Semantic Type Soundness and Language Interoperability

Amal Ahmed

June 8, 2024

1 Logical Relations

Logical Relations are a proof tactic to prove properties about programs and programming languages. Some properties of interest are:

- Program termination
- Type soundness/safety
- Program equivalence (Contextual equivalence)
- Representation Independence (Interface and existential type)
- Parametricity
- Non-interference in security-typed programming languages

These relations are said to be *unary* when reasoning about a property of a single program, as in the case of termination or type soundness, or *binary*, when reasoning about the relation between two programs. Binary relations can apply to programs written in two different languages, such as a source language and a target language.

2 Simply Typed Lambda Calculus

The language used in these lectures is the Simply Typed Lambda Calculus, with type annotations *à la Church* and extended with boolean and if-then-else expressions.

2.1 Formal Definition

The formal definition of the language follows, where the meta-variables τ, e, v, E and Γ represent respectively types, expressions, values, evaluation contexts and typing contexts.

$$
\begin{aligned}
\tau &::= \texttt{bool} \mid \tau_1 \to \tau_2 \\
e &::= x \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \lambda x : \tau.\ e \mid e_1\ e_2 \\
v &::= \texttt{true} \mid \texttt{false} \mid \lambda x : \tau.\ e \\
E &::= [\,\cdot\,] \mid \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid E\ e \mid v\ E \\
\Gamma &::= \cdot \mid \Gamma,\ x : \tau
\end{aligned}
$$

2.2 Operational Semantics

$\boxed{e \mapsto e'}$

$$\texttt{if true then } e_1 \texttt{ else } e_2 \mapsto e_2 \qquad \texttt{if false then } e_1 \texttt{ else } e_2 \mapsto e_2 \qquad (\lambda x : \tau .\ e)\, v \mapsto e[v/x]$$

$$\frac{e \mapsto e'}{E[e] \mapsto E[e']}$$

2.3 Typing Judgements

$\boxed{\Gamma \vdash e : \tau}$

$$\Gamma \vdash \texttt{true} : \texttt{bool} \qquad \Gamma \vdash \texttt{false} : \texttt{bool} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 .\ e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau} \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : \tau}$$

3 Type Soundness

The theorem of type soundness, introduced by Milner, states that if a term is well-typed then it is guaranteed to have well-defined behaviour when evaluated.

Theorem 3.1 (Type Soundness)**.**

$$\cdot \vdash e : \tau \implies \forall e' \, [(e \mapsto^* e') \to (\text{val}(e') \vee [\, \exists e''.\, e' \mapsto e''])]$$

In English: "If a well-typed term e evaluates to e', the e' is either a value, or can be evaluated further to e'' "

3.1 Syntactic Type Soundness

Type soundness is most commonly proved *syntactically*, by proving *progress* and *preservation.* This technique was introduced by Wright and Felleisen.

Lemma 3.2 (Progress)**.** If $\cdot \vdash e : \tau$, then $\text{val}(e) \vee \exists\, e' .\, e \mapsto e'$

Lemma 3.3 (Preservation)**.** If $\cdot \vdash e : \tau$ and $e \mapsto e'$, then $\cdot \vdash e' : \tau$

Theorem 3.4. Progress & Preservation $\implies$ Type-Safety

Proof. Suppose $\cdot \vdash e : \tau$, $e \mapsto^* e'$ be arbitrary. Then $\cdot \vdash e' : \tau$ by preservation (and induction). Hence $\text{val}(e') \vee \exists\, e' .\, e \mapsto e'$ by progression. Thus we get type-safety □

Remark. Recall what we want to achieve is type soundness. Proving progress and preservation is one way of achieving this goal. However, proving progress and preservation may be more than necessary. In the lecture, we commented on Rust. We may write unsafe Rust in an unsafe block, which during execution may temporarily enter a state that is considered unsafe or not well-typed. However, when the entire unsafe block is executed, the program returns to a safe state.

4 Semantic Type Soundness

While syntactic type soundness is the most used technique, Milner's original approach used denotational semantics. While the syntactic approach characterizes soundness by syntactic well-typedness, the *semantic* approach characterizes soundness by the behaviour of programs. One of the advantages of the semantic approach is in proving the safety of programs with *unsafe* behaviour, such as in Rust or Haskell, where the type system is too restrictive to allow programs that behave correctly.

