Programming = proving?
The Curry-Howard correspondence today

Eight lecture

Step carefully:
step-indexing techniques

Xavier Leroy

Collège de France

2019-01-09
Logical relations in operational semantics
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 $t \rightarrow s$ if and only if they map arguments related at type $t$ to results related at type $s$.

Example:
the functions $\lambda n \cdot n + n$ and $\lambda n \cdot n \times 2$ are related by $R(\text{int} \rightarrow \text{int})$, assuming that $R(\text{int})$ 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 $a \rightarrow a'$.

$$ a \rightarrow a_1 \rightarrow \cdots \rightarrow a_n \rightarrow \cdots $$

- divergence
- $a \rightarrow a_1 \rightarrow \cdots \rightarrow v \not\rightarrow v \in \text{Val}$ normal termination
- $a \rightarrow a_1 \rightarrow \cdots \rightarrow a_n \not\rightarrow a_n \not\in \text{Val}$ termination on an error

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

$$ (\lambda x. a) v \rightarrow a[x \leftarrow v] \quad \text{if } v \in \text{Val} \quad (\beta_v \text{ reduction}) $$
Operational logical relations

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

\[
V(\text{int}) = \{(n, n) \mid n \text{ integer}\}
\]

\[
V(t \to s) = \{(\lambda x_1. a_1, \lambda x_2. a_2) \mid \\
\forall (v_1, v_2) \in V(t), (a_1[x_1 \leftarrow v_1], a_2[x_2 \leftarrow v_2]) \in E(s) \}
\]

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

\[
E(t) = \{(a_1, a_2) \mid \forall b_1, a_1 \xrightarrow{\ast} b_1 \land b_1 \text{ irreducible} \Rightarrow \\
\exists b_2, a_2 \xrightarrow{\ast} b_2 \land (b_1, b_2) \in V(t) \}
\]

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:

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

**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 \( t \rightarrow \text{int} \) and all integers \( n \),

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

(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 $V(t)$:

$$V(\text{int}) = \{ n \mid n \text{ integer} \}$$

$$V(t \rightarrow s) = \{ \lambda x. a \mid \forall v \in V(t), a[x \leftarrow v] \in E(s) \}$$

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

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

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. \hspace{1cm} (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 $t + s$:

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

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

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

For inductive types such as lists

\[
\text{type } \textbf{'a list} = \text{Nil} \mid \text{Cons of 'a * 'a list}
\]

we have an apparent circularity:

\[
V(t \text{ list}) = \{\text{Nil}\} \cup \{\ \text{Cons}(v, w) \mid v \in V(t) \land w \in V(t \text{ list})\}
\]

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:

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

that is,

\[
V(t \text{ list}) = \{\ \text{Cons}(v_1, \ldots, \text{Cons}(v_n, \text{Nil})) \mid v_i \in V(t)\}
\]
General recursive types

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

\[
\text{type } \text{lam} = \text{Lam of } (\text{lam} \rightarrow \text{lam})
\]

The “definition” of \(V(\text{lam})\) is obviously circular:

\[
V(\text{lam}) = \{ \text{Lam}(f) \mid f \in V(\text{lam} \rightarrow \text{lam}) \}
\]

\[
= \{ \text{Lam}(\lambda x. a) \mid \forall v \in V(\text{lam}), a[x \leftarrow v] \in E(\text{lam}) \}
\]

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 \(D_\infty \approx D_\infty \rightarrow_{cont} D_\infty\).)
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 $\texttt{int}$?

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

“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 $\texttt{int}$ 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 $\texttt{int}$.

In the end, $a$ has type $\texttt{int}$ if for all $k \in \mathbb{N}$, $a$ seems to have type $\texttt{int}$ for $k$ steps.
An indexed model of recursive types
(A. Appel and D. McAllester, TOPLAS(23), 2001)

Notation: $a \rightarrow_k b$ means “$a$ reduces to $b$ in $k$ steps”.

$V_k(\text{int}) = \{ n \mid n \text{ integer} \}$

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

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

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 $\beta$-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”.

\[
V_k(\text{int}) = \{ n \mid n \text{ integer} \}
\]

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

\[
E_k(t) = \{ a \mid \forall j \leq k, \forall b, a \to_j b \land b \text{ irreducible } \Rightarrow b \in V_{k-j}(t) \}
\]

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 $\beta$-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: $a \rightarrow^k b$ means “$a$ reduces to $b$ in $k$ steps”.

$$V_k(\text{int}) = \{ n \mid n \text{ integer} \}$$

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

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

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 $t \rightarrow s$ in $k$ steps if the application $(\lambda x. a) v$ seems to have type $t$ for at most $k$ steps. The $\beta$-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: $a \rightarrow_k b$ means “$a$ reduces to $b$ in $k$ steps”.

