Data and Mutable Store

Massimo Merro

14 November 2017

- So far we have only looked at very simple basic data types: int, bool, unit, and functions over them.
- Let us explore now more structured data, maintaining them in the simplest form as possible, and revisit the semantics of mutable store.
- We start with two basic structured data: product and sum type.
- The product type $T_1 * T_2$ allows us you to tuple together values of type T_1 and T_2. In C this is done with **structs**; while in Java one can use a class.
- The sum type $T_1 + T_2$ lets you form a *disjoint union*, with a value of the sum type either being a value of type T_1 or a value of type T_2. In C this is done using **unions**, while in Java a class can implement more interfaces (although it can extends only one class).
- In most languages these features appear in richer forms: *labelled records* rather than simple products, or *labelled variants*, or ML *datatypes* with named *constructors*, rather than simple sums.

Products

Let us extend the grammars for expressions and types:

$$\begin{array}{rcl} e & ::= & \ldots \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e \\ T & ::= & \ldots \mid T_1 * T_2 \end{array}$$

Design choices (simplifications):

- pairs, not arbitrary tuples: we have both $\mathsf{int} * (\mathsf{bool} * \mathsf{unit})$ and $(\mathsf{int} * \mathsf{bool}) * \mathsf{unit}$, but we don't have $\mathsf{int} * \mathsf{bool} * \mathsf{unit}$;
- we have projections $\#1$ and $\#2$, not *pattern matching*;
- we don't have $\#e\ e'$ (cannot be typed).

Products - typing

$$\text{(pair)}\ \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 * T_2}$$

$$\text{(proj1)}\ \frac{\Gamma \vdash e : T_1 * T_2}{\Gamma \vdash \#1\ e : T_1}$$

$$\text{(proj2)}\ \frac{\Gamma \vdash e : T_1 * T_2}{\Gamma \vdash \#2\ e : T_2}$$

Products - operational semantics

Let us extend the possible *values* as follows:

$$v \quad ::= \quad \dots \quad \mid \quad (v_1, v_2)$$

$$(\text{pair1}) \ \frac{\langle e_1, s\rangle \rightarrow \langle e_1', s'\rangle}{\langle (e_1, e_2), s\rangle \rightarrow \langle (e_1', e_2), s'\rangle}$$

$$(\text{pair2}) \ \frac{\langle e_2, s\rangle \rightarrow \langle e_2', s'\rangle}{\langle (v, e_2), s\rangle \rightarrow \langle (v, e_2'), s'\rangle}$$

$$(\text{proj1}) \ \frac{-}{\langle \#1\ (v_1, v_2)\ ,\ s\rangle \rightarrow \langle v_1, s\rangle} \qquad (\text{proj2}) \ \frac{-}{\langle \#2\ (v_1, v_2)\ ,\ s\rangle \rightarrow \langle v_2, s\rangle}$$

$$(\text{proj3}) \ \frac{\langle e, s\rangle \rightarrow \langle e', s'\rangle}{\langle \#1\ e\ ,\ s\rangle \rightarrow \langle \#1\ e'\ ,\ s'\rangle} \qquad (\text{proj4}) \ \frac{\langle e, s\rangle \rightarrow \langle e', s'\rangle}{\langle \#2\ e\ ,\ s\rangle \rightarrow \langle \#2\ e'\ ,\ s'\rangle}$$

We have chosen left-to-right evaluation order for consistency.

Sums (or Variants, or tagged Unions)

Let us extend the grammars for expressions and types:

$$
\begin{array}{rcl}
e & ::= & \ldots \;\mid\; \mathsf{inl}\ e : T \;\mid\; \mathsf{inr}\ e : T \;\mid \\
 & & \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\,(x_1 : T_1) \Rightarrow e_1 \mid \mathsf{inr}\,(x_2 : T_2) \Rightarrow e_2 \\
T & ::= & \ldots \;\mid\; T_1 + T_2
\end{array}
$$

Note that x_1 and x_2 are bound in e_1 and e_2, respectively.