Definition 4.1 (Safety)**.**

$$\texttt{safe}(e) \overset{\text{def}}{=} \forall e' \left[\left(e \mapsto^* e'\right) \implies \left(\text{val}(e') \vee \left[\exists\, e''\,.\, e' \mapsto e''\right]\right)\right]$$

Below, we define what it means for a value and an expression to be semantically well-behaved in the STLC. Values are well-behaved under type τ if they are a member of $\mathcal{V}[\![\tau]\!]$. The same follows for expressions and $\mathcal{E}[\![\tau]\!]$.

$$\begin{aligned}
\mathcal{V}[\![\texttt{bool}]\!] &= \{\texttt{true}, \texttt{false}\}\\
\mathcal{V}[\![\tau_1 \to \tau_2]\!] &= \{\lambda x : \tau_1.\ e \mid \forall v \in \mathcal{V}[\![\tau_1]\!].\ e[v/x] \in \mathcal{E}[\![\tau_2]\!]\}\\
\mathcal{E}[\![\tau]\!] &= \{e \mid \forall e'.\ e \mapsto^* e' \wedge \texttt{irred}(e') \implies e' \in \mathcal{V}[\![\tau]\!]\}
\end{aligned}$$

where

$$\textbf{irred}(e) = \nexists e'.\ e \mapsto e'$$

Note that this definition of $\mathcal{E}[\![\tau]\!]$ is based on the fact that evaluation of expressions in the STLC always terminates. For extensions of STLC, particularly those in which this guarantee is lost, a new definition may be necessary. Furthermore, note that for $v \in \mathcal{V}[\![\tau]\!]$ to hold, we do not need to produce a typing derivation for $v : \tau$. Instead, we are focused on asking if during execution, the value "behaves" as a member of the claimed type; in particulary if it is "semanticallly well-behaved". For the STLC, there is little distinction between a value being well-typed and it being semantically well-behaved (under the same type τ). However, this has practical usage in languages such as Rust, where an unsafe segment of code which would not ordinarily typecheck is assumed to be overall semantically well-behaved and satisfy whatever guarantees are expected.

Definition 4.2 (Substitution Function)**.**

$$\begin{aligned}
\mathcal{G}[\![\cdot]\!] &= \emptyset\\
\mathcal{G}[\![\Gamma, x : \tau]\!] &= \{\gamma[v/x] \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \wedge v \in \mathcal{V}[\![\tau]\!]\}
\end{aligned}$$

where $\emptyset$ denotes the function with $\emptyset$ as domain. $\gamma[v/x]$ denotes function γ extended with x maps to v

Definition 4.3 (Semantically Well-Typed)**.**

$$\Gamma \vDash e : \tau \iff \forall\, \gamma \in \mathcal{G}[\![\Gamma]\!]\,.\, \gamma(e) \in \mathcal{E}[\![\tau]\!]$$

We may understand that a term is semantically well-typed if given any substitution function, the function substitutes to produce a well-behaved term. The definition of well-behaved depends on $\mathcal{E}[\![\tau]\!]$.

Theorem 4.1 (Semantic Type Soundness)**.** If $\cdot \vdash e : \tau$ then $\texttt{safe}(e)$

Proof. The proof comes in two parts

A. $\cdot \vdash e : \tau \implies e \in \mathcal{E}[\![\tau]\!]$

Proof. Here we assume the Fundamental Property of Logical Relations.
Suppose $\cdot \vdash e : \tau$. Then we get $\cdot \vDash e : \tau$.
By definition of $\vDash$, we have $\forall\, \gamma \in \mathcal{G}[\![\cdot]\!]\,.\, \gamma(e) \in \mathcal{E}[\![\tau]\!]$.
But this means $e \in \mathcal{E}[\![\tau]\!]$ (because a substitution function for the empty context does not modify e) ■