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

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 $t \rightarrow s$ in $k$ steps if the application $(\lambda x. a) v$ seems to have type $t$ for at most $k$ steps. The $\beta$-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 $F : \text{Type} \to \text{Type}$, we write $\mu F$ the algebraic type characterized by

$$\text{roll}: F(\mu F) \to \mu F \quad \text{unroll}: \mu F \to F(\mu F)$$

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

It suffices to define

$$V_0(\mu F) = \{ \text{roll}(v) \mid v \text{ value} \}$$
$$V_{k+1}(\mu F) = \{ \text{roll}(v) \mid v \in V_k(F(\mu F)) \}$$

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 $\mu F$, 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 $j < k$. 
We can encode the pure lambda-calculus using the type $D \overset{\text{def}}{=} \mu(\lambda t. t \rightarrow t)$.

Unsurprisingly, we have

$$V_0(D) = \{ \text{roll}(v) \mid v \text{ value} \}$$

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

$$a \ b \rightarrow_n (\lambda x. \ c) \ b \rightarrow_m (\lambda x. \ c) \ v \rightarrow_1 c[x \leftarrow v] \rightarrow_p d$$

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.
Extension to binary logical relations

\[ V_k(\text{int}) = \{ (n, n) \mid n \text{ integer} \} \]
\[ V_k(t \rightarrow s) = \{ (\lambda x_1. a_1, \lambda x_2. a_2) \mid \forall j < k, \forall (v_1, v_2) \in V_j(t), (a_1[x_1 \leftarrow v_1], a_2[x_2 \leftarrow v_2]) \in E_j(s) \} \]
\[ V_0(\mu F) = \{ (\text{roll}(v_1), \text{roll}(v_2)) \mid v_1, v_2 \text{ values} \} \]
\[ V_{k+1}(\mu F) = \{ (\text{roll}(v_1), \text{roll}(v_2)) \mid (v_1, v_2) \in V_k(F(\mu F)) \} \]
\[ E_k(t) = \{ (a_1, a_2) \mid \forall j \leq k, \forall b_1, a_1 \rightarrow_j b_1 \land b_1 \text{ irreducible} \Rightarrow \exists b_2, a_2 \rightarrow^* b_2 \land (b_1, b_2) \in V_{k-j}(t) \} \]

Note: we have \( a_1 \rightarrow_j b_1 \) (\( j \) steps) and \( a_2 \rightarrow^* b_2 \) (any number of steps), making it possible to relate computations \( a_1, a_2 \) of different durations.
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:

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

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 $k$:

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

with, conventionally, $E_{-1}(t) = \text{all the terms.}$
The return of the "later" modality ($\triangleright$)

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

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

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

$$V_0(\mu F) = \{ \text{roll}(v) \mid v \text{ value} \}$$
$$V_{k+1}(\mu F) = \{ \text{roll}(v) \mid v \in V_k(F(\mu F)) \}$$

into a single "modal" case

$$V_k(\mu F) = \{ \text{roll}(v) \mid v \in \triangleright V_k(F(\mu F)) \}$$
The return of the “later” modality ($\triangleright$)

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

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

and we can write instead

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

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

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

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 $\Diamond$

\[
A \Rightarrow \Diamond A
\]

\[\Diamond (A \land B) \text{ iff } \Diamond A \land \Diamond B\]

\[\Diamond (A \lor B) \text{ iff } \Diamond A \lor \Diamond B\]

\[\Diamond (A \Rightarrow B) \text{ iff } \Diamond A \Rightarrow \Diamond B\]

if $\Diamond A \Rightarrow A$ then $A$ \hspace{1cm} (Löb’s rule)

if $A \land (\Diamond A \Rightarrow \Diamond B) \Rightarrow B$ then $A \Rightarrow B$ \hspace{1cm} (“Löb induction”)
No more accounting for every step

### Lemma (the application case)

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

### Proof.

By Löb induction. The induction hypothesis is

\[
a' \in \triangleright E(t \rightarrow s) \land b' \in \triangleright E(t) \Rightarrow a' \ b' \in \triangleright E(s)
\]

for all \( a', b' \).

We argue by case whether \( a \) or \( b \) reduces.

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

  By definition of \( V(t \rightarrow s) \) and because \( b \in \triangleright V(t) \), we have \( c[x \leftarrow v] \in \triangleright E(s) \). Moreover, \( a \ b \rightarrow c[x \leftarrow v] \). Hence \( a \ b \in E(s) \).