Sums - typing

$$(\text{inl})\ \frac{\Gamma \vdash e : T_1}{\Gamma \vdash (\text{inl}\ e{:}T_1 + T_2) : T_1 + T_2}$$

$$(\text{inr})\ \frac{\Gamma \vdash e : T_2}{\Gamma \vdash (\text{inr}\ e{:}T_1 + T_2) : T_1 + T_2}$$

$$(\text{case})\ \frac{\Gamma \vdash e : T_1 + T_2 \quad \Gamma, x_1 : T_1 \vdash e_1 : T \quad \Gamma, x_2 : T_2 \vdash e_2 : T}{\Gamma \vdash (\text{case}\ e\ \text{of inl}(x_1 : T_1) \Rightarrow e_1 \mid \text{inr}(x_2 : T_2) \Rightarrow e_2)\ :\ T}$$

Sums - type annotations

Why do we have in the syntax type annotations for sums?

To maintain the *Uniqueness typing property*, i.e. each expression e, if typable, must have a unique type T in an environment Γ such that $\Gamma \vdash e : T$.

Without type annotations we would have:

$$\text{inl } 3 \text{ of type } \text{int} + \text{int}, \text{ but also}$$

$$\text{inl } 3 \text{ of type } \text{int} + \text{bool}$$

and, more generally:

$$\text{inl } 3 \text{ of type } \text{int} + T, \ \text{ for any type } T$$

Sums - operational semantics (1)

Let us extend the grammar of values as follows:

$$v \quad ::= \quad \ldots \quad \mid \quad \mathsf{inl}\ v : T \quad \mid \quad \mathsf{inr}\ v : T$$

Let us extend the operational semantics:

$$(\mathsf{inl})\ \frac{\langle e, s\rangle \rightarrow \langle e', s'\rangle}{\langle \mathsf{inl}\ e{:}T\ ,\ s\rangle \rightarrow \langle \mathsf{inl}\ e'{:}T\ ,\ s'\rangle}$$

$$(\mathsf{inr})\ \frac{\langle e, s\rangle \rightarrow \langle e', s'\rangle}{\langle \mathsf{inr}\ e{:}T\ ,\ s\rangle \rightarrow \langle \mathsf{inr}\ e'{:}T\ ,\ s'\rangle}$$

$$(\mathsf{case1})\ \frac{\langle e, s\rangle \rightarrow \langle e', s'\rangle}{\begin{array}{c}\langle \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\,(x_1{:}T_1) \Rightarrow e_1 \mid \mathsf{inr}\,(x_2{:}T_2) \Rightarrow e_2\ ,\ s\rangle \\ \rightarrow \langle \mathsf{case}\ e'\ \mathsf{of}\ \mathsf{inl}\,(x_1{:}T_1) \Rightarrow e_1 \mid \mathsf{inr}\,(x_2{:}T_2) \Rightarrow e_2\ ,\ s'\rangle\end{array}}$$

Sums - operational semantics (2)

$$\text{(case2)}\quad \frac{-}{\langle \mathsf{case\ inl}\ v{:}T\ \mathsf{of\ inl}\,(x_1{:}T_1) \Rightarrow e_1 \mid \mathsf{inr}\,(x_2{:}T_2) \Rightarrow e_2\ ,\ s\rangle \rightarrow \langle e_1\{{}^{v}/_{x_1}\}\ ,\ s\rangle}$$

$$\text{(case3)}\quad \frac{-}{\langle \mathsf{case\ inr}\ v{:}T\ \mathsf{of\ inl}\,(x_1{:}T_1) \Rightarrow e_1 \mid \mathsf{inr}\,(x_2{:}T_2) \Rightarrow e_2\ ,\ s\rangle \rightarrow \langle e_2\{{}^{v}/_{x_2}\}\ ,\ s\rangle}$$

Records

A generalization of products.

Each field is associated with a label.

Labels $lab \in \mathbb{LAB}$ for a set $\mathbb{LAB} = \{\mathrm{p}, \mathrm{q}, \ldots\}$.

