Logical relations and references

Edited by:
Lau Skorstengaard
lask@cs.au.dk
April 17, 2016

This note is based on lectures by Lars Birkedal and work written by Aleš Bizjak and Kasper Svendsen among others.

In Section 1 we sketch a unary model for $F_{\mu,\text{ref}}$. The main point of presenting a unary model is to motivate and explain the challenges that appear when we consider a language with references (Subsection 1.4 on c.o.f.e.’s is precise, the remaining remaining subsections are somewhat sketched). In Section 2 we present a relational model which we do precisely.

1 Unary model

1.1 Syntax and operational semantics

We extend $F_{\mu}$ to $F_{\mu,\text{ref}}$ in the following way:

\[

tau ::= \ldots | \text{ref } \tau \\
e ::= \ldots | \text{ref } e | e := e | !e \\
E ::= \ldots | \text{ref } E | E := e | v := E | !E \\
v ::= \ldots | l
\]

We now want to define the semantics such that \text{ref } e allocates a new location on the heap, $e_1 := e_2$ assigns or writes the value $e_2$ evaluates to to the location $e_1$ evaluates to, and $! e$ dereferences or reads the location $e$ evaluates to. To do this we need to define what a heap, $h$, is.

\[
b : \text{Loc} \xrightarrow{\text{fin}} \text{Val}
\]

A heap is a finite, partial function from locations to values. This seems to be a sensible abstraction of a real heap as any program will only allocate a finite amount of memory cells. At the same time, we have an infinite amount of memory cells available for allocation, so we can always allocate new memory.

We define a configuration to be a pair consisting of a heap and an expression, $(h, e)$. We define the “pure” reductions to be

\[
\langle h, e \rangle \rightarrow \langle h, e' \rangle
\]

if there exists a reduction $e \rightarrow e'$ in $F_{\mu}$. The pure reduction rules do not modify the heap, which should not come as a surprise as they were defined before we had a heap.

The reduction rule for allocation is

\[
\langle h, E[\text{ref } v] \rangle \rightarrow \langle h[l \mapsto v], E[l] \rangle, \quad l \notin \text{dom}(h).
\]
Here we ensure that the location allocated is new by requiring that it is not already in the domain of the heap. The reduction rule for assignment is

\[(b, E[l := v]) \rightarrow (b[l \mapsto v], E[\{\}]), \quad l \in \text{dom}(h).\]

The location we assign to must already be allocated in the heap. That is, the location must be in the domain of the heap. The assignment is done by updating the value at location \(l\) in the heap to be \(v\).

Finally the reduction rule for dereference is

\[(b, E[l]) \rightarrow (b, E[b(l)]), \quad l \in \text{dom}(h).\]

The lookup is done by applying \(b\) to \(l\) which is always defined as we require the location to be in the domain of the heap.

Before we move on and define a type system for \(F_{\mu,\text{ref}}\), we will first note some properties of the language:

**The operational semantics is non-deterministic.** In the reduction rule for allocation, we only require \(l\) to be a new location that is not in the heap. It can, however, be any new location which is the cause of the non-determinism. By leaving allocation underspecified, our system can handle different real implementations of allocation. One drawback of non-determinism is that we cannot rely on determinism in our proofs (which we have previously done).

**Evaluation can get stuck.** The reduction rules for assignment and dereference require the location to be in the heap. If this is not the case, then we are stuck. One can think of this as a memory fault. No surface language for expressing locations is provided, so to obtain a reference one has allocate it (or obtain it from a source that allocated it).

**Only mutation on the heap.** The only place we have introduced mutation is on the heap. Variables bound with a \(\lambda\)-expression can still not be modified. This is like ML but unlike C, which has mutable stack variables.

**Values in memory cells stay the same type.** The type system will enforce this invariant that memory cells always store values of the same type.

**The language is very expressive.** If we removed our fixed-point operator and recursive types, then we would still possible have non-terminism as we can make recursion through the heap. Take for instance the following program:

\[
\text{fac} := \begin{align*}
\text{let } x &= (\text{ref} (\lambda n : \text{int}. n) \\
\text{in } (x := \lambda n : \text{int}. \text{if } n = 0 \text{ then } 1 & \\
\text{else } (tx (n - 1)) \ast n); &
\end{align*}
\]

which recurses through the heap to compute the factorial of a number. The above program does this by first making a reference available to a dummy function (namely the identity in the first line). Then the program overwrites the dummy on the heap with a closure which, from the perspective of the lambda, contains one free variable that allows it refer to the closure “itself”. The last line dereferences the closure which allows for the initial application of it. This way of doing recursion is known as *Landin’s knot* \([\text{Landin}, 1964]\).

Before we move on to the formal typing system let us try to get familiar with the language. We can for instance define a type of mutable lists of elements of type \(\lambda\) by:

\[\text{mlist}(\alpha) = \mu \beta . \text{ref} (1 + \alpha \times \beta)\]

Here the first part of the sum type, \((1 + \alpha \times \beta)\), can be seen as \textit{nil} and the second component can be seen as \textit{cons}.
Exercise 1.1. Write an in-place-reverse function for linked lists in memory:

```plaintext
fun in-place-reverse (l : mlist (α)) = ...  
```

Hint: make a helper function with two arguments, a pointer and a cell.

1.2 Typing

Our typing judgment will have this form: \( \Sigma; \Delta; \Gamma \vdash e : \tau \). The \( \Sigma \) is a context for location typing, and it is defined as a list of pairs of locations and types, \( l : \tau \). This typing judgment will allow us to do a proof using progress and preservation, as we will be able to type configurations. Later on when we define a relational model, we will use a different type judgment that does not have a context for location typing.

The typing rules for the expressions we have from \( F_\mu \) remain the same, but with \( \Sigma \) added to the contexts. The remaining typing rules are:

\[
\begin{align*}
\Sigma; \Delta; \Gamma \vdash e : \tau & \quad \Sigma; \Delta; \Gamma \vdash e_1 : \text{ref } \tau \quad \Sigma; \Delta; \Gamma \vdash e_2 : \tau & \quad \Sigma; \Delta; \Gamma \vdash e : \text{ref } \tau & \quad \Sigma; \Delta; \Gamma \vdash !e : \tau
\end{align*}
\]

Notice that none of the rules make changes to the environment \( \Sigma \). \( \Sigma \) is called the store typing. We now introduce a notion of what a well-typed heap is.

\[
\Delta; \Gamma \vdash h : \Sigma \text{ iff } \text{dom}(h) = \text{dom}(\Sigma) \land \forall l \in \text{dom}(h). \Sigma; \Delta; \Gamma \vdash h(l) : \Sigma(l)
\]

We can now formulate progress and preservation which we can use to show safety of \( F_{\mu,\text{ref}} \) if we wanted to make a syntactic proof.

**Theorem 1.1** (Preservation). If \( \Sigma; \Delta; \Gamma \vdash e : \tau \), and \( \Delta; \Gamma \vdash h : \Sigma \), and \( \langle h, e \rangle \rightarrow \langle h', e' \rangle \), then there exists \( \Sigma' \supseteq \Sigma \) such that \( \Sigma'; \Delta; \Gamma \vdash e' : \tau \) and \( \Delta; \Gamma \vdash h' : \Sigma' \).

The left side of preservation looks fairly standard. It expresses that we have a well-typed configuration and that configuration can take a step. Previously, our configurations were just an expression, but now a configuration is an pair of an expression and a heap, so we require both components to be well-typed. The essence of the right side is also familiar as it says the configuration we end up in is well-typed. The new thing is that we require it to be well-typed under a store typing \( \Sigma' \) that is larger or equal to the one we had \( \Sigma \). When the expression took a step, it may have caused a new location to be allocated which we need to take into account.

**Theorem 1.2** (Progress). \( \Sigma; *; * \vdash e : \tau \) and \( *; * \vdash h : \Sigma \) then either \( e \) is a value or there exists an expression \( e' \) and a heap \( h' \) such that \( \langle e, h \rangle \rightarrow \langle e', h' \rangle \).

Exercise 1.2. Prove progress and preservation.

1.3 A first attempt on a unary logical relation for \( F_{\mu,\text{ref}} \)

Before we start developing a unary logical relation it is worth noting that this is not our final goal. What we want to have is a relational model we can use to prove contextual equivalence. However, a unary logical relation will help shed light on the challenges we will face when we move on to a relational model.

When we defined a unary logical relation for \( F_\mu \), the key idea was to use a semantic universe of types.

\[
T = \text{UPred}(V)
\]

\(^1\)Notice that store and heap are used interchangeably.
For $F_{\mu, \text{ref}}$, we would like the universe of semantic types to be indexed over the worlds of locations we have allocated so far:

\[
T = W \rightarrow \text{UPred}(V) \\
W = \text{Loc} 
\]

(You can think of a world as a semantic store typing.) Now we can make an attempt at an interpretation of the reference type:

\[
V[\text{ref } \tau] = \lambda w. \{ l \mid l \in \text{dom}(W), w(l) = V[\tau] \}
\]

The intuition here is that the location $l$ has to be allocated, and when we look up that type, then it has to be the same as if we interpreted the type. It is something like the above we want, but if we take another look at our universe of semantic types, then it is recursive! If we substitute $W$ into the definition of $T$, then we get

\[
T = (\text{Loc} \rightarrow T) \rightarrow \text{UPred}(V)
\]

Here the $T$ on the right hand occurs negatively in the “set equation”, so there is no solution if we use normal sets\(^2\). To solve this we use an idea similar to the one we need for recursively defined predicates, where we indexed predicates with numbers. Now we will use a notion of indexed set.

