texttt{Val} \\ \perp_{\texttt{Val}} & \text{otherwise} \end{cases}$$

- Non-Strict $\mathrm{op}_{\perp_{NS}}$. Only some arguments are evaluated. The following must hold: $e_i \in \texttt{Val}\ (\forall i \in [1..k]) \Rightarrow \mathrm{op}_{\perp_{NS}}(e_1, ..., e_k) = \mathrm{op}(e_1, ..., e_k)$

Example

Let op $= \star$ be integer product. Then
$\star_{\perp_S}$ is the only one possible strict extension of op.
$\star_{\perp_{NS}}$ could be any of the following non-strict extensions of op:
$\star_{\perp_{NS}} = \lambda(\mathtt{x}, \mathtt{y}).\mathtt{if}\ (\mathtt{x} == 0)\ \mathtt{then}\ 0\ \mathtt{else} \star_{\perp_S} (x, y)$
$\star_{\perp_{NS}} = \lambda(\mathtt{x}, \mathtt{y}).\mathtt{if}\ (\mathtt{x} == 0\ ||\ \mathtt{y} == 0)\ \mathtt{then}\ 0\ \mathtt{else} \star_{\perp_S} (x, y)$

Expressions: Strict and non-strict Operators (functions)/3

Implementation

- Primitive Operators:
  - Strict: Apply only to terms of `Val`
  - Non-Strict: Apply also to some unevaluated arguments
- Functions (i.e. user defined operators):
  - Strict: call by value
  - Non-Strict: call by name / call by need

Example

The following Haskell expression, `h 3 (f 5)`, when `h` e `f` are:

```
h = \x y -> if (x\=0) then x else y
f n = f(n+1)
```

evaluates to 3.
Can You rephrase it in OCaml, C or Java?

Use

Non-Strict Functions are:

- Useful in Programming (to cope with certain algorithms)
- Not always computable

Example

A non-strict but non-computable function is *Halting*. The function could be described in this form:

- Naming: halting
- Type: $\texttt{Val}_\perp \longrightarrow \texttt{Bool}$
- Behaviour: When it applies to an expression, it results `true` if the expression computes a value, i.e. a term of `Val`. it results `false` if the evaluation of the expression, diverges.
- Use. it predicts when evaluation is non-terminating.

Expressions: Eager vs. Lazy Evaluation/1

- Structured Values further extend the notion of *non-strict computation*
  - Constructors of values
  - Operators for the component selection (and modification, when mutable)
- **Lazy**. Constructors do not evaluate arguments;
- **Eager**. Constructors, as well as any other operation, apply only to values.

Example

The Haskell expression `g [4,(f 5)]`, when `h` and `f` are:

```
g u = if ((head u)==0) then 3 else 7
f n = f(n+1)
```

computes 7.
Can You rephrase it in OCaml, C or in Java?

Espressioni: Lazy Evaluation/2

- Structured Values further extend the notion of *non-strict computation*
- **Use**
  - **Delayed Evaluation** of the expressions that are components of a structured value
  - **Storable Values** include expressions that are components of a structured value
  - **Infinite Values** (finitely approximated) may be introduced in programming
  - **Finitary Infinite Values** (i.e. infinite structures that have a finite representation) may be introduced as full value, i.e. without using pointers and *ciclic structures*
- **Implementation** Many, different implementations that extend call by-need

Expressions: Lazy Evaluation/3

Example

Finitely Approximated, Infinite Values:

```
nat n = n:nat(n+1)
naturali = nat 0
v = take 3 naturali
```

In Haskell, the 3 expressions above, compute one function, the infinite list of naturals, the list of the first 3 naturals.
Can You rephrase it in Caml, C or Java?

Example

Finitary Infinite values:

`data Tree a = T(a,Tree a)` - it defines a polymorphic type of Haskell

`treeM = T(3,treeM)` - a value of Haskell

treeM computes a infinite tree that can be finitely represented (with pointers !?)
Can You rephrase it in Caml, C or Java?

Expressions: Denotable E./1

Expressions `E` that compute mutable values have 2 different interpretations.

- Storable Value. It is a reference to the associated storable value: `Val(E)`
- Denotable Value. It is a reference to the full mutable value (it is used in modification): `Den(E)`
- Abstract Syntax points out such a distinction in programs

Example

In C (and other procedural languages) the following declaration:

```
int x;
```

introduces a mutable value named `x`.
The abstract syntax distinguishes the different interpretations of expression `x` by using:
`Den(x)` to express the mutable value (sometime called, l-value), and
`Val(x)` to express the storable value (sometime called, r-value) associated to `Den(x)`.