B. $e \in \mathcal{E}[\![\tau]\!] \implies \texttt{safe}(e)$

Proof. Suppose $e \in \mathcal{E}[\![\tau]\!]$.
Let $e \mapsto^* e'$ be arbitrary.
If $\neg\,\textbf{irred}(e')$, then it means $\exists\, e''\,.\, e' \mapsto e''$, hence $\left(\text{val}(e') \vee \left[\exists\, e''\,.\, e' \mapsto e''\right]\right)$
If $\textbf{irred}(e')$, then notice by definition of $\mathcal{E}[\![\tau]\!]$, we get $e' \in \mathcal{V}[\![\tau]\!]$. Examining the elements of $\mathcal{V}[\![\tau]\!]$ shows that e' must be a value, hence $\text{val}(e')$, and $\left(\text{val}(e') \vee \left[\exists\, e''\,.\, e' \mapsto e''\right]\right)$
Thus $\texttt{safe}(e)$. ■

Thus we get $\cdot \vdash e : \tau \implies \texttt{safe}(e)$ □

Theorem 4.2 (Fundamental Property of Logical Relations)**.** If $\Gamma \vdash e : \tau$ then $\Gamma \vDash e : \tau$

Proof. Proof by induction on typing derivation.

Case 4.1. $\Gamma \vdash \texttt{true}$
Want to show: $\Gamma \vDash \texttt{true} : \texttt{bool}$
Suppose that $\gamma \in \mathcal{G}$,
We only need to show that $\gamma(\texttt{true}) \in \mathcal{E}([\![\texttt{bool}]\!])$
then it suffices to show that
$\texttt{true} \in V[\![\texttt{bool}]\!]$

Case 4.2. $\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$
Want to show: $\Gamma \vDash x : \tau$
Suppose $\gamma \in \mathcal{G}[\![\Gamma]\!]$
then it suffices to show that
$v = \gamma(x) \in \mathcal{E}[\![\tau]\!]$ which is immediate from the definition of $\mathcal{E}[\![\tau]\!]$

Case 4.3. λ:
$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash x : \tau_1 . e : \tau_1 - > \tau_2}$$
Need to show
$\gamma(\lambda x : \tau_1 . e) \in \mathcal{E}[\![\tau_1 \to \tau 2]\!]$
which is equivalent to

$\lambda x.\tau_1.\gamma(e) \in \mathcal{E}[\![\tau_1 \to \tau_2]\!]$
From the definition of entailment and $\mathcal{E}$ it suffices to show that:
$\lambda x : \tau_1.\gamma(e) \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$
Suppose that $v \in \mathcal{V}[\![\tau_1]\!]$.
As v is arbitrary, it suffices to show $\gamma(e)[v/x] \in \mathcal{E}[\![\tau_2]\!]$
which is equivalent to
$\gamma[v/x](e) \in \mathcal{E}[\![\tau_2]\!]$

By our inductive hypothesis (IH) we have that $\Gamma, x.\tau_1 \vDash e : \tau_2$
in the definition of entailment IH let us instantiate $\forall\gamma$ with $\gamma, [v/x]$,
which is an element of $\mathcal{G}[\![\gamma, x : \tau_1]\!]$ by our assumptions.
The gives us $\gamma[v/x](e) \in \mathcal{E}[\![\tau_2]\!]$. The desired result.

□

5 Recursive Types

In order to express the types of recursive data structures such as trees and lists, we must add sufficient typing rules to the STLC. Intuitively, in an ML-like language, we could express the type of a binary tree with values on the nodes as:

$$\texttt{tree} = 1 + (\texttt{int} * \texttt{tree} * \texttt{tree})$$

We transform this type definition into a function F which, given a type α, constructs anew type resembling the `tree` type above.

$$F = \lambda\alpha : \texttt{type}.\ 1 + (\texttt{int} * \alpha * \alpha)$$

Let $\mu\alpha.\ \tau$ be a least fixpoint constructor (recall that a fixpoint x of a function f is a value such that $x = f(x)$). If we apply $\mu\alpha.\ \tau$ to F from above, we find the following:

$$\begin{aligned} \mu\alpha.\ F(\alpha) &= F(\mu\alpha.\ F(\alpha)) && \text{By definition} \\ \mu\alpha.\ \tau &= F(\mu\alpha.\ \tau) && \text{Let } \tau = F(\alpha) \\ \mu\alpha.\ \tau &= \tau[\mu\alpha.\ \tau/\alpha] \end{aligned}$$

Note this final equivalence. To replace $\mu\alpha.\ \tau$ with $\tau[\mu\alpha.\ \tau/\alpha]$ is to unfold the type, and to go in the reverse direction is to fold it. We formalize this in an extension of the STLC below.