Before looking into this, one more intuitive comment first. Our universe of semantic types will not consist of all functions from worlds to uniform predicates but only the monotone functions:

\[
T = W \rightarrow^{\text{mon}} \text{UPred}(V) \tag{1}
\]

The ordering on worlds will be defined to be the extension ordering ($w \leq w'$ if $\text{dom}(w) \subseteq \text{dom}(w')$ and $\forall l \in \text{dom}(w), w(l) = w'(l)$). The idea is semantic types can only open when more references are allocated. Intuitively this definition says that everything that was allocated will stay allocated. In other words, references are never invalidated. This means that we will not have a model that explicitly supports garbage collection. Garbage collection is not observable, so we are okay with this.

Before we continue with the unary logical relation, we need to define a special kind of indexed sets, which we will do in the following digression.

### 1.4 Complete ordered families of equivalences

Ordered families of equivalences (o.f.e.’s) are sets equipped with a family of equivalence relations that approximate the actual equality on the set $X$. These relations must satisfy some basic coherence conditions.

**Definition 1.3** (o.f.e.). An **ordered family of equivalences** is a pair $\langle X, (\equiv_n)_{n \in \mathbb{N}} \rangle$ where $X$ is a set and for each $n$, the binary relation $\equiv_n$ is an equivalence relation on $X$ such that the relations $\equiv_n$ satisfy the following conditions

- $\equiv_0$ is the total relation on $X$, i.e., everything is equal at stage 0.
- for any $n \in \mathbb{N}$, $\equiv_{n+1} \subseteq \equiv_n$ (monotonicity)
- for any $x, x' \in X$, if $\forall n \in \mathbb{N}, x \equiv_n x'$ then $x = x'$.

We say that an o.f.e. $\langle X, (\equiv_n)_{n \in \mathbb{N}} \rangle$ is **inhabited** if there exists an element $x \in X$. \(\blacksquare\)

\(^2\)If you are not satisfied with the semantic type, then read on anyway.

\(^3\)The issue is that there are more functions in $\text{Loc} \rightarrow T$ than there are elements in $T$. We know from set theory that $\mathcal{D} \not\sim \mathcal{D} \rightarrow 2$. 

4
Example 1.4. A canonical example of an o.f.e. is a set of strings (finite and infinite) over some alphabet. The strings \( x, x' \) are \( n \)-equal, \( x \equiv^n x' \) if they agree for the first \( n \) characters.

Remark 1.5. If you are familiar with metric spaces observe that o.f.e.'s are but a different presentation of bisected 1-bounded ultrametric spaces.

Definition 1.6 (Cauchy sequences and limits). Let \( (X, \equiv) \) be an o.f.e. and \( \{x_n\}_{n=0}^\infty \) be a sequence of elements of \( X \). Then \( \{x_n\}_{n=0}^\infty \) is a Cauchy sequence if

\[
\forall k \in \mathbb{N}, \exists j \in \mathbb{N}, \forall n \geq j, x_j \equiv^n x_n
\]

or in words, the elements of the chain get arbitrarily close.

An element \( x \in X \) is the limit of the sequence \( \{x_n\}_{n=0}^\infty \) if

\[
\forall k \in \mathbb{N}, \exists j \in \mathbb{N}, \forall n \geq j, x_j \equiv^n x_n.
\]

A sequence may or may not have a limit. If it has it we say that the sequence converges. The limit is necessarily unique in this case (Exercise 1.3) and we write \( \lim_{n \to \infty} x_n \) for it.

Remark 1.7. These are the usual Cauchy sequence and limit definitions for metric spaces specialized to o.f.e.'s.

Exercise 1.3. Show that limits are unique. That is, suppose that \( x \) and \( y \) are limits of \( \{x_n\}_{n=0}^\infty \). Show \( x = y \).

One would perhaps intuitively expect that every Cauchy sequence has a limit. This is not the case in general.

Exercise 1.4. Show that if the alphabet \( \Sigma \) contains at least one letter then the set of finite strings over \( \Sigma \) admits a Cauchy sequence without a limit. The equivalence relation \( \equiv \) relates strings that have the first \( n \) characters equal.

Hint: Pick \( \sigma \in \Sigma \) and consider the sequence \( x_n = \sigma^n \) (i.e., \( x_n \) is \( n \) \( \sigma \)'s).

We are interested in spaces which do have the property that every Cauchy sequence has a limit. These are called complete. Completeness allows us to have fixed points of suitable contractive functions which we define below.

Definition 1.8 (c.o.f.e.). A complete ordered family of equivalences is an ordered family of equivalences \( (X, \equiv) \) such that every Cauchy sequence in \( X \) has a limit in \( X \).

Example 1.9. A canonical example of a c.o.f.e. is the set of infinite strings over an alphabet. The relation \( \equiv \) relates streams that agree on at least the first \( n \) elements.

Exercise 1.5. Show the claims made in Example 1.9.

To have a category we also need morphisms between (complete) ordered families of equivalences.

Definition 1.10. Let \( (X, \equiv_X) \) and \( (Y, \equiv_Y) \) be two ordered families of equivalences and \( f \) a function from the set \( X \) to the set \( Y \). The function \( f \) is

- non-expansive if for any \( x, x' \in X \), and any \( n \in \mathbb{N} \),

\[
x \equiv^n x' \implies f(x) \equiv^n f(x')
\]
• contractive if for any \( x, x' \in X \), and any \( n \in \mathbb{N} \),
\[
\frac{x}{n} x' \implies f(x)_{n+1} f(x')
\]

Exercise 1.6. Show that non-expansive functions preserve limits, i.e., show that if \( f \) is a non-expansive function and \( \{x_n\}_{n=0}^{\infty} \) is a converging sequence, then so is \( \{f(x_n)\}_{n=0}^{\infty} \) and that
\[
f \left( \lim_{n \to \infty} x_n \right) = \lim_{n \to \infty} f(x_n).
\]

The reason for introducing complete ordered families of equivalences, as opposed to just o.f.e.’s, is that any contractive function on a inhabited c.o.f.e. has a unique fixed point.

Theorem 1.11 (Banach’s fixed point theorem). Let \( (X, \{x\}_{n=0}^{\infty}) \) be a inhabited c.o.f.e. and \( f : X \to X \) a contractive function. Then \( f \) has a unique fixed point.

Proof. First we show uniqueness. Suppose \( x \) and \( y \) are fixed points of \( f \), i.e. \( f(x) = x \) and \( f(y) = y \). By definition of c.o.f.e.’s we have \( x \equiv y \). From contractiveness we then get \( f(x) \equiv f(y) \) and so \( x \equiv y \). Thus by induction we have \( \forall n, x \equiv y \). Hence by another property in the definition of c.o.f.e.’s we have \( x = y \).

To show existence, we take any \( x_0 \in X \) (note that this exists since by assumption \( X \) is inhabited). We then define \( x_{n+1} = f(x_n) \) and claim that \( x_n \equiv x_{n+m} \) for any \( n \) and \( m \) which we prove by induction on \( n \). For \( n = 0 \) this is trivial. For the inductive step we have, by contractiveness of \( f \)
\[
x_{n+1} = f(x_n)_{n+1} f(x_{n+m}) = x_{n+m+1},
\]
as required. This means that the sequence \( \{x_n\}_{n=0}^{\infty} \) is Cauchy. Now we use completeness to conclude that \( \{x_n\}_{n=0}^{\infty} \) has a limit, which we claim is the fixed point of \( f \). Let \( x = \lim_{n \to \infty} x_n \). We have (using Exercise 1.6)
\[
f(x) = f \left( \lim_{n \to \infty} x_n \right) = \lim_{n \to \infty} f(x_n) = \lim_{n \to \infty} x_{n+1} = \lim_{n \to \infty} x_n = x
\]
concluding the proof.

Definition 1.12 (The category \( \mathcal{U} \)). The category \( \mathcal{U} \) of complete ordered families of equivalences has as objects complete ordered families of equivalences and as morphisms non-expansive functions.

If you are not familiar with category theory, then think of \( \mathcal{U} \) as a collection of c.o.f.e.’s. We want functions between c.o.f.e.’s to preserve the c.o.f.e. structure which we achieve by requiring them to be non-expansive. We want to use c.o.f.e.’s instead of sets in the equation we ended last section with. For this to work out, we have to require the functions in the equation to be non-expansive.

To show the likeness between sets and we show how to define c.o.f.e. products and c.o.f.e. of functions between c.o.f.e.’s.

Example 1.13 (Product c.o.f.e.’s). For c.o.f.e.’s \( (X, \{X\}_{n=0}^{\infty}) \) and \( (Y, \{Y\}_{n=0}^{\infty}) \), we write the product as \( (X, \{X\}_{n=0}^{\infty}) \times (Y, \{Y\}_{n=0}^{\infty}) \). The product is defined as the pair \( (X \times Y, \{X' \times Y\}_{n=0}^{\infty}) \) where \( X \times Y \) is the product set between \( X \) and \( Y \). The \( n \)-equality is defined as for all \( x \) and \( x' \) in \( X \) and for all \( y \) and \( y' \) in \( Y \)
\[
(x, y) = (x', y') \iff x =_{X} x' \quad \text{and} \quad y =_{Y} y'
\]
Example 1.14 (c.o.f.e. of functions between c.o.f.e.’s). For c.o.f.e.’s \((X, (\equiv_X)_{n=0}^{\infty})\) and \((Y, (\equiv_Y)_{n=0}^{\infty})\), we write the c.o.f.e. of functions between them \((X, (\equiv_X)_{n=0}^{\infty}) \to (Y, (\equiv_Y)_{n=0}^{\infty})\), and it is defined as
\[
\{ f : X \to Y | f \text{ non-expansive} \}, (\equiv)_{n=0}^{\infty}
\]
with for all functions \(f\) and \(g\) from \(X\) to \(Y\) we have
\[
f\equiv_n g \iff \forall x \in X. f(x)\equiv_n g(x)
\]

Exercise 1.7. Show that the constructions in Example 1.13 and Example 1.14 satisfy the definition of a c.o.f.e.