Example

Consider the following C expression:

```
z = x = y
```

The abstract syntax of it, resulting from the compiler or interpreter front-end, in the notation adopted in the provious slides is:

```
Val(Den(z) = Val(Den(x) = Val(y)))
```

Do the same with the following C expression:

```
A[*v+j] = x = y + A[*v+1]
....
```

Expressions: with S.E./1

In Procedural Programming, Expressions do not compute only values: In addition they may modify the computation state.

Table9 – Semantics of Espressions with S.E. – 1
Domini Sintattici
$\mathtt{D} ::= \ldots \mid \mathtt{Var\ I;} \mid \mathtt{Var\ I} = \mathtt{E;} \mid \ldots$
$\mathtt{E} ::= \mathtt{Val(E)} \mid \mathtt{Den(E)} \mid \mathtt{I} \mid \mathtt{VL} \mid op_k(\mathtt{E}_1 \ldots \mathtt{E}_k) \mid \mathtt{E} = \mathtt{E} \mid \ldots$
Semantic Domains
$\mathtt{Env}, \rho, \delta \equiv \ldots$
$\mathtt{Store} \equiv (\mathtt{Loc} \times (\mathtt{Loc} \to \mathtt{Mem}_\perp))$ (*store with allocation*)
$\mathtt{Store}_\perp, \mathtt{s}, \mathtt{r} \equiv \mathtt{Store} + \{\perp_S\}$ (*finite store*)

Finite Store: Operations

$$\texttt{Store} \equiv (\texttt{Loc} \times (\texttt{Loc} \to \texttt{Mem}_\perp))$$
$$\texttt{Store}_\perp, \texttt{s}, \texttt{r} \equiv \texttt{Store} + \{\perp_S\}$$

$$\texttt{upd} : \texttt{Loc} \times \texttt{Mem}_\perp \times \texttt{Store}_\perp \to \texttt{Store}_\perp$$
$$\texttt{upd}(l, m, (L, u))$$
$$\equiv \texttt{if}((l \in [0, L)), \lambda v \,.\, \texttt{if}((v \ \texttt{eq} \ l), m, u(l)), \perp_S)$$

$$\texttt{look} : \texttt{Loc} \times \texttt{Store}_\perp \to \texttt{Mem}_\perp$$
$$\texttt{look}(l, (L, u)) \equiv \texttt{if}((l \in [0, L)), u(l), \perp_M)$$

$$\texttt{allocate} : \texttt{Store}_\perp \to (\texttt{Loc} \times \texttt{Store}_\perp)$$
$$\texttt{allocate}_k((L, u))$$
$$\equiv \texttt{if}((L > k), \perp_{LS}, (L, (L + 1, upd(L, \perp_M, u))))$$

$$\perp_S : \texttt{Store}_\perp$$
$$\perp_S \equiv (\texttt{Y}f \,.\, \lambda \texttt{x} \,.\, f(\texttt{x}))(u), \qquad con \ u \in \texttt{Store}$$

Semantic Functions

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

$$\begin{array}{l}
\mathcal{D}[\![\mathtt{Var\ I;}\ ]\!]_\rho(s) \\
\quad = \mathtt{Let}\{(l, s_l) = \mathit{allocate}(s)\}\ (\mathtt{bind}(\mathtt{I}, l, \rho), s_l) \\
\mathcal{D}[\![\mathtt{Var\ I = E;}\ ]\!]_\rho(s) \qquad\qquad\qquad - \textit{compile time} \\
\quad = \mathtt{Let}\{(l, s_l) = \mathit{allocate}(s)\} \\
\qquad\qquad \{(v_e, s_e) = \mathcal{E}[\![\mathtt{E}]\!]_\rho(s_l)\} \\
\qquad\qquad \{s_m = \mathtt{upd}(l, v_e, s_l)\}\ (\mathtt{bind}(\mathtt{I}, l, \rho), s_m)
\end{array}$$

Expressions: with S.E./3 - Values

Semantic Functions

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

$$\mathcal{E}[\![\mathtt{Val(E)}]\!]_\rho(\mathtt{s}) = \begin{cases} (\mathtt{MV}(\mathtt{s}(\rho(\mathtt{I}))), \mathtt{s}) & \text{if } \mathtt{E} \equiv \mathtt{I} \in \mathtt{Ide} \;\&\; \rho(\mathtt{I}) \in \mathtt{Loc} \\ (\mathtt{DV}(\rho(\mathtt{I})), \mathtt{s}) & \text{if } \mathtt{E} \equiv \mathtt{I} \in \mathtt{Ide} \;\&\; \rho(\mathtt{I}) \notin \mathtt{Loc} \\ (\mathtt{MV}(\mathtt{s}_1(\mathtt{l})), \mathtt{s}_1) & \text{if } \mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s}) = (\mathtt{l}, \mathtt{s}_1) \;\&\; \mathtt{l} \in \mathtt{Loc} \\ (\mathtt{v}, \mathtt{s}_\mathtt{v}) & \text{if } \mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s}) = (\mathtt{v}, \mathtt{s}_\mathtt{v}) \;\&\; \mathtt{v} \notin \mathtt{Loc} \end{cases}$$

