Programming = proving?

The Curry-Howard correspondence today

Eight lecture

Step carefully: step-indexing techniques

Xavier Leroy

Collège de France

2019-01-09

I

Logical relations in operational semantics

Reminder: logical relations

(Lecture of Dec 19th 2018)

A logical relation is a family of relations $R(t)$, indexed by a type t, between two (denotations of) programs, such that

> *Two functions are related at type $t \to s$ if and only if they map arguments related at type t to results related at type s.*

Example:
the functions $\lambda n.\, n + n$ and $\lambda n.\, n \times 2$ are related by $R(\mathtt{int} \to \mathtt{int})$, assuming that $R(\mathtt{int})$ is the identity relation.

An operational semantics framework

In the following, we will not use denotational semantics, but only operational approaches.

Logical relations relate two expressions of the language a_1, a_2.

The semantics is given by a reduction relation $a \to a'$.

$$\begin{array}{lll} a \to a_1 \to \cdots \to a_n \to \cdots & & \text{divergence} \\ a \to a_1 \to \cdots \to v \not\to & v \in \mathsf{Val} & \text{normal termination} \\ a \to a_1 \to \cdots \to a_n \not\to & a_n \notin \mathsf{Val} & \text{termination on an error} \end{array}$$

To simplify even further, we fix a reduction strategy: call by value.

$$(\lambda x.\, a)\ v \to a[x \leftarrow v] \quad \text{if } v \in \mathsf{Val} \qquad (\beta_v \text{ reduction})$$

Operational logical relations

We define two logical relations: $V(t)$ over values

$$
\begin{aligned}
V(\texttt{int}) &= \{\ (n, n) \mid n \text{ integer}\ \} \\
V(t \to s) &= \{\ (\lambda x_1.\, a_1, \lambda x_2.\, a_2) \mid \\
&\qquad\qquad \forall (v_1, v_2) \in V(t),\ (a_1[x_1 \leftarrow v_1], a_2[x_2 \leftarrow v_2]) \in E(s)\ \}
\end{aligned}
$$

and $E(t)$ over expressions (computations)

$$
\begin{aligned}
E(t) = \{\ (a_1, a_2) \mid \forall b_1,\ a_1 \stackrel{*}{\to} b_1 \wedge b_1 \text{ irreducible} \Rightarrow \\
\exists b_2,\ a_2 \stackrel{*}{\to} b_2 \wedge (b_1, b_2) \in V(t)\ \}
\end{aligned}
$$

The definition is well founded by induction over t:
if we expand the definition of $E(s)$ in the definition of $V(t \to s)$, we see that the latter depends only on $V(t)$ and on $V(s)$.

Logical relations and contextual equivalence

Theorem (fundamental theorem of logical relations)

If $x_1 : t_1, \ldots, x_n : t_n \vdash a : t$, the interpretations of a in two related environments are related:

$$\textit{if } (v_i, v_i') \in V(t_i) \textit{ for } i = 1, \ldots, n, \textit{ then } (a[x_i \leftarrow v_i], a[x_i \leftarrow v_i']) \in E(t)$$

Corollary (contextual equivalence)

If $(a_1, a_2) \in E(t)$ and $(a_2, a_1) \in E(t)$,
then for all contexts $C[.]$ of type $t \rightarrow \texttt{int}$ and all integers n,

$$C[a_1] \xrightarrow{*} n \textit{ if and only if } C[a_2] \xrightarrow{*} n$$

(Other corollaries: representation independence if we add abstract types; "theorems for free" if we add polymorphism; see lecture of Dec 19th 2018.)

Unary logical relations

In this operational framework, unary logical relations provide us with an interpretation of types t as sets of values $V(t)$:

$$V(\texttt{int}) = \{\, n \mid n \text{ integer} \,\}$$
$$V(t \to s) = \{\, \lambda x.\, a \mid \forall v \in V(t),\ a[x \leftarrow v] \in E(s) \,\}$$

and as sets of expressions $E(t)$:

$$E(t) = \{\, a \mid \forall b,\ a \xrightarrow{*} b \wedge b \text{ irreducible} \Rightarrow b \in V(t) \,\}$$

Note: an erroneous expression (irreducible but not a value, such as 1 2) does not belong to any $V(t)$. Hence, an expression that terminates on an error (such as $a \xrightarrow{*} 1\ 2$) does not belong to any $E(t)$.

Logical relations and type soundness

Theorem (fundamental theorem of logical relations)

If $x_1 : t_1, \ldots, x_n : t_n \vdash a : t$, and if $v_i \in V(t_i)$ pour $i = 1, \ldots, n$, then $a[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n] \in E(t)$

Corollary (type soundness)

If $\vdash a : t$, then a does not terminate on an error: either a terminates on a value, or a diverges.