From now on, we often use the underlying set \(X\) to denote a (complete) o.f.e. \((X, (\equiv)_{n=0}^{\infty})\), leaving the family of equivalence relations implicit.

If we wish to find a fixed point of a function \(f\) from a set \(X\) to a set \(Y\) we really have nothing to go on. What the o.f.e.’s give us is the ability to get closer and closer to a fixed point, if \(f\) is well-behaved. What the c.o.f.e.’s additionally give us is that the “thing” we get closer and closer to is in fact an element of the o.f.e.

1.5 A unary logical relation \(F_{\mu, ref}\)

The technical machinery and properties c.o.f.e.’s provide, can be used to make a well-founded definition of our semantic types. In this sub-section, we provide the definitions for a couple more c.o.f.e.’s, the semantic type, some of the value interpretation, and the expression interpretation.

We can find solutions to recursive c.o.f.e. equations\(^4\) when the recursion variable occurs under a “later” operator \(\triangleright\) on c.o.f.e.’s, defined as follows

Definition 1.15. Let \((X, (\equiv)_{n=0}^{\infty})\) be a c.o.f.e.. We define
\[
\triangleright (X, (\equiv)_{n=0}^{\infty}) = (X, (\equiv')_{n=0}^{\infty})
\]
where for all \(x\) and \(x'\) in \(X\)
\[
x \equiv_n x' \iff \begin{cases} \top & \text{if } n = 0 \\ \forall j < n. (x \equiv j \implies x' \equiv j) & \text{if } n > 0 \end{cases}
\]
The set in the c.o.f.e. remains the same, but the equality relation is “moved” so it is equal at an earlier time.

To define \(T\) as a c.o.f.e., we will show how uniform predicates can be seen as a c.o.f.e. The general construction is as follows:

Definition 1.16. Let \((A, \leq)\) be a preordered set. Now define \(\text{UPred}(A)\) to be
\[
\text{UPred}(A) = \{ p \subseteq \mathbb{N} \times A | \forall (k, a) \in p. \forall j \leq k. \forall b \geq a. (j, b) \in p \}
\]
We further define the \(k\)-cut of \(p \in \text{UPred}(A)\) to be
\[
| p |_k = \{ (j, a) \in p | j < k \}.
\]

\(^4\)The proof of this is beyond the scope of this note. Please refer to Section 6.2 of the tutorial notes by Birkedal and Bizjak 2014.
We use this to define a family of equivalences \( \left( \bigwedge_n \right)_n = \bigwedge_n \) to be, for \( p \) and \( q \) in \( \text{UPred}(A) \)

\[
p_n = q \iff \lfloor p \rfloor_n = \lfloor q \rfloor_n
\]

**Example 1.17.** Consider the following \( p \) and \( q \):

\[
p = \{(0,a), (1,b)\}
\]

\[
q = \{(0,a), (1,c)\}
\]

Here we have \( p_0 = q \). This is trivially true as a 0-cut cuts everything away, which also fits nicely with the fact that we want to use it in a c.o.f.e. We also have \( p_1 = q \) as 

\[
\lfloor p \rfloor_1 = \{(0,a)\}
\]

\[
\lfloor q \rfloor_1 = \{(0,a)\}
\]

Finally, we do not have \( p_2 = q \) as \((1,b) \neq (1,c)\). □

Now one can check that \( \left( \text{UPred}(A), \left( \bigwedge_n \right)_n = \bigwedge_n \right) \) is a well-defined c.o.f.e.

Now let us consider what c.o.f.e. \( \left( N \xrightarrow{\text{fin}} \ast T \right) \) is. For a c.o.f.e. \( \left( X, \left( \bigwedge_n \right)_n = \bigwedge_n \right) \) we define 

\[
\left( N \xrightarrow{\text{fin}} X \right) = \left( \bigwedge_n \right)_n = \bigwedge_n \]

\[\text{to be the set of all partial functions from } N \text{ to } X \text{ where for any } f \text{ and } f' \text{ in } N \xrightarrow{\text{fin}} X \]

\[
f_n = f' \iff \begin{cases}
\top & \text{if } n = 0 \\
\text{dom}(f) = \text{dom}(f') \land \forall i \in \text{dom}(f). f(i)_n = f'(i) & \text{if } n > 0
\end{cases}
\]

We can see this as a generalization of tuples. The \( \text{dom}(f) = \text{dom}(f') \) can be read as \textit{tuples of equal length} and \( \forall i \in \text{dom}(f). f(i)_n = f'(i) \) can be read as \textit{has } n \text{-equal elements}.

Now we consider what the monotone function \( \xrightarrow{\text{mon}} \) used in Equation 1 should formally be. As a reminder, a function \( g : X \rightarrow Y \) is monotone if the following holds:

\[
\forall x, y \in X. x \leq_X y \Rightarrow g(x) \leq_Y g(y)
\]

The less than or equal in \( \left( N \xrightarrow{\text{fin}} T \right) \) is defined as

\[
f \leq f' \iff \text{dom}(f) \subseteq \text{dom}(f') \land \forall n \in \text{dom}(f). f(n) = f'(n)
\]

and for \( \text{UPred} \) it is just \( \subseteq \). As we have a function between c.o.f.e.'s, we require \( \xrightarrow{\text{mon}} \) to be non-expansive. If we used all non-expansive functions (and not just the monotone and non-expansive), then we know from Example 1.14 that we have a c.o.f.e. of functions. Since we now require monotonicity, the resulting c.o.f.e. has fewer elements than the c.o.f.e. of functions from Example 1.14. Does that still preserve the limits and thus the c.o.f.e. structure? One can check that the limits are in fact preserved.

In [Birkedal et al., 2011] they consider the recursive domain equation

\[
\hat{T} \cong \left( \left( N \xrightarrow{\text{fin}} \hat{T} \right) \xrightarrow{\text{mon}} \text{UPred}(Val) \right)
\]

All the ingredients on the right-hand side have been defined and we know that recursive domain equations on this form can be solved, so the isomorphism exists. The isomorphism gives us two non-expansive
functions that allow us to go back and forth between the two. The functions are non-expansive because both sides of the isomorphism are c.o.f.e.’s.

We introduce a notion of worlds. A world semantically describes a heap in the sense that it specifies what kind of value should be stored at a location at any given point. The c.o.f.e. of worlds \( W \) is

\[
W = \mathbb{N}^{\text{fin}} \rightarrow T
\]

and the semantic universe of types is indexed over the world

\[
T = W \rightarrow \text{UPred}(Val)
\]

The semantics of the types in context is defined as a non-expansive function

\[
[\Delta \vdash \tau] : T^{[\Delta]} \rightarrow T
\]

The function takes an environment that gives meaning to each of the type variables. It is like the relational substitution \( \rho \) we used for universal and existential types. The value interpretation of unit type is

\[
\forall \Delta \vdash 1 \quad T, \rho \quad \lambda w. \{(n,()) | n \in \mathbb{N}\}
\]

This definition is pretty standard. The only new thing is that we indexed the relation with a world, but we do not use the world for anything here. The value interpretation of the reference type is defined as

\[
\forall \Delta \vdash \text{ref} \quad \tau, \rho \quad \lambda w. \{(k,l) | l \in \text{dom}(w) \land w(l) \overset{k}{=} V\forall \Delta \vdash \tau, \rho\}
\]

Here we only require what is on location \( l \) in the world \( w \) to be \( k \)-equal with the value interpretation of \( \tau \). Intuitively, the \( k \) gives us the “time” for which this remains good. The \( k \)-equality is also there for technical reasons. With it the interpretation becomes non-expansive, but if it is removed, then this is no longer the case.

As part of defining the value interpretation, we need to show that it is well-defined. Part of this is to show it is non-expansive, that is, for any worlds \( w \) and \( w' \) and any \( n \in \mathbb{N} \),

\[
w^n = w' \quad \implies \quad V[\tau], w^n = V[\tau], w'
\]

In the case where \( \tau = \text{ref} \quad \tau' \) we need to show for any \( w \) and \( w' \) and any \( n \in \mathbb{N} \),

\[
w^n = w' \quad \implies \quad \{ (k,l) | l \in \text{dom}(w) \land w(l) \overset{k}{=} V[\tau], w' \}
\]

**Exercise 1.8.** Show that the interpretation of the reference type is non-expansive.

Hint: Use that equality relations are transitive and the many definitions of \( n \)-equality.