5.1 Language

The syntax of the STLC presented earlier extended with recursive types and folding operations.

$$\begin{aligned} \tau &::= \cdots \mid \mu\alpha.\tau \\ e &::= \cdots \mid \texttt{fold}\ e \mid \texttt{unfold}\ e \\ v &::= \cdots \mid \texttt{fold}\ v \\ E &::= \cdots \mid \texttt{fold}\ E \end{aligned}$$

5.2 Typing Rules

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \texttt{fold}\ e : \mu\alpha.\tau} \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \texttt{unfold}\ e : \tau[\mu\alpha.\tau\alpha]}$$

5.2.1 Nontermination

As an aside, note that we now support nontermination in our programs as we can typecheck terms whose evaluation does not terminate.

Let SA be the self-application term $\lambda x : \mu\alpha.\ \alpha \to \tau.\ (\texttt{unfold}\ x)\ x$. Then $\texttt{unfold}\ x$ has the type $(\mu\alpha.\ \alpha \to \tau) \to \tau$. Therefore, SA produces a τ and thus has the type $(\mu\alpha.\ \alpha \to \tau) \to \tau$.

Now we construct the term $\Omega = SA\ (\texttt{fold}\ SA)$. From the type of SA, we see that the term $(\texttt{fold}\ SA)$ has the type $\mu\alpha.\ \alpha \to \tau$. Therefore, Ω has the type τ. We have successfully typechecked a term which causes nonterminating evaluation!

5.3 Denotation

$$\mathcal{V}[\![\mu\alpha.\tau]\!] = \{\texttt{fold}\ v \mid v \in \mathcal{V}[\![\tau[\alpha \to \mu\alpha.\tau]]\!]\}$$

5.4 Operational Semantics

$\boxed{e \mapsto e'}$

$$\texttt{unfold}\ (\texttt{fold}\ v) \mapsto v$$

6 Step Index

6.1 Motivation

Here we will build up the motivation for the technique known as Step Index.

Now we have added recursive types to our language, we must also define the set of well-behaved $\mu\alpha.\tau$ terms. Let us consider the following naive approach:

$$\mathcal{V}[\![\mu\alpha.\tau]\!] = \{\texttt{fold}\ v \mid \texttt{unfold}\ (\texttt{fold}\ v) \in \mathcal{V}[\![\tau[\alpha \to \mu\alpha.\tau]]\!]\}$$

The approach is analogous to the set for $\tau_1 \to \tau_2$, where $e[v/x]$ is the same as $(\lambda x.e)v$, which is the eliminator of function types. Similarly here, $\texttt{unfold}$ is the eliminator of recursive types.

However, this approach is flawed. Consider the type $\tau[\alpha \to \mu\alpha.\tau]$ on the right-hand side, which almost certainly is a larger type than the $\mu\alpha.\tau$ on the left-hand side. We will not be able to use the induction hypothesis on a larger type, thus previous approach will fail.

6.2 Solution: Step Index

We can solve the problem by introducing the concept of step. Intuitively, we want "terms that behave well for k steps", for "arbitrarily large k".

6.3 Denotation

$$\begin{aligned}
\mathcal{V}_k[\![\texttt{bool}]\!] &= \{\texttt{true},\ \texttt{false}\}\\
\mathcal{V}_k[\![\tau_1 \to \tau_2]\!] &= \{\lambda x : \tau_1.e \mid \forall j \leq k,\ \forall v.\ v \in \mathcal{V}_j[\![\tau_1]\!] \implies e[v/x] \in \mathcal{E}_j[\![\tau_2]\!]\}\\
\mathcal{V}_k[\![\mu\alpha.\tau]\!] &= \{\texttt{fold}\ v \mid \forall j < k.\ v \in V_j[\![\tau[\mu\alpha.\tau/\alpha]]\!]\} \quad (\text{note } \texttt{unfold}\ (\texttt{fold}\ v) \mapsto v)\\
\mathcal{E}[\![\tau]\!] &= \{e \mid \forall j < k, \forall e'.e \mapsto^j e' \wedge \texttt{irred}(e') \implies e' \in \mathcal{V}_{k-j}[\![\tau]\!]\}\\
\mathcal{G}_k[\![\Gamma, x : \tau]\!] &= \{\gamma[v \mapsto x] \mid \gamma \in \mathcal{G}_k[\![\Gamma]\!] \wedge v \in \mathcal{V}_k[\![\tau]\!]\}
\end{aligned}$$

6.4 Type Soundness

Theorem 6.1 (Step-indexed Semantic Type Soundness)**.**

$$\Gamma \vDash e : \tau \overset{\mathsf{def}}{=} \forall k \geq 0.\ \forall \gamma \in \mathcal{G}_k[\![\Gamma_k]\!].\ \gamma(e) \in \mathcal{E}_k[\![\tau]\!]$$

Proof. Proof of type soundness by induction on the rules of typing derivation.