- \( a \rightarrow a' \). Then, \( a' \in \triangleright E(t \rightarrow s) \) and by induction hypothesis \( a' \ b' \in \triangleright E(s) \).

  Since \( a \ b \rightarrow a' \ b' \), it follows that \( a \ b \in E(s) \).

- \( a \) is irreducible and \( b \rightarrow b' \). Similar to the previous case.
Counting some reductions only

We can elect to count \( \text{unroll}(\text{roll}(v)) \rightarrow v \) reductions but not \( \beta \)-reductions, which amounts to using \( \triangleright \) for \( \mu F \) types but not for \( t \rightarrow s \) types.

\[
V(\text{int}) = \{ n \mid n \text{ integer} \}
\]

\[
V(t \rightarrow s) = \{ \lambda x. a \mid \forall v \in V(t), \ a[x \leftarrow v] \in E(s) \}
\]

\[
V(\mu F) = \{ \text{roll}(v) \mid v \in \triangleright V(F(\mu F)) \}
\]

\[
E(t) = \{ a \mid (\forall b, \ a \xrightarrow{\beta} b \land b \text{ irreducible} \implies b \in V(t))
\land (\forall b, \ a \xrightarrow{\beta \rightarrow \text{unroll}} b \implies b \in \triangleright E(t)) \}
\]

The definition of \( V(t) \) and \( E(t) \) is well founded by induction on the structure of the type \( t \) then by L"ob induction.
Extension to binary logical relations

Nothing surprising.

\[ V(\text{int}) = \{ (n, n) \mid n \text{ integer} \} \]
\[ V(t \to s) = \{ (\lambda x_1. a_1, \lambda x_2. a_2) \mid \forall (v_1, v_2) \in \triangledown V(t), (a_1[x_1 \leftarrow v_1], a_2[x_2 \leftarrow v_2]) \in \triangledown E(s) \} \]
\[ V(\mu F) = \{ (\text{roll}(v_1), \text{roll}(v_2)) \mid (v_1, v_2) \in \triangledown V(F(\mu F)) \} \]
\[ E(t) = \{ (a_1, a_2) \mid (a_1 \text{ irreducible} \Rightarrow \exists b_2, a_2 \to^* b_2 \land (a_1, b_2) \in V(t)) \land (\forall b_1, a_1 \to b_1 \Rightarrow \exists b_2, a_2 \to^* b_2 \land (b_1, b_2) \in \triangledown E(t)) \} \]
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)

```c
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;
```

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

\[
\text{type 'a mlist} = \text{Nil} | \text{Cons of 'a ref * 'a mlist ref}
\]

Operations over references:

\[
\begin{align*}
\text{ref} : t & \rightarrow t \text{ ref} & \text{create and initialize} \\
!: t \text{ ref} & \rightarrow t & \text{dereference (get current value)} \\
:= : t \text{ ref} & \rightarrow t \rightarrow \text{unit} & \text{assign (change the value)}
\end{align*}
\]
Semantics of references

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

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

We need one level of indirection:

- references evaluate to locations $\ell$ ($\approx$ integers);
- a store $m : \text{location} \rightarrow_{\text{fin}} \text{value}$ 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

\[ \langle (\lambda x. a) \, v, \, m \rangle \rightarrow \langle a[x \leftarrow v], \, m \rangle \]  

(usual $\beta_v$ reduction)

\[ \langle \text{ref} \, v, \, m \rangle \rightarrow \langle \ell, \, m + \{ \ell \mapsto v \} \rangle \]  

if $\ell \notin \text{Dom}(m)$

\[ \langle !\ell, \, m \rangle \rightarrow \langle m(\ell), \, m \rangle \]  

if $\ell \in \text{Dom}(m)$

\[ \langle \ell := v, \, m \rangle \rightarrow \langle (), \, m + \{ \ell \mapsto v \} \rangle \]  

if $\ell \in \text{Dom}(m)$
A store is an “heterogeneous” object: two different locations can contain values of different types.