When we define the interpretation of the function type, we use our usual mantra related argument to related results. But in what world should they be related? In all future worlds:

\[
\forall \Delta \vdash \tau \rightarrow \tau' , \rho \quad \lambda w. \{ (k,v) | \forall w' \supseteq w. \forall i \leq k. (i,v') \in V[\Delta \vdash \tau], w' \implies (i,v') \in V[\Delta \vdash \tau], w' \}
\]
This definition allows $w'$ to have grown relatively to $w$ which makes sense as more locations may be allocated in the time between the definition and the application of a function. This also makes sure that the interpretation becomes monotone. We also require a smaller index $i$ as a number of execution steps may have occurred. The definitions of the value interpretation for the remaining types can be found in [Birkedal et al., 2011, Figure 1]. Lemma 2.5 in [Birkedal et al., 2011] states the properties required for the value interpretation to be well-defined.

Now we define the expression interpretation.

$$
E[\Delta \vdash \tau]_\rho = \lambda w. \{ (k, e) | \forall i \leq k. \forall b, b', e'.
\begin{align*}
&b : k \ w \land \langle b, e \rangle \rightarrow^i \langle b', e' \rangle \\
&\implies \\
&\exists w' \supseteq w. b' : k - i \ w' \land (k - i, e') \in \mathcal{V}[\Delta \vdash \tau]_\rho w'
\end{align*}
$$

The definition says: $b : k \ w$, take a heap that fits our semantic world (up to $k$ steps). $\langle b, e \rangle \rightarrow^i \langle b', e' \rangle$, run $e$ on this heap for a number of steps until we reach something irreducible. $\exists w' \supseteq w.$, then there exists some future semantic world, $b' : k - i \ w'$, which the heap we ended up in fits (up to $k - i$ steps - we used $i$ steps), and then the $e'$ we ended up with will be in the value interpretation of $\tau$ at time $k - i$ in world $w'$. Notice that unlike the value interpretation of the function type, here we get to choose $w'$.

Next we want to show the fundamental theorem of logical relations. As usual, it says that if a term is well-typed, then if we close it off, then it is in the relation (or in the expression interpretation). The theorem is precisely stated in [Birkedal et al., 2011, Theorem 2.6].

We will now give an informal sketch for the reference type case of the proof. First using the induction hypothesis, it is possible to argue that $ref e \rightarrow^* ref v$ and that this $v$ is in the value interpretation of $\tau$. Now starting from a heap $h$ where we know the heap fits a semantic world $w (h : w)$, we know from the operational semantics that $\langle h, ref v \rangle \rightarrow \langle h[l \mapsto v], () \rangle$. Then we need to choose a new semantic world that fits the heap we end up in. We choose $w[l \mapsto \mathcal{V}[\tau]_\rho]$ which we then need to show actually fits the world. The important thing to notice in this very informal argument is that we got to choose the semantic world for the new heap.

Hopefully, the unary logical relation presented here has shed light on some of the challenges one face when defining a logical relation for a language with references. Next section defines a binary logical relation for a similar language in a more formal manner than the presentation in this section. In the unary logical relation presented here, the worlds mapped from locations to semantic types. In next section, the worlds are defined to be from invariant names (which is just a natural number) to invariants which allows locations to be related in refined ways.

---

5We have not introduced all the machinery needed to state it, so we will make do with this informal version.
2 Relational model

2.1 Syntax and operational semantics

In Figure 1 we define the syntax of a higher-order functional language with general references and existential types. We assume countably infinite and disjoint sets of type variables, term variables and locations, with $\alpha$ ranging over type variables and $x$ over term variables and $l$ over locations. We use a Curry-style presentation and thus do not annotate $\lambda$-abstractions or pack/unpack with types. The typing rules have the form $\Delta; \Gamma \vdash e : \tau$, where $\Delta$ is a context of type variables and $\Gamma$ is a context of term variables. The well-formed type judgment, $\Delta \vdash \tau$ expresses that all free type variables in $\tau$ are bound in $\Delta$. The typing rules can be found in Figure 2. The well-formed type rules are standard and have been omitted.

Note that our type system does not include store typings, assigning types to locations. It is not necessary to include store typings, since well-typed expressions do not contain any location constants. Store typings are typically used to facilitate syntactic progress and preservation proofs. However, they are unnecessary for our semantic approach.

The operational semantics is defined as a small-step reduction relation between configurations consisting of an expression $e$ and a heap $h : e, h \rightarrow e', h'$. A heap is a finite map from locations to values. Figure 3 includes the reduction rules. Note that dereferencing or assigning to a location that has not already been allocated results in a stuck configuration. It will follow from our logical relation that well-typed programs never get stuck and thus never try to dereference a location that has not been allocated.

From the small-step reduction semantics we define a step-indexed reduction relation, $e, h \rightarrow^n e', h'$, which expresses that $e, h$ reduces in $n$ steps to $e', h'$. We count every reduction step. We use $e, h \rightarrow^* e', h'$ to denote the transitive closure of the small-step reduction relation. Our formal definition of contextual approximation is given in Definition 2.1 below. We say that $e_i$ contextually approximates $e_j$ if for any closing context $C$ of unit type, if $C[e_j]$ terminates with value (), then $C[e_i]$ terminates with value (). Since well-typed expressions do not contain any location constants, we can simply reduce $C[e_j]$ and $C[e_i]$ with an empty heap. The $C : (\Delta; \Gamma, \tau) \Rightarrow (\Delta', \Gamma', \tau')$ relation expresses that context $C$ takes a term $e$ such that $\Delta; \Gamma \vdash e : \tau$ to a term $C[e]$ such that $\Delta'; \Gamma' \vdash C[e] : \tau'$. The rules for $C : (\Delta; \Gamma, \tau) \Rightarrow (\Delta', \Gamma', \tau')$ are standard and have been omitted.

**Definition 2.1 (Contextual approximation).** If $\Delta; \Gamma \vdash e_i : \tau$ and $\Delta; \Gamma \vdash e_j : \tau$, then $e_i$ contextually approximates $e_j$, written $\Delta; \Gamma \vdash e_i \leq_{ctx} e_j : \tau$ if,

$\forall C : (\Delta; \Gamma, \tau) \Rightarrow (\Delta'; \Gamma', \tau') \Rightarrow (\Delta''; \Gamma'', \tau'').$}

We refer to $e_i$ in $\Delta; \Gamma \vdash e_i \leq_{ctx} e_j : \tau$ as the left expression or implementation and we refer to $e_j$ as the right expression or specification. Contextual equivalence, $\Delta; \Gamma \vdash e_i \cong_{ctx} e_j : \tau$, is then defined as the conjunction of left-to-right contextual approximation and right-to-left contextual approximation.

| $\tau, \sigma$ | ::= | $1 | N | \tau \times \sigma | \tau \rightarrow \sigma | \text{ref } \tau | \exists x. \tau | \alpha$ |
| $v \in \text{Val}$ | ::= | () | $\lambda x f. (x, e) | e | \text{pack } v | (v_1, v_2)$ |
| $e \in \text{Exp}$ | ::= | $v | e_1 e_2 | (e_1, e_2) | \text{fst } e \mid \text{snd } e$ |
| $K \in \text{ECtx}$ | ::= | $\bullet | K e \mid v K \mid (K, e) \mid (v, K) \mid \text{fst } K \mid \text{snd } K$ |

Figure 1: Types, terms and evaluation contexts
Typing rules

\[
\begin{align*}
\frac{\Delta \vdash \tau \quad \Delta \vdash \sigma}{\Delta \vdash \tau \times \sigma} &\quad \frac{\Delta \vdash \tau \quad \Delta \vdash \sigma}{\Delta \vdash \tau \rightarrow \sigma} &\quad \frac{\Delta \vdash \tau}{\Delta, x : \tau \vdash x : \tau} &\quad \frac{}{\Delta \vdash \text{ref}} &\quad \frac{}{\Delta \vdash \exists x. \tau} &\quad \frac{}{\Delta, \alpha \vdash \alpha}
\end{align*}
\]

\[
\Delta; \Gamma \vdash \Delta \vdash \tau \quad \Delta; \Gamma \vdash \Delta \vdash \sigma \quad \Delta; \Gamma \vdash \Delta \vdash \tau
\]

\[
\frac{\Delta \vdash \tau}{\Delta; \Gamma, x : \tau \vdash x : \tau} &\quad \frac{\Delta; \Gamma \vdash () : 1}{\Delta ; \Gamma \vdash \text{fix } f (x). \ e : \tau \rightarrow \sigma} &\quad \frac{}{\Delta ; \Gamma \vdash e_1 : \tau \rightarrow \sigma} \quad \frac{}{\Delta ; \Gamma \vdash e_2 : \tau}
\]

\[
\frac{}{\Delta ; \Gamma \vdash e : \text{ref } \sigma} &\quad \frac{}{\Delta ; \Gamma \vdash e_1 : \text{ref } \tau} &\quad \frac{}{\Delta ; \Gamma \vdash e_2 : \tau} &\quad \frac{}{\Delta ; \Gamma \vdash e : \sigma [\tau / \alpha]} &\quad \frac{}{\Delta ; \Gamma \vdash \text{pack } e : \exists x. \sigma}
\]

\[
\frac{}{\Delta ; \Gamma \vdash e_1 : \exists x. \tau} &\quad \frac{}{\Delta, \sigma ; \Gamma, x : \tau \vdash e_2 : \sigma} &\quad \frac{}{\Delta \vdash \tau} &\quad \frac{}{\Delta ; \Gamma \vdash (e_1, e_2) : \tau \times \sigma}
\]

\[
\frac{}{\Delta ; \Gamma \vdash e : \tau \times \sigma} &\quad \frac{}{\Delta ; \Gamma \vdash f s t \ e : \tau} &\quad \frac{}{\Delta ; \Gamma \vdash s n d \ e : \sigma}
\]