Well-typed terms do not go wrong. *(R. Milner)*

II

Recursive types

Non-recursive data types

It is easy to extend the relation V to non-recursive data types such as products $t \times s$ and sums $t + s$:

$$V(t \times s) = \{(v, w) \mid v \in V(t) \wedge w \in V(s)\}$$

$$V(t + s) = \{\texttt{inj}_1(v) \mid v \in V(t)\} \cup \{\texttt{inj}_2(w) \mid w \in V(s)\}$$

The definition of $V(t)$ remains well founded by induction over t.

Inductive types

For inductive types such as lists

```
type 'a list = Nil | Cons of 'a * 'a list
```

we have an apparent circularity:

$$V(t\,\texttt{list}) = \{\texttt{Nil}\} \cup \{\, \texttt{Cons}(v, w) \mid v \in V(t) \land w \in V(t\,\texttt{list}) \,\}$$

However, the definition of $v \in V(t)$ remains well founded: in the case of lists, we do a local induction on the structure of value v; then, a global induction on the structure of type t. In other words:

$$V(t\,\texttt{list}) = \mu X.\ \{\texttt{Nil}\} \cup \{\, \texttt{Cons}(v, w) \mid v \in V(t) \land w \in X \,\}$$

that is,

$$V(t\,\texttt{list}) = \{\, \texttt{Cons}(v_1, \ldots, \texttt{Cons}(v_n, \texttt{Nil})) \mid v_i \in V(t) \,\}$$

General recursive types

Problem: recursive types that are not inductive
(non strictly positive occurrences in the types of constructors)

```
type lam = Lam of (lam -> lam)
```

The "definition" of $V(\mathtt{lam})$ is obviously circular:

$$\begin{aligned} V(\mathtt{lam}) &= \{ \mathtt{Lam}(f) \mid f \in V(\mathtt{lam} \to \mathtt{lam}) \} \\ &= \{ \mathtt{Lam}(\lambda x.\, a) \mid \forall v \in V(\mathtt{lam}),\ a[x \leftarrow v] \in E(\mathtt{lam}) \} \end{aligned}$$

This "definition" is just a fixed point equation, which we cannot solve in set theory, but we can solve in other categories such as Scott domains.
(Recall the domain $D_\infty \approx D_\infty \to_{cont} D_\infty$.)

An indexed model of recursive types

(A. Appel and D. McAllester, TOPLAS(23), 2001)

Appel and McAllester imagined to base the definition of $V(t)$ not by induction on the structure of t, but by induction on another index (a natural number):
the number of reduction steps we allow ourselves to perform on expressions and (applications of) values.

The technique becomes known in the literature under the name of step-indexing.

Intuitions for step indexing

What does it mean, semantically, that expression a has type `int`?

Usual answer:

- if $a \xrightarrow{*} n$ (n integer) or a diverges: yes, a has type `int`;
- if a reduces to an error or to a value that is not an integer: no, a does not have type `int`.

"Step-indexed" answer: for a given number k,

- if, in k steps at most, a reduces to an integer or does not reach a normal form: yes, a seems to have type `int` for k steps;
- if, in k steps at most, a reduces to an error or to a value that is not an integer: no, a does not have type `int`.

In the end, a has type `int` if for all $k \in \mathbb{N}$, a seems to have type `int` for k steps.

An indexed model of recursive types

(A. Appel and D. McAllester, TOPLAS(23), 2001)

Notation: $a \to_k b$ means "a reduces to b in k steps".

$$
\begin{aligned}
V_k(\texttt{int}) &= \{\, n \mid n \text{ integer} \,\} \\
V_k(t \to s) &= \{\, \lambda x.\, a \mid \forall j < k,\ \forall v \in V_j(t),\ a[x \leftarrow v] \in E_j(s) \,\} \\
E_k(t) &= \{\, a \mid \forall j \leq k,\ \forall b,\ a \to_j b \wedge b \text{ irreducible} \Rightarrow b \in V_{k-j}(t) \,\}
\end{aligned}
$$

Intuitions:

- Expression a seems to have type t in k steps if, having spent $j \leq k$ steps to reduce a to b, b seems to be a value of type t for $k - j$ remaining steps.
- An abstraction $\lambda x.a$ seems to be a value of type $t \to s$ in k steps if the application $(\lambda x.a)\ v$ seems to have type t for at most k steps. The β-reduction spends one step, hence $a[x \leftarrow v] \in E_j(s)$ for $j < k$.
- In j steps, expression $a[x \leftarrow v]$ cannot examine value v for more than j steps! Hence, it suffices that v seems to be of type t for j steps.