Note: We only provide the proof for the fold type judgement, the remaining derivations are left to the reader.

Proof. Fold Recall: type judgement of fold

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \texttt{fold}\ e : \mu\alpha.\tau}$$

We will need to show that $\Gamma \vDash fold\ e : \mu\alpha.\tau$

- Assumption: $k \geq 0$
- Assumption: $\gamma \in \mathcal{G}[\![\Gamma]\!]$

To satisfy the definition of entailment we must now show that

$\gamma(fold\ e) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$
which is equivalent to
$fold\ \gamma(e) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$

- Assumption: for some j ¡ k $\gamma(e) \mapsto e' \wedge irreducible(e)$

Note that our language is deterministic so the following must be true:

- $fold\gamma(e) \mapsto_{j\leq k} (e') \wedge irreducible(e) \implies \gamma(e) \mapsto_{j_1 \leq j} (e') \wedge irreducible(e)$

notice that we are focusing on the e rather that fold e.
also note that we are focusing upon j_1 and k

now we call upon our inductive hypothesis:

$\Gamma \vDash e : \tau \overset{\mathsf{def}}{=} \forall k' \geq 0.\ \forall \gamma' \in \mathcal{G}[\![\Gamma]\!].\ \gamma'(e) \in \mathcal{E}_k'[\![\tau]\!]$
Let us specialize γ' with γ and k' with k

By the definition of $\mathcal{E}$ and by our previous assumptions
e must evaluate to some value e" s.t. irreducible(e") and $\gamma(e'') \in \mathcal{V}[\![\mu\alpha.\tau/\tau]\!]$

We know that e_1 satisfies this predicate, so, by the definition
of operational semantics j = j_1 and e' = e_1

therefore $fold\ e \mapsto_{k-j} fold\ e'$

therefore $fold\ e' \in \mathcal{V}_{k-j}[\![fold]\!]$ by definition of $\mathcal{V}$
and from monotonicity we also have that $\forall j' \leq k - j. fold\ e' \in \mathcal{V}_{j'}[\![fold]\!]$

Which satisfies our definition of entailment.

□

□

In the proof of the previous theorem, the proof for the case of recursive types requires the following monotonicity theorem, which states that values which are semantically well-behaved for k steps also behave as such for fewer steps.

Theorem 6.2 (Monotonicity or Downward Closure)**.**

$$v \in V_k[\![\tau]\!] \implies \forall j \leq k. v \in V_j[\![\tau]\!]$$

7 Mutable References

We now consider an extension of the STLC with mutable references but *without* recursive types.

7.1 Language

$$\begin{aligned}
\tau &::= 1 \mid \texttt{bool} \mid \tau_1 \to \tau_2 \mid \texttt{ref}\ \tau \\
e &::= \cdots \mid () \mid \texttt{new}\ e \mid !e \mid e_1 := e_2 \\
v &::= \cdots \mid l \\
E &::= \cdots \mid \texttt{new}\ E \mid !E \mid E := e_2 \mid v := E
\end{aligned}$$

7.2 Operational Semantics

Program configurations are represented as pairs (S, e) where S is a store (also called a heap) and e is the current expression being evaluated. Stores map locations to closed values.

$$\begin{aligned}
&S : \text{Loc} \to \text{CVal} \\
&S = \{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\}
\end{aligned}$$

$$\frac{l \notin \text{dom}(S)}{(S, \texttt{new}\ v) \mapsto (S[l \mapsto v], l)} \qquad \frac{l \mapsto v \in S}{(S, !l) \mapsto (S, v)} \qquad \frac{l \in \text{dom}(S)}{(S, l := v) \mapsto (S[l \mapsto v], ())}$$

7.3 Typing Rules

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{new}\ e : \texttt{ref}\ \tau} \qquad \frac{\Gamma \vdash e : \texttt{ref}\ \tau}{\Gamma \vdash\ !e : \tau} \qquad \frac{\Gamma \vdash e_1 : \texttt{ref}\ \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \coloneqq e_2 : 1}$$

7.4 Aliasing and Types

Aliasing can be an issue for many languages that implement mutable references. One danger comes from "use after freed" in languages like C. Here we present another source of error.