Figure 2: The typing rules

Operational semantics

\[
\begin{align*}
\text{EvalFix} & \quad \text{EvalUnpack}
\end{align*}
\]

\[
(fix \ f (x), \ e) \nu, \ b \rightarrow e[fix \ f (x). \ e/f, \ \nu/x], \ b &\quad \text{unpack} \ (\text{pack } \nu) \ as \ x \ in \ e, \ b \rightarrow e[\nu/x], \ b
\]

\[
\begin{align*}
\text{EvalRead} & \quad \text{EvalWrite} & \quad \text{EvalAlloc} & \quad \text{EvalCtx}
\end{align*}
\]

\[
\begin{align*}
&\begin{array}{ll}
&\begin{array}{l}
\text{l} \in \text{dom}(b)
\end{array}\\
&l \in \text{dom}(b)
\end{array} &
\begin{array}{l}
\text{l} : \nu, \ b \rightarrow () , \ b[l \mapsto \nu]\\
\text{ref } \nu, \ b \rightarrow l, \ b[l \mapsto \nu]
\end{array} &
\begin{array}{l}
\text{eval } \text{ctx}
\end{array} &
\begin{array}{l}
\text{e, } \ b \rightarrow e', \ b'\\
\text{e, } \ b \rightarrow e', \ b' \rightarrow^* \ e'', \ b''
\end{array}
\]

\[
\begin{align*}
\text{EvalFst} & \quad \text{EvalSnd}
\end{align*}
\]

\[
\begin{align*}
\text{fst } (\nu_1, \nu_2), \ b \rightarrow \nu_1, \ b &\quad \text{snd } (\nu_1, \nu_2), \ b \rightarrow \nu_2, \ b
\end{align*}
\]

\[
\begin{align*}
\text{and}
\end{align*}
\]

\[
\begin{align*}
e, \ b \rightarrow \delta \ e, \ b &\quad e, \ b \rightarrow e', \ b' &\quad e', \ b' \rightarrow^* \ e'', \ b''
\end{align*}
\]

Figure 3: The reduction rules
2.2 Logical relation

Assume a c.o.f.e. Inv and isomorphism
\[ \xi : \text{Inv} \cong (\mathbb{N} \rightarrow \text{Inv} \rightarrow \text{UPred}(\text{Heap} \times \text{Heap})) \]

Define Type and World as follows

\[ \text{World} \overset{\text{def}}{=} \mathbb{N} \rightarrow \text{Inv} \quad \text{Type} \overset{\text{def}}{=} \text{World} \rightarrow \text{UPred}(\text{Val} \times \text{Val}) \]

Reference invariant

\[ \text{inv} : \text{Type} \times \mathcal{L} \times \mathcal{L} \rightarrow \text{World} \rightarrow \text{UPred}(\text{Heap} \times \text{Heap}) \]

We use \( \widehat{\text{Inv}} \) as a shorthand for \( \text{World} \rightarrow \text{UPred}(\text{Heap} \times \text{Heap}) \).

Heap satisfaction. To define the relational interpretation of types, we first need to define heap satisfaction, which expresses when two heaps are related at a given world. Intuitively, this is the case when the heaps satisfy all invariants in world. However, to support local reasoning about invariant satisfaction, we borrow the idea of ownership from separation logic Reynolds [2002] and require each invariant to hold for a disjoint part of the heap. Two heaps \( h_I \) and \( h_S \) are thus related, if they can be split into disjoint parts \( h_I1, h_I2, \ldots, h_IS \) and \( h_S1, h_S2, \ldots, h_S5 \), respectively, such that \( h_I1 \) and \( h_I5 \) are related by the \( j \)th invariant:

\[
[ W ]_I \overset{\text{def}}{=} \{(n + 1, h_I, h_S) \mid l_I \in \text{dom}(h_I) \land l_S \in \text{dom}(h_S) \land (n, h_I(l_I), h_S(l_I)) \in \nu(W) \}
\]

The \( I \) subscript is a set of invariant names, indicating which invariants are active (i.e., currently required to hold). Since each invariant asserts exclusive ownership of the parts of each heap that are related by the invariant, we can use \( I \) to reason locally about satisfaction of individual invariants. This is captured by Lemma 2.2. We use \( [ W ] \) as shorthand for heap satisfaction in the case where all invariants are active, i.e., \( [ W ]_{\text{dom}(W)} \).

**Lemma 2.2.**

\[ \forall W \in \text{World}. \forall h_I, h_S \in \text{Heap}. \forall n \in \mathbb{N}. \forall I_1, I_2 \in \mathcal{P}(\mathbb{N}). (n, h_I, h_S) \in [ W ]_{I_1 \cup I_2} \iff (\exists h_I1, h_I2, h_I5 \in \text{Heap}. \quad \begin{align*}
(h_I &= h_I1 \cup h_I2 \land h_I5 = h_I1 \cup h_I25) \\
\land (n, h_I11, h_I15) \in [ W ]_{I_1} \land (n, h_I22, h_I25) \in [ W ]_{I_2}
\end{align*} \]

As expected, two heaps satisfy the heap invariant \( \text{inv}(\nu, l_I, l_S) \) if and only if \( h_I(l_I) \) and \( h_S(l_S) \) contain \( \nu \)-related values (Lemma 2.3). Note that Lemma 2.3 requires that there is at least one reduction step left, even just to prove that \( l_I \) and \( l_S \) are in the domain of \( h_I \) and \( h_S \).

**Lemma 2.3.**

\[ \forall W \in \text{World}. \forall \nu \in \text{Type}. \forall h_I, h_S \in \text{Heap}. \forall n \in \mathbb{N}. \quad n > 0 \land \xi(W(\nu)) \overset{n}{=} \text{inv}(\nu, l_I, l_S) \Rightarrow (n, h_I, h_S) \in [ W ]_0 \iff l_I \in \text{dom}(h_I) \land l_S \in \text{dom}(h_S) \land (n - 1, h_I(l_I), h_S(l_S)) \in \nu(W) \]
To ensure that definitions using heap satisfaction are suitably non-expansive, we require heap satisfaction to satisfy the following non-expansiveness property (Lemma 2.4). Intuitively, this property holds because we “go down” one step index from $n$ to $n - 1$ in the definition of heap satisfaction.

**Lemma 2.4.**

\[ \forall W_1, W_2 \in \text{World. } \forall b_1, b_3 \in \text{Heap. } \forall n \in \mathbb{N}. \]
\[ W_1 \overset{\text{world}}{=} W_2 \land (n, b_1, b_3) \in [W_1] \Rightarrow (n, b_1, b_3) \in [W_2] \]

**Sketch.** Assume that two heap parts $b_1$ and $b_3$ are $n - 1$ related at $\xi(W_1(i))$. By non-expansiveness of $\xi$ it follows that $\xi(W_1(i))$ is $n$ equivalent to $\xi(W_2(i))$ in c.o.f.e. $\triangleright \text{inv}$. Since $n - 1 < n$ it follows that they are also $n - 1$ equivalent in c.o.f.e. \text{inv}. Hence, by the equivalence on uniform predicates, $\xi(W_1(i))(W_1)$ and $\xi(W_2(i))(W_2)$ agree on all step-indices strictly below $n - 1$. It follows that $b_1$ and $b_3$ are $n - 1$ related at $\xi(W_1(i))(W_1)$.

**Expression closure**

\[ \mathcal{E} : \text{Type} \rightarrow \text{World} \overset{\text{mon}}{\rightarrow} \text{UPred(Exp \land Exp)} \]

\[ \mathcal{E}(v) \overset{\triangleleft}{=} \lambda W. \{(n, e_j, e_3) | \forall m \leq n. \forall i < m. \forall b_1, b_3, b_3' \in \text{Heap. } \forall e_j. \forall W' \geq W. \]
\[ \quad m > 0 \land (m, b_1, b_3) \in [W'] \land e_j, b_3' \rightarrow e'_j', b'_3' \Rightarrow \]
\[ \quad \exists v_3, b_3'. \exists W''. \forall W'. e_j, b_3 \rightarrow v_3, b_3' \land e_j \in \text{Val} \land \]
\[ \quad (m - i, b_3', b'_3) \in [W''] \land (m - i, e'_j, v_3) \in \nu(W'') \}

**Lemma 2.5.**

\[ \forall W_1, W_2 \in \text{World. } \forall v \in \text{Type. } \forall n \in \mathbb{N}. \forall e_j, e_3 \in \text{Exp.} \]
\[ W_1 \overset{\text{world}}{=} W_2 \land (n, e_j, e_3) \in \mathcal{E}(v)(W_1) \Rightarrow (n, e_j, e_3) \in \mathcal{E}(v)(W_2) \]

**Sketch.** The result follows easily from non-expansiveness of $\nu$, Lemma 2.4, and the following property that allows an $n$ equivalence to be extended to a future world.

\[ \forall W_1, W_2, W_2' \in \text{World. } \forall n. \]
\[ W_1 \overset{\text{world}}{=} W_2 \land W_2 \leq W_1' \Rightarrow \exists W_2' \geq W_2. W_1' \overset{\text{world}}{=} W_2' \]