A store typing \( M : \text{location} \rightarrow_{\text{fin}} \text{type} \) 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 $\ell$ must remain the same throughout execution. Otherwise, we could break type safety:

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

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

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

Hence an ordering between store typings: $M' \sqsubseteq M$

meaning “$M$ can evolve into $M'$ during execution”.

$$M' \sqsubseteq M \overset{\text{def}}{=} \text{Dom}(M') \supseteq \text{Dom}(M) \land \forall \ell \in \text{Dom}(M), M'(\ell) = M(\ell)$$
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.

\[
V(\text{int})(M) = \{ n \mid n \text{ integer} \}
\]
\[
V(\text{t ref})(M) = \{ \ell \mid M(\ell) = t \}
\]
\[
V(t \rightarrow s)(M) = \{ \lambda x. a \mid \forall M' \supseteq M, \forall v \in \triangleright V(t)(M'), a[x \leftarrow v] \in \triangleright E(s)(M') \}
\]
\[
[M] = \{ m \mid \text{Dom}(m) = \text{Dom}(M) \land \forall \ell \in \text{Dom}(m), m(\ell) \in V(M(\ell))(M) \}
\]
\[
E(t)(M) = \{ a \mid \forall m \in [M],
\]
\[
(\langle a, m \rangle \text{ irreducible} \Rightarrow a \in V(t)(M))
\]
\[
\land (\forall b, \forall m', \langle a, m \rangle \rightarrow \langle b, m' \rangle \Rightarrow
\]
\[
\exists M' \supseteq M, m' \in [M'] \land b \in \triangleright E(t)(M')) \}
\]
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:

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

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

Let’s try to take $StoreType \overset{def}{=} Loc \rightarrow_{fin} TypeSem$.

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:

$$TypeSem = StoreType \rightarrow \mathcal{P}(Val)$$

$$StoreType = Loc \rightarrow_{fin} TypeSem$$

or, in other words,

$$TypeSem = (Loc \rightarrow_{fin} TypeSem) \rightarrow \mathcal{P}(Val)$$
A semantic model of reference types

\[ \text{TypeSem} = (\text{Loc} \rightarrow_{\text{fin}} \text{TypeSem}) \rightarrow \mathcal{P}(\text{Val}) \]

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 \( \text{TypeSem} \) associated with a location \( \ell \) can be “later” and therefore “less precise” than the \( \text{TypeSem} \) associated with an expression such as \( !\ell \).
A semantic model of reference types

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

\[ TypeSem_k = StoreType_k \rightarrow \mathcal{P}(Val) \]
\[ StoreType_0 = \text{Loc} \rightarrow_{\text{fin}} \text{unit} \] (arbitrary)
\[ StoreType_{k+1} = \text{Loc} \rightarrow_{\text{fin}} TypeSem_k \]

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

\[ TypeSem = (\text{Loc} \rightarrow_{\text{fin}} TypeSem) \rightarrow \mathcal{P}(Val) \]

In this logic, we can do L"ob inductions on all types, not just on logical propositions.
The corresponding unary logical relation

\[ V(\text{int})(M) = \{ \; n \mid n \text{ integer} \; \} \]
\[ V(t\text{ ref})(M) = \{ \; \ell \mid M(\ell)(\overline{M}) \subseteq \triangleright V(t)(\overline{M}) \; \} \]
\[ V(t \rightarrow s)(M) = \{ \; \lambda x. a \mid \forall M' \supseteq M, \forall v \in \triangleright V(t)(M'), a[x \leftarrow v] \in \triangleright E(s)(M') \; \} \]
\[ [M] = \{ \; m \mid \text{Dom}(m) = \text{Dom}(M) \]
\[ \quad \land \forall \ell \in \text{Dom}(m), m(\ell) \in M(\ell)(\overline{M}) \; \} \]
\[ E(t)(M) = \{ \; a \mid \forall m \in [M], \]
\[ \quad (\langle a, m \rangle \text{ irreducible} \Rightarrow \langle a, m \rangle \in V(t)(M)) \]
\[ \quad \land (\forall b, \forall m', \langle a, m \rangle \rightarrow \langle b, m' \rangle \Rightarrow \]
\[ \quad \exists M' \supseteq M, \; m' \in [M'] \land b \in \triangleright E(t)(M')) \; \} \]

We write \( \overline{M} \) the truncation next\((M)\) with next : \( \forall A. A \rightarrow \triangleright A \).

In \( M' \supseteq M \), \( M' \) is “later” than \( M \), hence \( M' \supseteq M \) is defined as \( \text{Dom}(M') \supseteq \text{Dom}(M) \) and \( M'(\ell) = \overline{M}(\ell) \) for all \( \ell \in \text{Dom}(M) \).
Extension to binary logical relations

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


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

\begin{verbatim}
let up = let c = ref 0 in fun () -> c := !c + 1; !c
let down = let c = ref 0 in fun () -> c := !c - 1; - !c
\end{verbatim}

by interpreting the locations \( \ell_1, \ell_2 \) 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 \( \square \) to deal with higher-order aspects.
(https://iris-project.org/)
Further reading
Further reading

The seminal paper on unary step-indexed logical relations:


Extension to binary relations:


Formulations based on modal logics:


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