Again let us extend the syntax of expressions and types:

$$
\begin{array}{rcllll}
e & ::= & \ldots & \mid & \{lab_1 = e_1, \ldots, lab_k = e_k\} & \mid \quad \# lab\ e \\
T & ::= & \ldots & \mid & \{lab_1 : T_1, \ldots, lab_k : T_k\} &
\end{array}
$$

where in each record (type or expressions) no *lab* occurs more than once.

Records - typing

$$\text{(record)}\quad \frac{\Gamma \vdash e_1 : T_1 \quad \dots \quad \Gamma \vdash e_k : T_k}{\Gamma \vdash \{lab_1 = e_1, \dots, lab_k = e_k\} \ : \ \{lab_1 : T_1, \dots, lab_k : T_k\}}$$

$$\text{(recordproj)}\quad \frac{\Gamma \vdash e : \{lab_1 : T_1, \dots, lab_k : T_k\}}{\Gamma \vdash \# lab_i \ e \ : \ T_i}$$

Here the field order matters so, for example, the expression

$$\big(\text{fn } \mathrm{x} : \{l_1 : \text{int}, l_2 : \text{bool}\} \Rightarrow x\big)\{l_2 = \mathit{true}, l_1 = 17\}$$

is ill-typed.

The same label can be used in different records. In some languages (e.g. OCaml) this is not allowed.

Records - operational semantics

Let us extend the grammar of values as follows:

$$v \quad ::= \quad \dots \quad | \quad \{lab_1 = v_1, \dots, lab_k = v_k\}$$

And the operational semantics:

$$(\text{record1}) \; \frac{\langle e_i, s\rangle \rightarrow \langle e_i', s'\rangle}{\begin{array}{l}\langle \{lab_1 = v_1, \dots, lab_i = e_i, \dots lab_k = e_k\}\,,\, s\rangle \\ \rightarrow \langle \{lab_1 = v_1, \dots, lab_i = e_i', \dots lab_k = e_k\}\,,\, s'\rangle\end{array}}$$

$$(\text{record2}) \; \frac{-}{\langle \#lab_i\;\{lab_1 = v_1, \dots, lab_i = v_i, \dots lab_k = v_k\}\,,\, s\rangle \rightarrow \langle v_i, s\rangle}$$

$$(\text{record3}) \; \frac{\langle e, s\rangle \rightarrow \langle e', s'\rangle}{\langle \#lab\; e\;,\; s\rangle \rightarrow \langle \#lab\; e'\;,\; s'\rangle}$$

Mutable Store

Most languages have some kind of *mutable store*. Two main choices:

1. What we have done in our language is the following:

$$e \;::=\; \ldots \;\mid\; l := e \;\mid\; !l \;\mid\; x$$

- locations store mutable values: we use the assignment construct to change the value associated to a location
- variables refer to a previously-calculated value: once we associate a value to a variable we can not change it anymore
- explicit dereferencing for locations only

$$\mathsf{fn}\ x : \mathsf{int} \Rightarrow \mathrm{l} := !\mathrm{l} + \mathrm{x}; \ldots$$

2. in other language like C and Java:
   - variables let you refer to a previously calculated value and you can *overwrite* that value with another one
   - implicit dereferencing. The function of the previous slide becomes in Java:

$$\text{void foo}(\mathtt{x} : \text{int})\{\mathtt{l} := \mathtt{l} + \mathtt{x}; \ldots\}$$

   - have some limited type machinery to limit mutability.

In our language we are staying with option 1.