**Value relation**

\[ \forall [\Delta \vdash \tau]_\rho(W) \overset{\triangleleft}{=} \{(n, l_1, l_3) | \exists \xi \in \text{dom}(W), \xi(W(i)) \overset{\triangleright \text{inv}}{=} \nu_1(W)\}\]
\[ \forall [\Delta, \alpha \vdash \tau]_\rho(W) \overset{\triangleleft}{=} \rho(\alpha)(W) \]
\[ \forall [\Delta \vdash \exists x. \tau]_\rho(W) \overset{\triangleleft}{=} \{(n, \text{pack } v_1, \text{pack } v_3) | \exists v \in \text{Type. } (n, v_1, v_3) \in [\Delta, \alpha \vdash \tau]_\rho[\alpha := \nu_1](W)\}\]
We unfold the definition of expression closure: take 

\[
\mathcal{V}[\Delta \vdash \Gamma]_\rho(W) \overset{\text{def}}{=} \{(n, \gamma_1, \gamma_3) \mid \forall (x : \tau) \in \Gamma. (n, \gamma_1(x), \gamma_3(x)) \in \mathcal{V}[\Delta \vdash \tau]_\rho(W)\}
\]

**Lemma 2.6.** The logical relation is well-defined. In particular,

- \(\text{inv}(v, l_1, l_3)\) is non-expansive and monotone and \(\text{inv}(v, l_1, l_3)(W)\) is downwards-closed for all \(v \in \text{Type}\), \(l_1, l_3 \in \text{Loc}\) and \(W \in \text{World}\).
- \(\mathcal{V}[\Delta \vdash \tau]_\rho\) is non-expansive and monotone and \(\mathcal{V}[\Delta \vdash \tau]_\rho(W)\) is downwards-closed for all \(\rho \in \text{Type}^\Delta\) and \(W \in \text{World}\).

**Definition 2.7** (Logical relation).

\[
\Delta; \Gamma \models e_1 \leq_{\text{log}} e_3 : \tau \iff \forall n \in \mathbb{N}. \forall W \in \text{World}. \forall \rho \in \text{Type}^\Delta. (n, \gamma_1, \gamma_3) \in \mathcal{V}[\Delta \vdash \Gamma]_\rho(W).
\]

\[
(n, \gamma_1(e_1), \gamma_3(e_3)) \in \mathcal{E}(\mathcal{V}[\Delta \vdash \tau]_\rho(W))
\]

\[
\square
\]

The logical relation is compatible with the typing rules of the language. Here we include two of the interesting cases of the compatibility proof, namely dereference and unpack. For illustrative purposes we take a more constrained versions of the lemmas where the arguments of the dereferencing and unpacking operations are values.

**Lemma 2.8.** If \(\Delta; \Gamma \models v_l : \text{ref} \tau\) then \(\Delta; \Gamma \vdash !v_l : !v_s : \tau\).

**Proof.** We unfold the definition of the logical relation: take \(n \in \mathbb{N}, W \in \text{World}, \gamma_1, \gamma_3 \in \text{Val}^\Gamma\) and \(\rho \in \text{Type}^\Delta\) such that

\[
(n, \gamma_1, \gamma_3) \in \mathcal{V}[\Delta \vdash \Gamma]_\rho(W)
\]

We unfold the definition of expression closure: take \(m \leq n, i < m, W' \geq W\) and \(b_i, b'_i, b_s \in \text{Heap}\) such that

\[
(m, b_i, b_s) \in [W']
\]

\[
!\gamma_i(v_i), b_i \rightarrow^i e_i', b'_i \not\!
\]

Since \(\gamma_i(v_i)\) and \(\gamma_3(v_s)\) are both values, we can use the assumption to obtain a world \(W'' \geq W'\) such that \((m, b_i, b_s) \in [W'']\) and

\[
(m, \gamma_i(v_i), \gamma_3(v_s)) \in \mathcal{V}[\text{ref} \tau]_\rho(W'').
\]

By definition of the interpretation of reference types, our values must be locations, and they must be related by the reference invariant in the world. In other words, we get two locations \(l_1\) and \(l_3\) and an invariant identifier \(i \in \text{dom}(W'')\) such that

\[
\gamma_i(v_i) = l_1 \quad \gamma_3(v_s) = l_3 \quad \xi(W''(i)) \equiv \text{inv}(\mathcal{V}[\Delta \vdash \tau]_\rho, l_1, l_3)
\]

Using Lemma 2.2 we can obtain parts of \(b_i\) and \(b_s\) that satisfy the invariant \(\xi\): we have \(b_j \subseteq b_i\) and \(b'_j \subseteq b_s\) such that \((m, b'_j, b_j) \in [W'']_{l_i}\). Since \(0 \leq i < m\), we can now use Lemma 2.3 which gives us that,

\[
l_j \in \text{dom}(b'_j) \quad l_3 \in \text{dom}(b_j) \quad (m - 1, b_j(l_i), b'_j(l_3)) \in \mathcal{V}[\Delta \vdash \tau]_\rho(W'')
\]
Now we can establish, by definition of the operational semantics, that \( i = 1, e'_i = b_j(l_j) = b'_j(l'_j) \) and \( b'_j = b_j \). We are now ready to pick witnesses required by the expression closure. We pick \( v_s = b_2(l_2), b'_2 = b_2 \) and use the same world, \( W'' \). Clearly, \( !I_k, b_2 \rightarrow^* b_2(l_2) \). \( b_2 \) holds trivially and since the heaps did not change, we get the heap-satisfaction obligation by downwards-closure. Finally, we need to show that \( (m - 1, b_2(l_2), b_1(l_1)) \in V[\Delta \vdash \tau]_{\rho}(W'') \), which is precisely what we obtained from Lemma 2.3 since \( b_k(l_k) = b'_k(l'_k) \) for \( k \in \{I, S\} \).

\[\square\]

**Lemma 2.9.** If \( \Delta; \Gamma \models v_i \leq e_s : \tau \) and \( \Delta, x; \Gamma, x : \tau \vdash e_j \leq e_s : \sigma \) and \( \Delta \vdash \sigma \) then
\[
\Delta; \Gamma \models \text{unpack } v_i \text{ as } x \text{ in } e_j \leq \text{unpack } v_s \text{ as } x \text{ in } e_s : \sigma
\]

**Proof.** We unfold the definition of the logical relation: take \( n \in \mathbb{N} \), \( W \in \text{World} \), \( \gamma_i, \gamma_S \in \text{Val}^\Delta \) such that
\[
(n, \gamma_I, \gamma_S) \in V[\Delta \vdash \Gamma]_{\rho}(W)
\] (2)

We unfold the definition of expression closure: take \( m \leq n, i < m, W' \geq W \) and \( b_j, b'_j, b_S \in \text{Heap} \) such that
\[
(m, b_j, b_S) \in [W']
\]
and
\[
\text{unpack } \gamma_I(v_i) \text{ as } x \text{ in } \gamma_I(e_j), b_j \rightarrow^i e'_j, b'_j \not\rightarrow
\] (3)

Since \( \gamma_I(v_i) \) and \( \gamma_S(v_S) \) are both values, we can use the first assumption to obtain a world \( W'' \geq W' \) such that \( (m, b_j, b_S) \in [W''] \) and
\[
(m, \gamma_I(v_I), \gamma_S(v_S)) \in V[\Delta \vdash \exists x : \tau]_{\rho}(W'').
\]

By definition of interpretation of existential types, we obtain two values, \( v'_I \) and \( v'_S \), and \( v \in \text{Type} \) such that
\[
\gamma_I(v_I) = \text{pack } v'_I
\]
\[
\gamma_S(v_S) = \text{pack } v'_S
\]
\[
(m, v'_I, v'_S) \in V[\Delta, x \vdash \tau]_{\rho[\alpha \rightarrow v]}(W'') \]
(4)

Looking at our reduction [3], we now observe that \( i > 0 \) and
\[
\text{unpack } \gamma_I(v_I) \text{ as } x \text{ in } \gamma_I(e_j), b_j \rightarrow \gamma_I(e_j)[v'_I/x_I], b_j \rightarrow^{i-1} e'_j, b'_j \not\rightarrow.
\] (5)

Note that \( \gamma_I(e_j)[v'_I/x_I] = (\gamma_I[x \rightarrow v'_I])(e_j) \).

It is now time to turn to the second of our assumptions. We instantiate the definition of the logical relation with \( m - 1, W'', \gamma_I[x \rightarrow v'_I], \gamma_S[x \rightarrow v'_S] \), and \( \rho[\alpha \rightarrow v] \). We need to show that the substitutions are related, i.e., that
\[
(m - 1, \gamma_I[x \rightarrow v'_I], \gamma_S[x \rightarrow v'_S]) \in V[\Delta, x : \tau]_{\rho[\alpha \rightarrow v]}(W'').
\]

Which by definition of context relation is the same as showing
\[
(m - 1, \gamma_I[x \rightarrow v'_I](y), \gamma_S[x \rightarrow v'_S](y)) \in V[\Delta, x : \tau]_{\rho[\alpha \rightarrow v]}(W'')
\]
for any \((y : \tau') \in \Gamma, x : \tau\). For all the variables other than \(x\) this holds by \([2]\), weakening, world monotonicity, and downwards-closure; for \(x\) we obtained the necessary property relating \(v'_{\tau}\) to \(v'_{\tau}\) from the first assumption (that was property \([4]\) using downwards-closure).