Consider the following setup, where x, y are references that both point to the same location l:

$$\begin{tikzcd} x \arrow[dr] & & y \arrow[dl] \\ & l & \end{tikzcd}$$

Now consider the following code:

$$\begin{array}{l} x := 5; \\ \texttt{if}\ (!\,y)\ \texttt{then}\ \ldots\ \texttt{else}\ \ldots \end{array}$$

This is a dangerous piece of code because when de-referencing y, as we will get stuck when evaluating "`if 5` …". To avoid this issue, we will enforce the constraint "once a reference is created with a value of type τ, it will always store a value of type τ". We already see this in the typing rules, where an assignment expression has a type only when the value type matches the reference type. We will see later how to express this in logical relations.

7.5 Non-termination

As an aside, note that this extension of the STLC supports recursion and nontermination! We utilize the following technique called Landin's Knot.

$$\begin{array}{l} \textit{let}\ x = \texttt{new}\ \lambda z : 1.\ z \\ \textit{let}\ f = \lambda z : 1.\ (!x)\ z \\ x \coloneqq f; \\ f\ () \end{array}$$

7.6 Storing Typing Information

The denotational semantics are extended do deal with mutable references as follows. This requires the use of a new context Ψ which maps locations to *semantic types*. This will be useful when we express "the type of stored expression will not change".

$$\Psi = \{l_1 \mapsto \mathcal{V}[\![\tau_1]\!], \ldots, l_n \mapsto \mathcal{V}[\![\tau_n]\!]\}$$

7.7 Denotation

Here we modify definitions of $\mathcal{V}$, $\mathcal{E}$, and $\mathcal{G}$ to properly support our new store typing.

$$\begin{aligned}
\mathcal{V}_{k,\Psi}[\![\texttt{ref }\tau]\!] &= \{l \mid \Psi(l) = \lfloor \mathcal{V}[\![\tau]\!] \rfloor_{k-1}\} \\
\mathcal{V}_{k,\Psi}[\![\tau_1 \to \tau_2]\!] &= \{\lambda x : \tau_1.e \mid \forall (j, \Psi') \sqsupseteq (k, \Psi). \forall v \in \mathcal{V}_{j,\Psi}[\![\tau_1]\!] \implies e[v/x] \in \mathcal{E}_{j,\Psi'}[\![\tau_2]\!]\} \\
\mathcal{E}_{k,\Psi}[\![\tau]\!] &= \{e \mid \forall S, S', e', j < k.\ S_k : \Psi \wedge (S, E) \mapsto^j (S', e') \wedge \mathrm{irred}(S', e') \\
&\qquad \implies \exists \Psi'.\ (k-j, \Psi') \sqsupseteq (k, \Psi) \wedge S' :_{k-j} \Psi' \wedge e' \in \mathcal{V}_{k-j,\Psi}[\![\tau]\!]\} \\
\mathcal{G}_{k,\Psi}[\![\Gamma,\ x : \tau]\!] &= \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}_{k,\Psi}[\![\Gamma]\!] \wedge v \in \mathcal{V}_{k,\Psi}[\![\tau]\!]\}
\end{aligned}$$

The new semantics definitions are designed such that the store should only be able to update a location with a value of the same type, for all future steps. But designing the semantics naively introduces inconsistencies in our logic. Thus we will need to also introduce new Meta-Type definitions, and a new concept 'worlds'. Reading about the new Meta-Types and worlds will give meaning to the new notation in these semantics.