An indexed model of recursive types

(A. Appel and D. McAllester, TOPLAS(23), 2001)

Notation: $a \to_k b$ means "a reduces to b in k steps".

$$
\begin{aligned}
V_k(\texttt{int}) &= \{\ n \mid n \text{ integer }\} \\
V_k(t \to s) &= \{\ \lambda x.\, a \mid \forall j < k,\ \forall v \in V_j(t),\ a[x \leftarrow v] \in E_j(s)\ \} \\
E_k(t) &= \{\ a \mid \forall j \leq k,\ \forall b,\ a \to_j b \land b \text{ irreducible} \Rightarrow b \in V_{k-j}(t)\ \}
\end{aligned}
$$

Intuitions:

- Expression a seems to have type t in k steps if, having spent $j \leq k$ steps to reduce a to b, b seems to be a value of type t for $k - j$ remaining steps.
- An abstraction $\lambda x.a$ seems to be a value of type $t \to s$ in k steps if the application $(\lambda x.a)\ v$ seems to have type t for at most k steps. The β-reduction spends one step, hence $a[x \leftarrow v] \in E_j(s)$ for $j < k$.
- In j steps, expression $a[x \leftarrow v]$ cannot examine value v for more than j steps! Hence, it suffices that v seems to be of type t for j steps.

An indexed model of recursive types

(A. Appel and D. McAllester, TOPLAS(23), 2001)

Notation: $a \to_k b$ means "a reduces to b in k steps".

$$
\begin{aligned}
V_k(\texttt{int}) &= \{\, n \mid n \text{ integer} \,\} \\
V_k(t \to s) &= \{\, \lambda x.\, a \mid \forall j < k,\ \forall v \in V_j(t),\ a[x \leftarrow v] \in E_j(s) \,\} \\
E_k(t) &= \{\, a \mid \forall j \leq k,\ \forall b,\ a \to_j b \wedge b \text{ irreducible} \Rightarrow b \in V_{k-j}(t) \,\}
\end{aligned}
$$

Intuitions:

- Expression a seems to have type t in k steps if, having spent $j \leq k$ steps to reduce a to b, b seems to be a value of type t for $k - j$ remaining steps.
- An abstraction $\lambda x.a$ seems to be a value of type $t \to s$ in k steps if the application $(\lambda x.a)\ v$ seems to have type t for at most k steps. The β-reduction spends one step, hence $a[x \leftarrow v] \in E_j(s)$ for $j < k$.
- In j steps, expression $a[x \leftarrow v]$ cannot examine value v for more than j steps! Hence, it suffices that v seems to be of type t for j steps.

An indexed model of recursive types

(A. Appel and D. McAllester, TOPLAS(23), 2001)

Notation: $a \to_k b$ means "a reduces to b in k steps".

$$\begin{aligned} V_k(\texttt{int}) &= \{\ n \mid n \text{ integer}\ \} \\ V_k(t \to s) &= \{\ \lambda x.\, a \mid \forall j < k,\ \forall v \in V_j(t),\ a[x \leftarrow v] \in E_j(s)\ \} \\ E_k(t) &= \{\ a \mid \forall j \leq k,\ \forall b,\ a \to_j b \wedge b \text{ irreducible} \Rightarrow b \in V_{k-j}(t)\ \} \end{aligned}$$

Intuitions:

- Expression a seems to have type t in k steps if, having spent $j \leq k$ steps to reduce a to b, b seems to be a value of type t for $k - j$ remaining steps.
- An abstraction $\lambda x.a$ seems to be a value of type $t \to s$ in k steps if the application $(\lambda x.a)\ v$ seems to have type t for at most k steps. The β-reduction spends one step, hence $a[x \leftarrow v] \in E_j(s)$ for $j < k$.
- In j steps, expression $a[x \leftarrow v]$ cannot examine value v for more than j steps! Hence, it suffices that v seems to be of type t for j steps.

Adding recursive types to the logical relation

If $F : \mathtt{Type} \to \mathtt{Type}$, we write μF the algebraic type characterized by

$$\mathtt{roll} : F(\mu F) \to \mu F \qquad \mathtt{unroll} : \mu F \to F(\mu F)$$

and the reduction rule $\mathtt{unroll}(\mathtt{roll}(v)) \to v$.

It suffices to define

$$\begin{aligned} V_0(\mu F) &= \{\, \mathtt{roll}(v) \mid v \text{ value} \,\} \\ V_{k+1}(\mu F) &= \{\, \mathtt{roll}(v) \mid v \in V_k(F(\mu F)) \,\} \end{aligned}$$

The definition of $V_k(t)$ is no longer well founded by induction over t, but remains well founded by induction over k. It is obvious for type μF, and it is true as well for type $t \to s$, since the definition of $V_k(t \to s)$ uses $V_j(t)$ and $V_j(s)$ only for $j < k$.