By showing the substitutions to be related, we have learned that the expressions \(e_{\tau}\) and \(e_{\tau}\) are also related, after applying the substitutions:

\[
(m - 1, (\gamma_f(e_{\tau}))[v'_{\tau}/x], (\gamma_s(e_{\tau}))[v'_{\tau}/x]) \in \mathcal{E}(\mathcal{V}[\Delta; \alpha \vdash \tau_{\beta \rightarrow \kappa}](W''')).
\]

If we unfold the definition of the expression closure, we notice that if we instantiate it with \(m - 1, i - 1, b_i, b_i', b_i, W''\) we can use it to complete the proof, since \(\text{unpack} \gamma_s(e_{\tau})\) as \(x\) in \(\gamma_s(e_{\tau})[v'_{\tau}/x], b_i \rightarrow \gamma_s(e_{\tau})[v'_{\tau}/x], b_i\). Thus, it suffices to show that

\[
(\gamma_f(e_{\tau}))[v'_{\tau}/x], b_i \rightarrow^i e_i, b_i' \not\rightarrow (m - 1, b_j, b_j) \in [W''']
\]

However, the first of these properties we already obtained (it was reduction \([5]\)), and the second holds by downwards-closure of erasure.

The fundamental theorem of logical relations \((\text{Lemma 2.10})\) and a similar property for contexts \((\text{Lemma 2.11})\) follow as corollaries of the compatibility lemmas. The proofs follow by induction on the respective typing derivations.

**Lemma 2.10.** If \(\Delta; \Gamma \vdash e : \tau\) then \(\Delta; \Gamma \models e \leq_{\log} e : \tau\).

**Lemma 2.11.** For any context \(C\) such that \(C : (\Delta; \Gamma, \tau_i) \leadsto (\Delta_0; \Gamma_0, \tau_0)\), if \(\Delta; \Gamma_0 \vdash e_j \leq_{\log} e_j : \tau_j\) then \(\Delta_0; \Gamma_0 \vdash C[e_j] \leq_{\log} C[e_j'] : \tau_0\).

**Theorem 2.12** (Soundness). If \(\Delta; \Gamma \vdash e_j \leq_{\log} e_j : \tau\) then \(\Delta; \Gamma \vdash e_j \leq_{\text{ctx}} e_j : \tau\).

**Proof.** Let \(C\) be an arbitrary context such that \(C : (\Delta; \Gamma, \tau) \leadsto (\dashv; \dashv, 1)\) and \(C[e_j], [] \rightarrow^* (\dashv, b_f)\). Then there exists an \(i\) such that \(C[e_j], [] \rightarrow^* (\dashv, b_f)\). By Lemma \([2.11]\) it follows that \(\dashv; \dashv \vdash C[e_j] \leq_{\log} C[e_j] : 1\) and thus \((i + 1, C[e_j], C[e_j]) \in \mathcal{E}(\mathcal{V}[\dashv; \dashv ; \vdash 1])([]).\) By definition of \(\mathcal{E}\) this gives us \(v_{\tau}, b_{\tau}\) and \(W\) such that \(C[e_j], [] \rightarrow^* v_{\tau}, b_{\tau}\) and \((1, (0), v_{\tau}) \in \mathcal{V}[\dashv; \dashv ; \vdash 1](W)\). From the latter we obtain \(v_{\tau} = (0)\), which ends the proof.

**Example 2.13** (Counter). In this example, we first present two counter modules, and then we show that they are equivalent. The type of the two modules is as follows:

\[
\tau = \exists x. (1 \rightarrow x) \times (x \rightarrow 1) \times (x \rightarrow \mathbb{N})
\]

The type is an existential over a triple. The first part of the triple is a function that constructs a new counter. The second part is a function that takes a counter and increments it by one. The last part is a function that given a counter produces the current count of the counter. The two counter modules we consider are:

\[
M_1 = \text{pack} (\text{fix new}(\_). \text{ref} 0, \text{fix inc}(l). l := \text{!}l + 1, \text{fix get}(l). \text{!}l)
\]

\[
M_2 = \text{pack} (\text{fix new}(\_). \text{ref} 0, \text{fix inc}(l). l := \text{!}l - 1, \text{fix get}(l). \text{abs}(!l))
\]

where \(\text{abs}\) is a function that given an integer returns the absolute value. We now want to show the following

\[
\bullet \bullet \models M_1 \leq_{\log} M_2 : \tau.
\]
To show that they are equivalent, we also need to show it the other way around, but we leave that for the interested reader.

First, recall that our semantic types are defined as

$$\text{Type} = \text{World} \overset{\text{\text{Val}}} \rightarrow \text{UPred} (\text{Val} \times \text{Val})$$

In our proof, we need to pick a semantic type $\nu$ that captures the invariant that the two values denoted by the two counters are “equal”.

$$\nu = \lambda W. \{ (n, l_1, l_2) \mid \exists v \in \text{dom}(W). \xi(W(v)) \overset{\text{inv}}{=} \text{inv}(R, l_1, l_2) \}$$

where $R \in \text{Type}$ is defined as

$$R(W) = \{(m, n, -n) \mid n \in \mathbb{N} \}.$$  

Now consider the definition of our logical relation, we first let $n''$ and $W$ be given. Next, we are given fitting substitutions to close of the expressions, but the expressions are already closed, so the substitutions have no effect. All in all, we need to show

$$(n', M_1, M_2) \in \mathcal{E}[\bullet \vdash \tau]_g(W).$$

There is not much to do here as $M_1$ and $M_2$ are values, so the two expressions take zero steps and do not modify the heap. So let $n'' \leq n'$, $b_{E_1}$, $b_{E_2}$, and $W' \geq W$ be given where

$$n'' \geq 0 \quad \quad \quad (n'', b_{E_1}, b_{E_2}) \in [W'].$$

we then now need to show

$$(n, M_1, M_2) \in \mathcal{V}[\bullet \vdash \tau]_g(W').$$

Taking a look at the definition of the value relation for existential types, we get to pick a semantic type which relates values of type $\alpha$. In our case $\alpha$ is a location which points to the count, and the $\nu$ we described above relates locations that denote the same count. If we pick $\nu$ as the semantic type, then it suffices to show:

$$(n'', v_1, v_2) \in \mathcal{V}[\alpha \vdash (1 \rightarrow \alpha) \times (\alpha \rightarrow 1) \times (\alpha \rightarrow \mathbb{N})](W').$$

If $n''$ is assumed to be greater than 0 so we can safely write $n = n'' - 1$. By the definition of the value relation for pairs (here for convenience extended directly to triples), we need to show the following three things:

$$(n, \text{fix new}(\_). \ \text{ref} \ 0, \text{fix new}(\_). \ \text{ref} \ 0) \in \mathcal{V}[\alpha \vdash 1 \rightarrow \alpha]_{[\alpha \vdash \nu]}(W')$$

(6)

$$(n, \text{fix inc}(l). \ l := !l + 1, \text{fix inc}(l). \ l := !l - 1) \in \mathcal{V}[\alpha \vdash \alpha \rightarrow 1]_{[\alpha \vdash \nu]}(W')$$

(7)

$$(n, \text{fix get}(l). \ !l, \text{fix get}(l). \ \text{abs}(!l)) \in \mathcal{V}[\alpha \vdash \alpha \rightarrow \mathbb{N}]_{[\alpha \vdash \nu]}(W')$$

(8)

Let us first consider [6]. Let $m < n$ and $W'' \geq W'$ be given. Assume that we have two related arguments for the function, that is $(m, v_1, v_2) \in \mathcal{V}[\alpha \vdash 1]_{[\alpha \vdash \nu]}(W'')$ for some $v_1$ and $v_2$. This immediately gives us $v_1 = v_2 = ()$ by the definition of value relation for the unit type. We then need to show that applying the functions to the arguments gives related results:

$$(m + 1, \text{fix new}(\_). \ \text{ref} \ 0) () (\text{fix new}(\_). \ \text{ref} \ 0) () \in \mathcal{E}[\mathcal{V}[\alpha \vdash \alpha]_{[\alpha \vdash \nu]}](W'').$$

18
To do this we look at the definition at expression closure, so let \( m' \leq m + 1, i < m, b_1, b_2, b'_1 \), and \( W^{(3)} \geq W'' \) be given and assume
\[
(m', b_1, b_2) \in [W^{(3)}] \\
(f x \ n e w (\_). \ ref \ 0), b_1 \rightarrow i' v'_1, b'_1.
\]
From the operational semantics, we have that \((f x \ n e w (\_). \ ref \ 0), b_1 \) will evaluate to some location \( l_1 \) where \( l_1 \notin \text{dom}(b_1) \) and the result heap will be \( b'_1 = b_1[l_1 \mapsto 0] \). We then need to show that there exists \( v'_2, b'_2 \), and \( W^{(4)} \geq W^{(3)} \) such that
\[
(m' - i, b'_1, b'_2) \in [W^{(4)}] \\
\begin{align*}
(f x \ n e w (\_). \ ref \ 0), b_2 & \rightarrow i' v'_1, b'_1 \\
(m' - i, v'_1, v'_2) & \in V[\alpha \vdash \alpha]_{[\alpha \rightarrow \nu]}(W^{(4)}).
\end{align*}
\]
Pick \( v'_2 = i_2, b'_2 = b_2[l_2 \mapsto 0] \) and \( W^{(4)} = W^{(3)}[\xi \mapsto \xi^{-1}(\text{inv}(R, l_1, l_2)) \xi] \) where \( l_2 \notin \text{dom}(b_2) \) and \( i \notin \text{dom}(W^{(3)}) \). Notice that we allocated new references, so it will not suffice to just pick \( W^{(3)} \) as \( W^{(4)} \) because it only relates the old references. Instead we have to extend \( W^{(3)} \) with a new invariant that relates the new references. \( W^{(4)} \) is clearly a future world of \( W^{(3)} \) as it contains the same invariants and an additional one. \( v'_i \) is \( l_1 \) which is a value as it is a location. From the operational semantics we see that \((f x \ n e w (\_). \ ref \ 0), b_2 \rightarrow i' l_2, b'_2 \) is okay. To show \((m' - i, v'_1, v'_2) \in [W^{(4)}]\) we use Lemma 2.2 so it suffices to show
\[
(m' - i, b_1, b_2) \in [W^{(4)}]_{\nu \rightarrow \xi} \\
(m' - i, l_1 \mapsto 0; l \mapsto \nu) \in [W^{(4)}].
\]
We have \((9)\) from our assumption \((m', b_1, b_2) \in [W^{(3)}]\) and downwards closure. For \((10)\) to be true \( l_1 \) and \( l_2 \) has to be related in \( R \) which they are as \( 0 = -0 \). It remains to show that \( l_1 \) and \( l_2 \) are related:
\[
(m' - i, l_1, l_2) \in v(W^{(4)}).
\]
To show this we pick \( \xi \) as a witness which means we have to show
\[
\xi(W^{(4)}(i)) = \text{inv}(R, l_1, l_2).
\]
\( \xi(W^{(4)}(i)) \) is \( \xi(\xi^{-1}(\text{inv}(R, l_1, l_2))) \) which is exactly the same as the right-hand side of the above equation. If the two are equal, then they are also \( m' - i \)-equal.