7.8 Type Hierarchy

As we have extended the STLC, the structure of our semantic types have changed as below.

$$\begin{aligned}
\text{Type} &= \{I \in \mathcal{P}(\text{CVal})\} && \text{Base STLC} \\
\text{Type} &= \{I \in \mathbb{N} \to \mathcal{P}(\text{CVal}) \mid \text{monotonicity}(I)\} && + \text{Recursive Types}
\end{aligned}$$

We might expect that we can extend this similarly to work with our new Ψ context:

$$\begin{aligned}
\text{Type} &= \{I \in \mathbb{N} \times \text{StoreTy} \to \mathcal{P}(\text{CVal}) \mid \text{monotonicity}(I)\} && + \text{Mutable References} \\
\text{StoreTy} &= \text{Loc} \xrightarrow{\text{fin}} \text{Type}
\end{aligned}$$

Unfortunately, this actually produces an inconsistency. Because StoreTy appears in the negative of Type, $|\text{Type}| > \text{StoreTy}$. Because Type appears in the negative of StoreTy, $|\text{StoreTy}| \geq \text{Type}$. These inequalities cannot hold together, so we need a less naive type structure. We will build one up hierarchically so as to stratify the types.

$$\begin{aligned}
\text{Type}_0 &= \emptyset \\
\text{Type}_{k+1} &= \mathbb{N} \times \text{StoreTy}_k \to \mathcal{P}(\text{CVal}) \\
\text{StoreTy}_k &= \text{Loc} \xrightarrow{\text{fin}} \text{Type}_k \\
\text{Type} &= \bigcup_{k \in \mathbb{N}} \text{Type}_k
\end{aligned}$$

7.9 Worlds

We can also express our types in terms of the concept of worlds. Logic that considers these worlds is also called Kripke Models or accessibility models. We define this system as below:

$$\begin{aligned}
\text{World}_n &= \{(k, \Psi) \mid k < n \wedge \Psi \in \text{StoreTy}_k\} \\
\text{StoreTy}_n &= \{\Psi \in \text{Loc} \xrightarrow{\text{fin}} \text{Type}_n\} \\
\text{Type}_n &= \{I \in \text{World}_n \to \mathcal{P}(\text{CVal}) \mid \text{monotonicity}(I)\}
\end{aligned}$$

Here we also define some convenient notations:

$$\lfloor \mathcal{V}[\![\tau]\!] \rfloor_k \overset{\text{def}}{=} \mathcal{V}_{k,\Psi}[\![\tau]\!]$$

$$\lfloor \Psi \rfloor_k \overset{\text{def}}{=} \{l \mapsto \lfloor \mathcal{V}[\![\tau]\!] \rfloor_k\} \text{ where } \Psi(l) = \mathcal{V}[\![\tau]\!]$$

7.10 Ordering on Worlds

While discussing recursive types, we introduced step indexing. The point was to consider how terms behave as time progresses. We are now replacing time with worlds. Thus we need to define order on worlds.

$$(j, \Psi') \sqsupseteq (k, \Psi) \overset{\text{def}}{=} j \le k \wedge \forall l \in \text{dom}(\Psi).\ \lfloor \Psi(l) \rfloor_{j-1} = \lfloor \Psi'(l) \rfloor_{j-1}$$

Here we first require $j < k$, representing fewer steps remaining. There is no free operation, so the new reference table must have at least as many locations, with each location storing closed a value of the same type as previous world. This relation is called the accessibility relation.

7.11 Storage Satisfying Store Type

Previously, when discussing whether a value is well-behaved, we considered eliminating it with all possible well-behaved values. We will use a similar argument for mutable reference. Previously, whether a term is well-behaved depends on the number of steps remaining. Now we need to take storage into account.

We define the following notation:

$$S :_k \Psi \overset{\text{def}}{=} \text{dom}(\Psi) \subseteq \text{dom}(S) \wedge \forall j < k, l \in \text{dom}(\Psi).\ S(l) \in \lfloor \Psi(l) \rfloor_j$$

The left-hand side reads "S is satisfied by Ψ". We say S satisfies a store type Ψ when

1. It has a value for each location in $\text{dom}(\Psi)$
2. Each value is well-behaved for the type in Ψ at j steps

7.12 Soundness

Lemma 7.1 (Semantic Type Soundness)**.**

$$\Gamma \vDash e : \tau \overset{\text{def}}{=} \forall k \in \mathbb{N}, \Psi, \gamma.\ (k, \Psi) \in \text{World}_{k+1} \wedge \gamma \in \mathcal{G}_{k,\Psi}[\![\Gamma]\!] \implies \gamma(e) \in \mathcal{E}_{k,\Psi}[\![\tau]\!]$$

Lemma 7.2 (Monotonicity)**.**

$$v \in \mathcal{V}_{k,\Psi}[\![\tau]\!] \wedge (j, \Psi) \sqsupseteq (k, \Psi) \implies v \in \mathcal{V}_{j,\Psi'}[\![\tau]\!]$$

8 Binary Logical Relation & Language Interoperability

So far the logical relations presented were unary, but binary relations are useful to prove properties relating two programs of the same language, or even between programs of different languages. A binary logical relation states that two programs behave in the same manner.