Application: pure lambda-calculus

We can encode the pure lambda-calculus using the type $D \stackrel{def}{=} \mu(\lambda t.\, t \to t)$.

Unsurprisingly, we have

$$\begin{aligned} V_0(D) &= \{\, \texttt{roll}(v) \mid v \text{ value} \,\} \\ V_{k+1}(D) &= \{\, \texttt{roll}(\lambda x.a) \mid \forall j < k, \forall v \in V_j(D),\ a[x \leftarrow v] \in E_j(D) \,\} \end{aligned}$$

Main properties

Monotonically decreasing: $V_k(t) \subseteq V_j(t)$ and $E_k(t) \subseteq E_j(t)$ if $k \geq j$.

Compatibility with reductions:
if $a \rightarrow b$ then $a \in E_{k+1}(t)$ if and only if $b \in E_k(t)$.

Fundamental theorem:
if $x_1 : t_1, \ldots, x_n : t_n \vdash a : t$, and if $v_i \in V_k(t_i)$ for $i = 1, \ldots, n$,
then $a[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n] \in E_k(t)$

Accounting for every step

Lemma (the application case)

If $a \in E_k(t \to s)$ and $b \in E_k(t)$, then $a\, b \in E_k(s)$.

Proof.

A reduction of $a\, b$ to an irreducible term d has the shape

$$a\, b \to_n (\lambda x.\, c)\, b \to_m (\lambda x.\, c)\, v \to_1 c[x \leftarrow v] \to_p d$$

with $j = n + m + 1 + p$ reduction steps and $j \leq k$.

To conclude, we must show $d \in V_q(s)$ where $q = k - j$.

By hyp on a, $\lambda x.\, c \in V_{k-n}(t \to s)$ hence $\lambda x.\, c \in V_{p+q+1}(t \to s)$ (1).

By hyp on b, $v \in V_{k-m}(t)$ hence $v \in V_{p+q}(t)$ (2).

By (1) and (2), $c[x \leftarrow v] \in E_{p+q}(s)$ (3).

By (3), $d \in V_q(s)$, QED. $\square$

Extension to binary logical relations

$$
\begin{aligned}
V_k(\mathtt{int}) &= \{\, (n, n) \mid n \text{ integer} \,\} \\
V_k(t \to s) &= \{\, (\lambda x_1.\, a_1, \lambda x_2.\, a_2) \mid \\
&\qquad \forall j < k,\ \forall (v_1, v_2) \in V_j(t),\ (a_1[x_1 \leftarrow v_1], a_2[x_2 \leftarrow v_2]) \in E_j(s) \,\} \\
V_0(\mu F) &= \{\, (\mathtt{roll}(v_1), \mathtt{roll}(v_2)) \mid v_1, v_2 \text{ values} \,\} \\
V_{k+1}(\mu F) &= \{\, (\mathtt{roll}(v_1), \mathtt{roll}(v_2)) \mid (v_1, v_2) \in V_k(F(\mu F)) \,\} \\
E_k(t) &= \{\, (a_1, a_2) \mid \forall j \le k,\ \forall b_1,\ a_1 \to_j b_1 \wedge b_1 \text{ irreducible} \Rightarrow \\
&\qquad \exists b_2,\ a_2 \xrightarrow{*} b_2 \wedge (b_1, b_2) \in V_{k-j}(t) \,\}
\end{aligned}
$$

Note: we have $a_1 \to_j b_1$ (j steps) and $a_2 \xrightarrow{*} b_2$ (any number of steps), making it possible to relate computations a_1, a_2 of different durations.

III

A modal formulation of step-indexing

Reformulating the accounting of steps

Consider again the definition of $E_k(t)$, the set of expressions a that seem to have type t for k steps:

$$E_k(t) = \{\ a \mid \forall j \leq k,\ \forall b,\ a \rightarrow_j b \wedge b \text{ irreducible} \Rightarrow b \in V_{k-j}(t)\ \}$$

Instead of considering $j \leq k$ reduction steps ($a \rightarrow_j b$),
we can consider two cases: 0 reductions (irreducible) and 1 reduction.

- If a is irreducible, $a \in E_k(t)$ iff $a \in V_k(t)$.
- If $a \rightarrow b$, $a \in E_k(t)$ iff $b \in E_{k-1}(t)$ or $k = 0$.

We get another definition, equivalent and still well-founded by induction over k:

$$E_k(t) = \{\ a \mid (a \text{ irreducible} \Rightarrow a \in V_k(t)) \wedge (\forall b,\ a \rightarrow b \Rightarrow b \in E_{k-1}(t))\ \}$$

with, conventionally, $E_{-1}(t) =$ all the terms.