Now we show \((7)\). Let \( m < n \), \( W'' \geq W' \) be given and assume
\[
(m, v_1, v_2) \in V[\alpha \vdash \alpha]_{[\alpha \rightarrow \nu]}(W'')
\]
where \( v_1 \) and \( v_2 \) are some values. We then need to show
\[
(m, (f x \ i n c (l). l := !l + 1) \ v_1, (f x \ i n c (l). l := !l - 1) \ v_2) \in E[V[\alpha \vdash \alpha]]_{[\alpha \rightarrow \nu]}(W'').
\]
Two \) are trivially related, so this may seem simple to show if we can show that the two expressions reduce to \( (\_). \). However, we also need to show that the result heaps are related, or in other words assuming the initial heaps are related, then the invariants have to be preserved by the evaluation. Let \( m' \leq m, i < m', b_1, b_2, b'_1, e'_1, \) and \( W^{(3)} \geq W'' \) be given. Further, assume
\[
(m', b_1, b_2) \in [W^{(3)}] \\
(f x \ i n c (l). l := !l + 1) \ v_1, b_1 \rightarrow i' e'_1, b'_1.
\]
Then, we need to show that there exists \( v'_1, h_2, \) and \( W^{(4)} \geq W^{(3)} \) that satisfy the following four conditions:

\[
\begin{align*}
e'_1 & \in \text{Val} \\
(\text{fix inc}(l), l := !l - 1) & v_2, h_2 \rightarrow^* v'_2, h'_2 \tag{13} \\
(m' - i, e'_1, v'_2) & \in V[1]_{[2 \rightarrow n]}(W^{(4)}) \tag{14} \\
(m' - i, h'_1, h'_2) & \in [W^{(4)}]. \tag{15}
\end{align*}
\]

Before we can show the above, we need to consider what assumption \( 11 \) entails. If \( v_1 \) and \( v_2 \) are related at time \( m \), then due to downwards closure they must also be related at time \( m' \). Further due to monotonicity, \( v_1 \) and \( v_2 \) are also related in future worlds, specifically \( W^{(3)} \). All in all, we end up with

\[
(m', v_1, v_2) \in V[\alpha \vdash x]_{[2 \rightarrow n]}(W^{(3)}).
\]

\( v \) relates locations, so \( v_1 \) and \( v_2 \) must be locations. To reflect this, we will henceforth refer to \( v_1 \) and \( v_2 \) as \( l_1 \) and \( l_2 \) respectively. By definition of \( v \), there must exist an invariant name \( \xi \) in the domain of \( W^\pi \) such that

\[
\xi(W^{(3)}(\xi)) \models \text{Inv}(R, l_1, l_2).
\]

We now use Lemma 2.2 and (13) to conclude:

\[
l_1 \in \text{dom}(b_1) \quad l_2 \in \text{dom}(b_2) \quad (m' - 1, b_1(l_1), b_2(l_2)) \in R \ W^{(3)}
\]

The last property can be unfolded to \( b_1(l_1) = -b_2(l_2) \). Now, we can say a bit more about reduction \( 12 \). As \( l_1 \in \text{dom}(b_1) \), we know that the configuration want get stuck. It will en three steps reduce to \(((), b_2[l_1 = b_1(l_1) + 1], b'_1 = b_1[l_1 = b_1(l_1) + 1] \).

Now we are ready to show the proof goals we originally set out to show. Pick \( \epsilon' = () \), \( h'_1 = b_2[l_2 = b_2(l_2) - 1] \), and \( W^{(4)} = W^{(3)} \). Notice, no new references are allocated, so we do not need to introduce any new invariants and can thus pick the world were in for \( W^{(4)} \). As \( e'_1 \) is the unit value it is indeed a value, so \( 13 \) is indeed the case. From the operational semantics and the fact that \( l_2 \in \text{dom}(b_2) \), we also have that \( 14 \) is satisfied. \( 15 \) is trivially satisfied as the unit value is always related to itself. It now remains to show \( 16 \) that is the resulting heaps, \( h'_1 \) and \( h'_2 \), should be related at time \( m' - i \) in world \( W^{(3)} \). We have not touched any locations but \( l_1 \) and \( l_2 \), so it suffices to show that the invariant is satisfied for those locations. This boils down to showing

\[
b_1(l_1) + 1 = -b_2(l_2) - 1
\]

which is true as we had \( b_1(l_1) = -b_2(l_2) \). This is due to the assumption that we started out with related heaps in a previous world which especially means that the values at location \( l_1 \) and \( l_2 \) were related in that world.

The last and possibly easiest case is \( 8 \). Here we do not modify the heap in anyway, so it is easy to show the invariants are kept. However, we do rely on the assumption that our starting heaps satisfy the invariant to show the results are related. Let \( m < n \), \( W'' \geq W' \) be given and assume

\[
(m, v_1, v_2) \in V[\alpha \vdash x]_{[2 \rightarrow n]}(W'') \tag{17}
\]

where \( v_1 \) and \( v_2 \) are some values. We then need to show

\[
(m, (\text{fix get}(l), !l) v_1, (\text{fix get}(l), \text{abs}(l)) v_2) \in E[V[N]_{[2 \rightarrow n]}](W'')
\]

\[\text{Note: For brevity, we skipped a bit of argumentation here. Lemma 2.2 splits the heap, so the properties are only for parts of the heap, but it can be extended to the entire heap. The reader is encouraged show this is in fact the case.}\]
Let $m' \leq m$, $i < m'$, $h_1$, $b_1$, $h_1'$, and $W^{(3)} \geq W''$ be given. Further, assume

$$m' > 0$$

$$(m', h_1, b_2) \in \{ W^{(3)} \}$$

$$(\text{fix get}(l). \,!) \, v_1, h_1 \to^1 e'_1, h'_1.$$  

(18)

Then, we need to show that there exists $v'_2$, $b_2$, and $W^{(4)} \geq W^{(3)}$ that satisfy the following four things:

$$e'_1 \in \text{Val}$$

$$(\text{fix get}(l). \text{abs}(!l)) \, v_2, b_2 \to^* v'_2, b'_2$$  

(19)

$$(m' - i, e'_1, v'_2) \in V[\mathbb{N}]_{[\alpha \mapsto \nu]}(W^{(4)})$$  

(20)

$$(m' - i, h'_1, h'_2) \in \{ W^{(4)} \}.$$  

(21)

So far, we have done exactly the same as in the previous case with the increment functions. We are going to use the same argumentation as before to get $v_1 = l_1$, $v_2 = l_2$, $l_1 \in \text{dom}(h_1)$, $l_2 \in \text{dom}(h_2)$, and $h_1(l_1) = -h_2(l_2)$ for locations $l_1$ and $l_2$. We further define $k$ to be $h_1(l_1)$ to emphasize that it is a value and in particular a natural number. Now from the operational semantics and $l_1 \in \text{dom}(h_1)$, we have

$$(\text{fix get}(l). \,!) \, l_1, b_1 \to^2 k, b_1$$

which means $e'_1 = k$ which is a natural number and thus a value. Now pick $v'_2 = k$, $b'_2 = b_2$, and $W^{(3)} = W^{(4)}$, then we have the evaluation

$$(\text{fix get}(l). \text{abs}(!l)) \, l_2, b_2 \to^* \text{abs}(-k), b_2$$

$$\to^* k, b_2$$

which satisfies [19, 21] is satisfied by the assumption that the initial heaps are related and downwards closure of heap satisfaction. Finally,

$$(m' - i, k, k) \in V[\mathbb{N}]_{[\alpha \mapsto \nu]}(W^{(3)})$$

is trivially satisfied.

This concludes the proof of the equivalence of the two counter modules. ■

Acknowledgments

Following the proud tradition from the previous lecture note, I accept no responsibility for any mistakes in this document. If you find anything amiss, please let me know so I can figure who of the following is the culprit: Lars Birkedal, Aleš Bizjak, Kasper Svendsen, Mathias Rav, or Kent Grigo.

References