The general form of binary logical relationship is as follows:

$$\mathcal{V}[\![\tau]\!] = \{(v_1, v_2) \mid \dots\}$$
$$\mathcal{E}[\![\tau]\!] = \{(e_1, e_2) \mid \dots\}$$

8.1 Example Usages

Now that we have binary logical relations, we can express ideas that involve two terms. For example, we can express the concept of term equivalence as follows:

$$\Gamma \vdash e_1 \approx e_2 : \tau = \forall(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!] \implies (\gamma_1(e_1), \gamma_2(e_2)) \in \mathcal{E}[\![\tau]\!]$$

The idea is that two terms are equivalent if when substituted with equivalence values, the resulting terms are still equivalent. Of course, we did not define $\mathcal{G}[\![\Gamma]\!]$, but the definition should be similar to what we have seen previously.

Of course, we can still encounter issue with induction like we had for unary relationships. In which case we may appeal to step-index again. Here is a possible usage:

$$\mathcal{E}_k[\![\tau]\!] = \{(e_1, e_2) \mid \forall e'.\forall j < k.e_1 \mapsto^j v_1 \implies \exists v_2.e_2 \mapsto^* v_2 \land (v1, v2) \in \mathcal{V}_{k-j}[\![\tau]\!]\}$$

We see the usage of k and j. The exact definition depends on the case. One trick to note is we don't need to index the number of reduction on e_1 and e_2 simultaneously. In fact, it may be too strong, since the it may take e_2 more than one reduction to achieve the same effect as one reduction on e_1.

Binary logical relationships can be used to prove the equivalence of terms from two different languages. Consider compiling a source language with a boolean to a target language with only integers. During compilation, we may choose 0 to represent `true`. Thus giving us the following relationship:

$$\mathcal{V}_{ST}[\![\texttt{bool}]\!] = \{(true, 0), (false, n \neq 0)\}$$

The above relationship states `true` is related to 0, while `false` is related to any non-zero number. Our choice of $\mathcal{V}_{ST}[\![\texttt{bool}]\!]$ will affect what we need to prove.

Similarly, a function in a source and target language may appear very differently. For example, in the target language, a lambda term may be represented by a piece of code, and the current environment. We can still express the relationship as follows:

$$\mathcal{V}_{ST}[\![\texttt{int} \to \texttt{int}]\!] = \{(\lambda x : \texttt{int}.e, (\text{code}, \text{env})) \mid (v_s, v_t) \in \mathcal{V}[\![\texttt{int}]\!] \implies \text{code}(\text{env}, v_t) \in \mathcal{E}[\![\texttt{int}]\!]\}$$

One application of binary logical relations is to prove the type soundness of programming languages that implement foreign function interfaces. A FFI provides an escape hatch to run code that may not be type safe.

8.2 Language Interop and Semantic Intermediate Representation

Following paper by Matthew-Findler, a method of proving correctness of interop between languages had been developed

1. Add boundary forms
2. Specify convertibility rules ($\tau_A \sim \tau_B$) between types of each language
3. Implement target-level conversions for each pair of convertible types, denoted $C_{\tau_B \mapsto \tau_A}(e), C_{\tau_A \mapsto \tau_B}(e)$
4. Define semantic IR (Which is a logical relationship):

$$\llbracket \tau_A \rrbracket = \{e \mid \cdots\}$$
$$\llbracket \tau_B \rrbracket = \{e \mid \cdots\}$$

5. Prove the conversion soundness: If $\tau_A \sim \tau_B$, then

$$e \in \llbracket \tau_B \rrbracket \implies C_{\tau_B \mapsto \tau_A}(e) \in \llbracket \tau_A \rrbracket$$
$$e \in \llbracket \tau_A \rrbracket \implies C_{\tau_A \mapsto \tau_B}(e) \in \llbracket \tau_B \rrbracket$$