The return of the “later” modality ($\triangleright$)

Define $\triangleright E$ by $(\triangleright E)_{k+1} = E_k$ and $(\triangleright E)_0 =$ all the terms. Then:

$$E_k(t) = \{ \, a \mid (a \text{ irreducible} \Rightarrow a \in V_k(t)) \wedge (\forall b,\ a \to b \Rightarrow b \in \triangleright E_k(t)) \, \}$$

Likewise, define $(\triangleright V)_{k+1} = V_k$ and $(\triangleright V)_0 =$ all the values.
We can rewrite the two cases of the definition

$$\begin{aligned} V_0(\mu F) &= \{ \, \texttt{roll}(v) \mid v \text{ value} \, \} \\ V_{k+1}(\mu F) &= \{ \, \texttt{roll}(v) \mid v \in V_k(F(\mu F)) \, \} \end{aligned}$$

into a single “modal” case

$$V_k(\mu F) = \{ \, \texttt{roll}(v) \mid v \in \triangleright V_k(F(\mu F)) \, \}$$

The return of the “later” modality ($\triangleright$)

In the same spirit, we can rewrite the case $V_k(t \to s)$ by using $\triangleright V$. We had

$$V_k(t \to s) = \{\ \lambda x.\, a \mid \forall j < k, \forall v \in V_j(t),\ a[x \leftarrow v] \in E_j(s)\ \}$$

and we can write instead

$$V_k(t \to s) = \{\ \lambda x.\, a \mid \forall v, \forall j \leq k, v \in \triangleright V_j(t) \Rightarrow a[x \leftarrow v] \in \triangleright E_j(s)\ \}$$

This gives a quantification $\forall j \leq k$ that has the shape of implication in intuitionistic Kripke models: $k \Vdash A \Rightarrow B$ iff $\forall j \leq k,\ j \Vdash A \Rightarrow j \Vdash B$.

A modal logical relation

Finally, we can make the k parameter (the step count) implicit by using the logic of the topos of trees from the previous lecture, with its modality $\triangleright$.

$E(t)$ and $V(t)$ are, then, defined by the equations

$$
\begin{aligned}
V(\mathtt{int}) &= \{\, n \mid n \text{ integer} \,\} \\
V(t \to s) &= \{\, \lambda x.\, a \mid \forall v \in \triangleright V(t),\ a[x \leftarrow v] \in \triangleright E(s) \,\} \\
V(\mu F) &= \{\, \mathtt{roll}(v) \mid v \in \triangleright V(F(\mu F)) \,\} \\
E(t) &= \{\, a \mid (a \text{ irreducible} \Rightarrow a \in V(t)) \wedge (\forall b,\ a \to b \Rightarrow b \in \triangleright E(t)) \,\}
\end{aligned}
$$

Note that E and V are defined as functions of $\triangleright E$ and $\triangleright V$.
Löb's rule guarantees the existence of a unique fixed point for E and V.

Properties of the modality $\triangleright$

$$A \Rightarrow \triangleright A$$

$$\triangleright(A \wedge B) \text{ iff } \triangleright A \wedge \triangleright B$$

$$\triangleright(A \vee B) \text{ iff } \triangleright A \vee \triangleright B$$

$$\triangleright(A \Rightarrow B) \text{ iff } \triangleright A \Rightarrow \triangleright B$$

$$\text{if } \triangleright A \Rightarrow A \text{ then } A \qquad \text{(Löb's rule)}$$

$$\text{if } A \wedge (\triangleright A \Rightarrow \triangleright B) \Rightarrow B \text{ then } A \Rightarrow B \qquad \text{(“Löb induction”)}$$

No more accounting for every step

Lemma (the application case)

If $a \in E(t \to s)$ and $b \in E(t)$, then $a\ b \in E(s)$.

Proof.

By Löb induction. The induction hypothesis is
$a' \in \triangleright E(t \to s) \wedge b' \in \triangleright E(t) \Rightarrow a'\ b' \in \triangleright E(s)$ for all a', b'.
We argue by case whether a or b reduces.

- a and b are irreducible. Then, $a \in V(t \to s)$ and therefore a has the shape $\lambda x.c$. Also, $b \in V(t)$ is a value.

  By definition of $V(t \to s)$ and because $b \in \triangleright V(t)$, we have $c[x \leftarrow v] \in \triangleright E(s)$. Moreover, $a\ b \to c[x \leftarrow v]$. Hence $a\ b \in E(s)$.
- $a \to a'$. Then, $a' \in \triangleright E(t \to s)$ and by induction hypothesis $a'\ b \in \triangleright E(s)$. Since $a\ b \to a'\ b$, it follows that $a\ b \in E(s)$.
- a is irreducible and $b \to b'$. Similar to the previous case.