Extending the store

In the following we overcome some limitations on references of our language. In particular, we recal that, at the moment:

- We can only store integers value
- We cannot create new locations (they are statically determined)
- We cannot write functions that abstracts on locations, such as

$$\mathsf{fn}\ \mathrm{l} : \mathsf{intref} \Rightarrow !\mathrm{l}$$

Let us extend syntax and types to overcome these limitations:

$$
\begin{array}{rcl}
e & ::= & \ldots \mid \cancel{l}{:=}e \mid !\cancel{l} \mid e_1{:=}e_2 \mid !e \mid \mathsf{ref}\ e \mid l \\
T & ::= & \ldots \mid \mathsf{ref}\ T \\
T_{loc} & ::= & \mathsf{in}\cancel{\mathsf{t}}\mathsf{ref} \mid \mathsf{ref}\ T
\end{array}
$$

References - Typing

$$(\text{ref})\ \frac{\Gamma \vdash e : T}{\Gamma \vdash \text{ref}\ e : \text{ref}\ T}$$

$$(\text{assign})\ \frac{\Gamma \vdash e_1 : \text{ref}\ T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash (e_1 := e_2) : \text{unit}}$$

$$(\text{deref})\ \frac{\Gamma \vdash e : \text{ref}\ T}{\Gamma \vdash\ !e : T}$$

$$(\text{loc})\ \frac{-}{\Gamma \vdash l : \text{ref}\ T}\ \Gamma(l) = \text{ref}\ T$$

References - Operational semantics

A locations is a value:

$$v \quad ::= \quad \dots \quad \Big| \quad l$$

Up to now a store s was a finite partial map from $\mathbb{L}$ to $\mathbb{Z}$. From now on,

$$s : \mathbb{L} \rightharpoonup \mathbb{V} \ .$$

Let us see the rules of the semantics:

$$\text{(ref1)} \ \frac{-}{\langle \text{ref } v, s \rangle \twoheadrightarrow \langle l, s[l \mapsto v] \rangle} \ l \notin \text{dom}(s)$$

$$\text{(ref2)} \ \frac{\langle e, s \rangle \twoheadrightarrow \langle e', s' \rangle}{\langle \text{ref } e, s \rangle \twoheadrightarrow \langle \text{ref } e', s' \rangle}$$

Rule (ref1) is for dynamic allocation of memory!

$$\text{(deref1)}\quad \frac{-}{\langle !l, s\rangle \twoheadrightarrow \langle v, s\rangle}\ \text{if } l \in \mathrm{dom}(s) \text{ and } s(l) = v$$

$$\text{(deref2)}\quad \frac{\langle e, s\rangle \twoheadrightarrow \langle e', s'\rangle}{\langle !e, s\rangle \twoheadrightarrow \langle !e', s'\rangle}$$

$$\text{(assign1)}\quad \frac{-}{\langle l := v, s\rangle \twoheadrightarrow \langle \mathsf{skip}, s[l \mapsto v]\rangle}\ \text{if } l \in \mathrm{dom}(s)$$

$$\text{(assign2)}\quad \frac{\langle e, s\rangle \twoheadrightarrow \langle e', s'\rangle}{\langle l := e, s\rangle \twoheadrightarrow \langle l := e', s'\rangle}$$

$$\text{(assign2)}\quad \frac{\langle e_1, s\rangle \twoheadrightarrow \langle e_1', s'\rangle}{\langle e_1 := e_2, s\rangle \twoheadrightarrow \langle e_1' := e_2, s'\rangle}$$

How things change

- An expression of the form ref v has to do something at runtime: should return a *new (fresh) location* associated to the value v
- Functions can abstract over locations: fn x : ref $T \Rightarrow !\mathrm{x}$
- When program starts they don't have locations: they must create new locations at runtime
- Typing and operational semantics permits locations to contain locations, e.g. ref(ref 3)
- In this semantics the Determinacy property is lost, for a technical reason: *new locations are chosen arbitrarly*. To recover Determinacy we would need to work "up to alpha-conversion for locations"
- Within our language you are not allowed to do arithmetic on locations, only assignments (it can be done in C but not in Java) or test whether one is bigger than another
- Our store just grows during computation - in a real programming language we would need a garbage collector.

Type-checking the store

Before introducing references in our type properties we used the condition

$$\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$$

to express that "all locations mentioned in Γ exist in the store s".

Now, with the introduction of references, we need more:

for each $l \in \mathrm{dom}(s)$ we need that $s(l)$ is typable.

Notice that $s(l)$ may contain functions and even some other locations...