$$\mathcal{E}[\![\mathtt{Den(E)}]\!]_\rho(\mathtt{s}) = \begin{cases} (\rho(\mathtt{I}), \mathtt{s}) & \text{if } \mathtt{E} \equiv \mathtt{I} \in \mathtt{Ide} \;\&\; \rho(\mathtt{I}) \in \mathtt{Loc} \\ (\mathtt{l}, \mathtt{s}_1) & \text{if } \mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s}) = (\mathtt{l}, \mathtt{s}_1) \;\&\; \mathtt{l} \in \mathtt{Loc} \\ (\perp_D, \mathtt{s}) & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![\mathtt{VL}]\!]_\rho(\mathtt{s}) = (\mathtt{IntoVal(VL)}, \mathtt{s})$$

Noting that: $\mathcal{E}[\![\mathtt{I}]\!]_\rho$ is no more defined since both $\mathcal{E}[\![\mathtt{Val(I)}]\!]_\rho$ and $\mathcal{E}[\![\mathtt{Den(I)}]\!]_\rho$ are defined instead.

Operators that propagate S.E. but do not generate it.

Semantic Functions

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

$$\mathcal{E}[\![op_k(\mathrm{E}_1 ... \mathrm{E}_k)]\!]_\rho(s) \qquad (strict)$$
$$= \mathtt{Let}\{(v_1, s_1) = \mathcal{E}[\![\mathrm{E}_1]\!]_\rho(s)\}$$
$$...$$
$$\{(v_k, s_k) = \mathcal{E}[\![\mathrm{E}_k]\!]_\rho(s_{k-1})\}\ (op_{\perp_S}(v_1 ... v_k), s_k)$$

$$\mathcal{E}[\![op_k(\mathrm{E}_1 ... \mathrm{E}_k)]\!]_\rho(s) \qquad (non\ -strict)$$

Expressions: with S.E./4 - Operators-2

Operators that propagate S.E. but do not generate it.

$$\mathcal{E}[\![op_k(\mathrm{E}_1 ... \mathrm{E}_k)]\!]_\rho(s) \qquad\qquad (non - strict)$$
$$= \mathtt{Let}\{(v_{i_1}, s_{i_1}) = \mathcal{E}[\![\mathrm{E}_{i_1}]\!]_\rho(s)$$
$$...$$
$$\{(v_{i_h}, s_{i_h}) = \mathcal{E}[\![\mathrm{E}_{i_h}]\!]_\rho(s_{i_{h-1}})\}\ (op_{\perp_{NS}}(\overline{v}_1 ... \overline{v}_k), s_{i_h})$$

$$where: (i_1 \neq ... \neq i_h \in [1..k])\ and\ (\overline{v}_{i_j} \equiv\ v_{i_j}(\forall j \in [1..h]))$$

Example

Consider a non-strict operator with the following behavior:
let op = fun x y z w $\longrightarrow$
if (w=0) then 0 else if (w>y) then y else if ...
Answering: 1) Which arguments does it evaluate? 2) In what order are they evaluated? 3) In what state is each argument evaluated?

Operators that generate S.E.

Semantic Functions

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

$$\begin{aligned}&\mathcal{E}[\![\mathtt{E}_l = \mathtt{E}_r]\!]_\rho(s)\\ &\quad = \mathtt{Let}\{(v_1, s_1) = \mathcal{E}[\![\mathtt{E}_r]\!]_\rho(s)\}\\ &\qquad\qquad \{(l_2, s_2) = \mathcal{E}[\![\mathtt{E}_l]\!]_\rho(s_1)\}\ (v_1, \mathtt{upd}(l_2, \mathtt{VM}(v_1), s_2))\end{aligned}$$

Auxiliary Functions

$\mathtt{L} : \mathtt{Loc} \rightarrow \mathtt{Den}$ $\qquad (\textit{injection}, \textit{i.e.constructor})$

$\mathtt{DV} : \mathtt{Den} \rightarrow \mathtt{Val}$

$\mathtt{VM} : \mathtt{Val} \rightarrow \mathtt{Mem}$