□

Counting some reductions only

We can elect to count $\mathtt{unroll}(\mathtt{roll}(v)) \to v$ reductions but not β-reductions, which amounts to using $\triangleright$ for μF types but not for $t \to s$ types.

$$
\begin{aligned}
V(\mathtt{int}) &= \{\, n \mid n \text{ integer} \,\} \\
V(t \to s) &= \{\, \lambda x.\, a \mid \forall v \in V(t),\ a[x \leftarrow v] \in E(s) \,\} \\
V(\mu F) &= \{\, \mathtt{roll}(v) \mid v \in \triangleright V(F(\mu F)) \,\} \\
E(t) &= \{\, a \mid (\forall b,\ a \xrightarrow{*}_{\beta} b \wedge b \text{ irreducible} \Rightarrow b \in V(t)) \\
&\qquad\quad \wedge (\forall b,\ a \xrightarrow{*}_{\beta} \rightarrow_{unroll} b \Rightarrow b \in \triangleright E(t)) \,\}
\end{aligned}
$$

The definition of $V(t)$ and $E(t)$ is well founded by induction on the structure of the type t then by Löb induction.

Extension to binary logical relations

Nothing surprising.

$$
\begin{aligned}
V(\texttt{int}) &= \{\, (n, n) \mid n \text{ integer} \,\} \\
V(t \to s) &= \{\, (\lambda x_1.\, a_1, \lambda x_2.\, a_2) \mid \\
&\qquad \forall (v_1, v_2) \in \triangleright V(t),\ (a_1[x_1 \leftarrow v_1], a_2[x_2 \leftarrow v_2]) \in \triangleright E(s) \,\} \\
V(\mu F) &= \{\, (\texttt{roll}(v_1), \texttt{roll}(v_2)) \mid (v_1, v_2) \in \triangleright V(F(\mu F)) \,\}
\end{aligned}
$$

$$
\begin{aligned}
E(t) = \{\, (a_1, a_2) \mid {} & (a_1 \text{ irreducible} \Rightarrow \exists b_2, a_2 \xrightarrow{*} b_2 \land (a_1, b_2) \in V(t)) \\
& \land (\forall b_1,\ a_1 \to b_1 \Rightarrow \exists b_2, a_2 \xrightarrow{*} b_2 \land (b_1, b_2) \in \triangleright E(t)) \,\}
\end{aligned}
$$

IV

Mutable state

Mutable state

It's the defining feature of imperative languages:
the ability to modify "in place" a data structure already built or a variable already defined.

Example (In-place concatenation of two lists)

```
struct list { int head; struct list * tail; }

void concat (struct list * l, struct list * m)
{
  while (l->tail != NULL) l = l->tail;
  l->tail = m;
}
```

References

A presentation of mutable state used by the ML family of languages (typed functional-imperative languages).

A reference $\approx$ a mutable indirection cell $\approx$ a 1-element array.

Example: an OCaml equivalent for C mutable lists

```
type 'a mlist = Nil | Cons of 'a ref * 'a mlist ref
```

Operations over references:

$\mathtt{ref} : t \to t\ \mathtt{ref}$	create and initialize
$! : t\ \mathtt{ref} \to t$	dereference (get current value)
$:= : t\ \mathtt{ref} \to t \to \mathtt{unit}$	assign (change the value)

Semantics of references

A simple semantics by β-reductions is wrong because it fails to account for sharing of a reference between a read and a write:

$$\texttt{let}\ r = \texttt{ref}\ 1\ \texttt{in}\ r := 2; !r \quad \neq \quad (\texttt{ref}\ 1 := 2); !(\texttt{ref}\ 1)$$

We need one level of indirection:

- references evaluate to locations ℓ ($\approx$ integers);
- a store m : location $\rightarrow_{fin}$ value records the current value of each reference;
- the operational semantics reduces configurations $\langle a,\ m\rangle$ (a term a in a store m).

Reduction rules for references

$$
\begin{aligned}
\langle (\lambda x.\, a)\; v,\; m\rangle &\to \langle a[x \leftarrow v],\; m\rangle && \text{(usual } \beta_v \text{ reduction)}\\
\langle \texttt{ref}\; v,\; m\rangle &\to \langle \ell,\; m + \{\ell \mapsto v\}\rangle && \text{if } \ell \notin Dom(m)\\
\langle !\ell,\; m\rangle &\to \langle m(\ell),\; m\rangle && \text{if } \ell \in Dom(m)\\
\langle \ell := v,\; m\rangle &\to \langle (),\; m + \{\ell \mapsto v\}\rangle && \text{if } \ell \in Dom(m)
\end{aligned}
$$

Typing the store

A store is an “heterogeneous” object: two different locations can contain values of different types.