Type-checking the store - Example 1

Consider

$$e \;=\; \textbf{let } x : \text{ref bool} = \text{ref } \mathit{true} \textbf{ in}$$
$$\textbf{while } !x \; (\textbf{do } \ldots; x := (\mathit{boolean\ expression}))$$

if the while will exit we will have the following reduction sequence:

$\langle e, \{\}\rangle \rightarrow^*$
$\langle e_1, \{l_1 \mapsto \mathit{true}\}\rangle$[1]$\rightarrow^*$
$\langle e_2, \{l_1 \mapsto \text{false}\}\rangle$

Thus, now, we can write on variables if they refer to locations!

[1] A new location l_1 is created and each occurrence of x is replaced with l_1.

Type-checking the store - Example 2

Consider

$$e \;=\; \textbf{let}\ f : \mathsf{ref}\,(\mathsf{int} \to \mathsf{int}) \;=\; \mathsf{ref}\,(\mathsf{fn}\ \mathrm{z} : \mathsf{int} \Rightarrow \mathrm{z})\ \textbf{in}$$
$$f := (\mathsf{fn}\ \mathrm{z} : \mathsf{int} \Rightarrow \mathsf{if}\ \mathrm{z} \geq 1\ \mathsf{then}\ \mathrm{z} + !f(\mathrm{z} + -1)\ \mathsf{else}\ 0);$$
$$!f\ 3$$

that has the following reduction sequence:

$\langle e, \{\}\rangle \rightarrow^*$
$\langle e_1, \{\mathrm{l}_1 \mapsto (\mathsf{fn}\ \mathrm{z} : \mathsf{int} \Rightarrow \mathrm{z})\}\rangle \rightarrow^*$
$\langle e_2, \{\mathrm{l}_1 \mapsto (\mathsf{fn}\ \mathrm{z} : \mathsf{int} \Rightarrow \mathsf{if}\ \mathrm{z} \geq 1\ \mathsf{then}\ \mathrm{z} + !\mathrm{l}_1(\mathrm{z} + -1)\ \mathsf{else}\ 0)\}\rangle \rightarrow^*$
. . . .
$\langle 6, \{\mathrm{l}_1 \mapsto (\mathsf{fn}\ \mathrm{z} : \mathsf{int} \Rightarrow \mathsf{if}\ \mathrm{z} \geq 1\ \mathsf{then}\ \mathrm{z} + !\mathrm{l}_1(\mathrm{z} + -1)\ \mathsf{else}\ 0)\}\rangle$

where:

$$e_1 \;\equiv\; \mathrm{l}_1 := (\mathsf{fn}\ \mathrm{z} : \mathsf{int} \Rightarrow \mathsf{if}\ \mathrm{z} \geq 1\ \mathsf{then}\ \mathrm{z} + !\mathrm{l}_1(\mathrm{z} + -1)\ \mathsf{else}\ 0); (!\mathrm{l}_1\ 3)$$
$$e_2 \;\equiv\; \mathit{skip}; (!\mathrm{l}_1\ 3)$$

We have made a recursive function without using the fix.*e* operator!

Typing properties

Well-typed store

We write $\Gamma \vdash s$ if

1. $\text{dom}(\Gamma) = \text{dom}(s)$, and
2. for all $l \in \text{dom}(s)$, if $\Gamma(l) = \text{ref } T$ then $\Gamma \vdash s(l) : T$.

Progress (reformulated)

If e is closed and $\Gamma \vdash e : T$ and $\Gamma \vdash s$ then

- either e is a value, or
- there exist e', s' such that $\langle e, s \rangle \twoheadrightarrow \langle e', s' \rangle$.

Type Preservation (reformulated)

If e is closed and $\Gamma \vdash e : T$ and $\Gamma \vdash s$ and $\langle e, s \rangle \twoheadrightarrow \langle e', s' \rangle$ then e' is closed and for some Γ' with disjoint domain to Γ we have $\Gamma, \Gamma' \vdash e' : T$ and $\Gamma, \Gamma' \vdash s'$.