A store typing $M : \text{location} \rightarrow_{\mathit{fin}} \text{type}$ associates a type to each location.

Initially, we take that $M(\ell)$ is a syntactic type (that is, a type expression), not a semantic type (a set of values).

Evolution of store typings

On the one hand: the type $M(\ell)$ of a valid location ℓ must remain the same throughout execution. Otherwise, we could break type safety:

$$\ell := 1 \quad \rightarrow \cdots \rightarrow \quad !\ell\ 2$$

(possible if $M(\ell) = \mathtt{int}$) (possible if $M(\ell) = \mathtt{int} \rightarrow \mathtt{int}$)

On the other hand: when we allocate a new reference at location ℓ, we must update $M(\ell)$ with the type t of its contents.

Hence an ordering between store typings: $M' \sqsupseteq M$ meaning "M can evolve into M' during execution".

$$M' \sqsupseteq M \stackrel{def}{=} Dom(M') \supseteq Dom(M) \wedge \forall \ell \in Dom(M),\ M'(\ell) = M(\ell)$$

A syntactic model of reference types

We interpret pairs (type t, store typing M) by sets of values $V(t)(M)$ or expressions $E(t)(M)$. A store typing M is interpreted by a set $[M]$ of stores.

$$
\begin{aligned}
V(\texttt{int})(M) &= \{\, n \mid n \text{ integer} \,\} \\
V(t\,\texttt{ref})(M) &= \{\, \ell \mid M(\ell) = t \,\} \\
V(t \to s)(M) &= \{\, \lambda x.\, a \mid \forall M' \sqsupseteq M,\ \forall v \in \triangleright V(t)(M'),\ a[x \leftarrow v] \in \triangleright E(s)(M') \,\} \\
[M] &= \{\, m \mid \mathit{Dom}(m) = \mathit{Dom}(M) \\
&\qquad\qquad \wedge \forall \ell \in \mathit{Dom}(m),\ m(\ell) \in V(M(\ell))(M) \,\} \\
E(t)(M) &= \{\, a \mid \forall m \in [M], \\
&\qquad\qquad (\langle a,\ m\rangle \text{ irreducible} \Rightarrow a \in V(t)(M)) \\
&\qquad\qquad \wedge (\forall b, \forall m',\ \langle a,\ m\rangle \to \langle b,\ m'\rangle \Rightarrow \\
&\qquad\qquad\qquad \exists M' \sqsupseteq M,\ m' \in [M'] \wedge b \in \triangleright E(t)(M')) \,\}
\end{aligned}
$$

A syntactic model of reference types

This typing of memory stores by syntactic types suffices to prove type soundness for references.

We would like a more “semantic” typing, where each location is associated to a set of values possibly stored at this address.

For instance, this is useful to represent invariants about the value of the reference that follow from its “encapsulation” within a function:

```
let gensym = let c = ref 0 in fun () -> c := !c + 1; !c
```

Assuming exact integer arithmetic (no overflows), we have an invariant `!c >= 0` that we would like to reflect in the model by taking $M(\ell) = \{n \mid n \geq 0\}$ where ℓ is the value of c.

A semantic model of reference types

Let's try to take $\mathit{StoreType} \stackrel{\mathit{def}}{=} \mathit{Loc} \rightarrow_{\mathit{fin}} \mathit{TypeSem}$.

Problem: a semantic type *TypeSem* is not just a set of values, it's a set of values parameterized by a store typing, as in $V(t)(M) = \{v \mid \cdots\}$.

Therefore, we run into a circularity:

$$\begin{aligned} \mathit{TypeSem} &= \mathit{StoreType} \rightarrow \mathcal{P}(\mathit{Val}) \\ \mathit{StoreType} &= \mathit{Loc} \rightarrow_{\mathit{fin}} \mathit{TypeSem} \end{aligned}$$

or, in other words,

$$\mathit{TypeSem} = (\mathit{Loc} \rightarrow_{\mathit{fin}} \mathit{TypeSem}) \rightarrow \mathcal{P}(\mathit{Val})$$

A semantic model of reference types

$$\mathit{TypeSem} = (\mathit{Loc} \rightarrow_{\mathit{fin}} \mathit{TypeSem}) \rightarrow \mathcal{P}(\mathit{Val})$$

No solutions with sets; probably a solution with domains. But, once more, step-indexing / the $\triangleright$ modality provide an easy solution!

Reading the contents of a reference ($!\ell$) consumes one step of computation.

Hence, the type *TypeSem* associated with a location ℓ can be “later” and therefore “less precise” than the *TypeSem* associated with an expression such as $!\ell$.

A semantic model of reference types

With explicit step-indexing, this leads to the family of types

$$\begin{aligned} \mathit{TypeSem}_k &= \mathit{StoreType}_k \to \mathcal{P}(\mathit{Val}) \\ \mathit{StoreType}_0 &= \mathit{Loc} \to_{\mathit{fin}} \texttt{unit} && \text{(arbitrary)} \\ \mathit{StoreType}_{k+1} &= \mathit{Loc} \to_{\mathit{fin}} \mathit{TypeSem}_k \end{aligned}$$

This is the solution to the following equation expressed in the logic of the topos of trees:

$$\mathit{TypeSem} = (\mathit{Loc} \to_{\mathit{fin}} \triangleright \mathit{TypeSem}) \to \mathcal{P}(\mathit{Val})$$

In this logic, we can do Löb inductions on all types, not just on logical propositions.

The corresponding unary logical relation

$$
\begin{aligned}
V(\texttt{int})(M) &= \{\, n \mid n \text{ integer} \,\} \\
V(t\,\texttt{ref})(M) &= \{\, \ell \mid M(\ell)(\overline{M}) \subseteq \triangleright V(t)(\overline{M}) \,\} \\
V(t \to s)(M) &= \{\, \lambda x.\, a \mid \forall M' \sqsupseteq M,\ \forall v \in \triangleright V(t)(M'),\ a[x \leftarrow v] \in \triangleright E(s)(M') \,\} \\
[M] &= \{\, m \mid \mathit{Dom}(m) = \mathit{Dom}(M) \\
&\qquad\qquad \wedge \forall \ell \in \mathit{Dom}(m), m(\ell) \in M(\ell)(\overline{M}) \,\} \\
E(t)(M) &= \{\, a \mid \forall m \in [M], \\
&\qquad\qquad (\langle a,\ m\rangle \text{ irreducible} \Rightarrow \langle a,\ m\rangle \in V(t)(M)) \\
&\qquad\qquad \wedge (\forall b, \forall m',\ \langle a,\ m\rangle \to \langle b,\ m'\rangle \Rightarrow \\
&\qquad\qquad\qquad \exists M' \sqsupseteq M,\ m' \in [M'] \wedge b \in \triangleright E(t)(M')) \,\}
\end{aligned}
$$

We write $\overline{M}$ the truncation $\texttt{next}(M)$ with $\texttt{next} : \forall A.\, A \to \triangleright A$.

In $M' \sqsupseteq M$, M' is “later” than M, hence $M' \sqsupseteq M$ is defined as $\mathit{Dom}(M') \sqsupseteq \mathit{Dom}(M)$ and $M'(\ell) = \overline{M}(\ell)$ for all $\ell \in \mathit{Dom}(M)$.

Extension to binary logical relations

This approach based on semantic store typings extends — with much effort! — to binary logical relations and to contextual equivalence propertiexs. Refer to:

- Ahmed, Dreyer, Rossberg. *State-Dependent Representation Independence*, POPL 2009.
- Dreyer, Neis, Rossberg, Birkedal. *A Relational Modal Logic for Higher-Order Stateful ADTs.* POPL 2010.

An example of use (Pitts & Stark, 1998): show that the `up` and `down` functions are contextually equivalent

```
let up   = let c = ref 0 in fun () -> c := !c + 1; !c
let down = let c = ref 0 in fun () -> c := !c - 1; - !c
```

by interpreting the locations ℓ_1, ℓ_2 of the two `c` by the relation $\{(n, -n) \mid n \geq 0\}$.

Recent developments

An ongoing rapprochement between

- Program logics for first-order imperative languages:
  Hoare logic, separation logic, concurrent separation logic.
- Logical relations for higher-order languages with mutable state.

A recent example of convergence: the Iris system, a general framework to define concurrent separation logics, which includes modalities $\triangleright$ and $\Box$ to deal with higher-order aspects.
(`https://iris-project.org/`)

V

Further reading

Further reading

The seminal paper on unary step-indexed logical relations:

- A. Appel and D. McAllester. *An indexed model of recursive types for foundational proof-carrying code.* TOPLAS 23(5), 2001.

Extension to binary relations:

- Amal Ahmed. *Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types.* ESOP 2006.

Formulations based on modal logics:

- A. Appel, P.-A. Melliès, C. Richards, J. Vouillon. *A very modal model of a modern, major, general type system.* POPL 2007.
- D. Dreyer, A. Ahmed, L. Birkedal. *Logical Step-Indexed Logical Relations.* LMCS 7, 2011.

The state of the art in program logics for imperative, concurrent, higher-order languages:

- Iris Project, *Tutorial Material*, `https://iris-project.org/tutorial-material.html`